Formation PERF1

PostgreSQL Performances

24.09

Table des matières

Sur ce document

Formation	Formation PERF1
Titre	PostgreSQL Performances
Révision	24.09
ISBN	978-2-38168-122-1
PDF	https://dali.bo/perf1_pdf
EPUB	https://dali.bo/perf1_epub
HTML	https://dali.bo/perf1_html
Slides	https://dali.bo/perf1_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

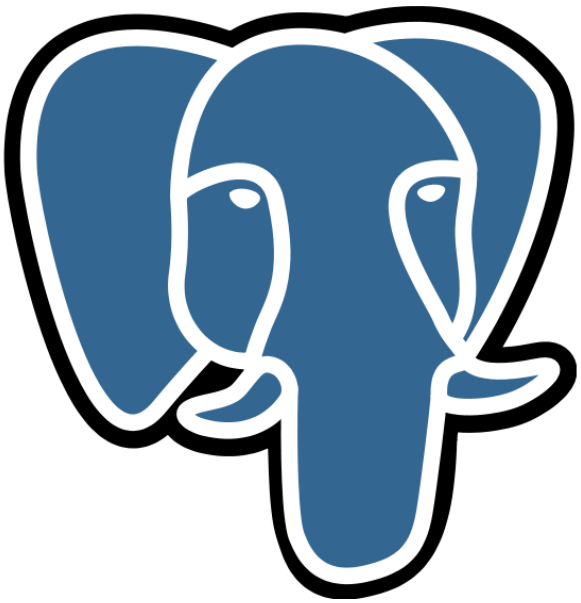
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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

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

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

1/ Configuration du système et de l’instance



1.1 INTRODUCTION

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

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

L'optimisation (aussi appelée *tuning*) doit donc être réalisée sur tous ces éléments **à la fois** pour être optimale !

1.1.1 Menu

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

1.1.2 Considérations générales - 1

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

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

1.1.3 Considérations générales - 2

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

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

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

Le plus important est donc de bien comprendre qu'un SGBD ne s'optimise pas qu'une seule fois, mais que ce travail d'optimisation sera à faire plusieurs fois au fur et à mesure de la vie du serveur.

À une échelle beaucoup plus petite, un travail d'optimisation sur une requête peut forcer à changer la configuration d'un paramètre. Cette modification peut faire gagner énormément sur cette requête… mais faire perdre encore plus sur les autres. Là aussi, tout travail d'optimisation doit être fait prudemment et ses effets surveillés sur une période représentative pour s'assurer que cette amélioration ne s'accompagne pas de quelques gros inconvénients.

1.2 MATÉRIEL

- Performances très liées aux possibilités du matériel et de l'OS
- 4 composants essentiels
  - les processeurs
  - la mémoire
  - les disques
  - le système disque (RAID, SAN)

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

Au niveau du matériel, les composants essentiels sont :

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

1.2.1 CPU

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

PostgreSQL est un système multiprocessus. Chaque connexion d'un client est gérée par un processus, responsable de l'exécution des requêtes et du renvoi des données au client.

Ces processus ne sont pas multithreadés. Par conséquent, chaque requête exécutée est généralement traitée par un seul processus, sur un cœur de processeur. Mais dans certains cas, d'autres processus peuvent intervenir sur la même requête pour utiliser d'autres cœurs. Lorsque la requête est en lecture

seule (et dans quelques rares cas en écriture) et que la parallélisation est activée, ce processus peut être aidé le temps de l'exécution de certains nœuds par un ou plusieurs processus appelés *workers*. Les détails figurent plus loin.

Parallélisation mise à part, plus vous voulez pouvoir exécuter de requêtes en simultané, plus vous devez avoir de processeurs (ou plus exactement de cœurs). On considère habituellement qu'un cœur peut traiter de 1 à 20 requêtes simultanément. Cela dépend notamment beaucoup des requêtes, de leur complexité, de la quantité de données manipulée et retournée, etc. Il est donc essentiel de connaître le nombre de requêtes traitées simultanément pour le nombre d'utilisateurs connectés.

S'il s'agit d'une instance pour une application web, il y a de fortes chances que le nombre de requêtes (simples) en parallèle soit assez élevé. Dans ce contexte, il faut prévoir un grand nombre de cœurs ou de processeurs. Par contre, sur un entrepôt de données, il y a généralement peu d'utilisateurs, mais des requêtes complexes et gourmandes en ressources, sur de gros jeux de données, mais ces requêtes sont, à priori, facilement parallélisables. Il est alors possible d'opter pour des processeurs avec une fréquence plus élevée (qui ont souvent moins de cœurs), mais plus le système aura de cœurs, plus sa capacité à pouvoir paralléliser les requêtes qui s'y prêtent sera élevé. Ainsi, la fréquence (et donc la puissance) des processeurs est un point important à considérer. Il peut faire la différence pour des requêtes complexes : temps de planification réduit, calculs plus rapides donc plus de requêtes exécutées sur une période de temps donnée.

Généralement, un système utilisé pour des calculs (financiers, scientifiques, géographiques) a intérêt à avoir des processeurs à fréquence élevée.

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

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

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

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

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

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

1.2.2 RAM

- Essentielle pour un serveur de bases de données
- Plus il y en a, mieux c'est
  - moins d'accès disque
- Pour le système comme pour PostgreSQL

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

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

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

1.2.3 Disques

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

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

Il existe actuellement quatre types de modèles de disques :

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

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

Il est immédiatement visible que la mémoire est toujours imbattable, y compris face aux disques SSD avec un facteur 10 000 en performance de temps d'accès entre les deux ! À l'autre bout de l'échelle se trouvent les disques magnétiques avec interface SATA. Leur faible performance en temps d'accès ne doit pas pour autant les disqualifier : leur prix est imbattable et il est souvent préférable de prendre un grand nombre de disques pour avoir de bonnes performances. Cependant, la fiabilité des disques SATA impose de les considérer comme des consommables et de toujours avoir des disques de secours prêts à remplacer une défaillance.

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

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

Le gros intérêt des disques SSD (et encore plus NVMe) est un temps d'accès très rapide. Ils se démarquent des disques magnétiques (SAS comme SATA) par une durée d'accès à une page aléatoire aussi rapide que celle à une donnée contiguë (ou séquentielle). C'est parfait pour accéder à des index.

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

De nos jours, ce n'est plus tellement le cas grâce à des algorithmes d'écriture complexes permettant d'atteindre des durées de vie équivalentes, voire plus importantes, que celles des disques magnétiques. Néanmoins, ces mêmes algorithmes mettent en péril la durabilité des données en cas d'interruption brutale.

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

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

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

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

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

Les disques bas de gamme mais rapides peuvent néanmoins servir à stocker des données volatiles, comme les fichiers temporaires pour le tri et le hachage, ainsi que les tables et index temporaires.

Il est possible de configurer le système d'exploitation pour optimiser l'utilisation des SSD. Par exemple sous Linux, les deux optimisations courantes dans le noyau sont :

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

1.2.4 RAID

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

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

Pour les performances en écriture, il est généralement préférable de se baser sur du RAID 10, soit deux grappes de disques en RAID 1 (en miroir) agrégés dans un RAID 0 : c'est tout aussi intéressant en termes de fiabilité, mais avec de bien meilleures performances en lecture et écriture. En contrepartie,

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

à volumétrie égale, il nécessite plus de disques et est donc beaucoup plus cher que le RAID 5. Pour réduire le budget, il peut être envisageable de choisir des disques SATA en RAID 10. Cela dit, un RAID 5 peut très bien fonctionner pour votre application. Le plus important est d'obtenir un RAID fiable avec de nombreux disques, surtout s'ils sont magnétiques.

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

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

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

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

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

La majorité des cartes RAID offre maintenant un système de cache de données en mémoire. Ce cache peut être simplement en lecture ou en lecture/écriture. En lecture, il faut évidemment toujours l'activer.

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

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

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

1.2.5 SAN

- Pouvoir sélectionner les disques dans un groupe RAID
- Attention au cache
  - toujours activer le cache en lecture
  - activer le cache en écriture que si batterie présente
- Attention à la latence réseau !
- Attention au système de fichiers
  - NFS : risques de corruptions et problèmes de performance

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

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

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

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

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

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

Dans un registre moins coûteux, la tentation est grande d'utiliser un simple NAS[3], avec par exemple

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

un accès NFS aux partitions. Il faut l'éviter, pour des raisons de performance et de fiabilité. Utilisez plutôt iSCSI, peu performant, mais plus fiable et moins complexe.

1.2.6 Virtualisation

- Masque les ressources physiques au système
  - plus difficile d'optimiser les performances
- Propose généralement des fonctionnalités d'*overcommit*
  - grandes difficultés à trouver la cause du problème du point de vue de la VM
  - dédier un minimum de ressources aux VM PostgreSQL
- En pause tant que l'hyperviseur ne dispose pas de l'ensemble des vCPU alloués à la machine virtuelle (*steal time*)
- Mutualise les disques = problèmes de performances
  - préférer attachement direct au SAN
  - disques de PostgreSQL en *Thick Provisionning*

L'utilisation de machines virtuelles n'est pas recommandée avec PostgreSQL. En effet, la couche de virtualisation cache totalement les ressources physiques au système, ce qui rend l'investigation et l'optimisation des performances beaucoup plus difficiles qu'avec des serveurs physiques dédiés.

Il est néanmoins possible, et très courant, d'utiliser des machines virtuelles avec PostgreSQL. Leur configuration doit alors être orientée vers la stabilité des performances. Cette configuration est complexe et difficile à suivre dans le temps. Les différentes parties de la plate-forme (virtualisation, système et bases de données) sont généralement administrées par des équipes techniques différentes, ce qui rend le diagnostic et la résolution de problèmes de performances plus difficiles. Les outils de supervision de chacun sont séparés et les informations plus difficiles à corréler.

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

Il est généralement conseillé d'utiliser au moins 4 cœurs physiques. En fonction de la complexité des requêtes, du volume de données, de la puissance du CPU, un cœur physique sert en moyenne de 1

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

Le point le plus négatif de la virtualisation de serveurs de bases de données concerne la performance des disques. La mutualisation des disques pose généralement des problèmes de performances car les disques sont utilisés pour des profils d'I/O généralement différents. Le RAID 5 est réputé offrir le meilleur rapport performance/coût… sauf pour les bases de données, qui effectuent de nombreux accès aléatoires. De ce fait, le RAID 10 est préconisé car il est plus performant sur les accès aléatoires en écriture pour un nombre de disques équivalent. Avec la virtualisation, peu de disques, mais de grande capacité, sont généralement prévus sur les hyperviseurs ; or cela implique un coût supérieur pour l'utilisation de RAID 10 et des performances inférieures sur les SGDB qui tirent de meilleures performances des disques lorsqu'ils sont nombreux.

Enfin, les solutions de virtualisation effectuent du *Thin Provisioning* sur les disques pour minimiser les pertes d'espace. Pour cela, les blocs sont alloués et initialisés à la demande, ce qui apporte une latence particulièrement perceptible au niveau de l'écriture des journaux de transaction (in fine, cela détermine le nombre maximum de commits en écriture par seconde possible). Il est donc recommandé de configurer les disques de PostgreSQL en *Thick Provisionning*.

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

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

1.2.7 Virtualisation : les bonnes pratiques

- Éviter le *time drift* : même source NTP sur les VM et l'ESXi
- Utiliser les adaptateurs réseau paravirtualisés de type VMXNET3
- Utiliser l'adaptateur paravirtualisé PVSCSI pour les disques dédiés aux partitions PostgreSQL
- Si architecture matérielle NUMA :
  - dimensionner la mémoire de chaque VM pour qu'elle ne dépasse pas le volume de mémoire physique au sein d'un groupe NUMA

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

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

Un aspect très important de la configuration de la mémoire des machines virtuelles est l'accès mémoire non uniforme (NUMA). Cet accès permet d'accélérer l'accès mémoire en partitionnant la mémoire physique de telle sorte que chaque socket dispose de sa propre mémoire. Par exemple, avec un système à 2 sockets et 128 Go de RAM, chaque socket ou nœud possède 64 Go de mémoire physique.

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

Pour approfondir : Fiche KB préconisations pour VMWare[4]

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

1.3 SYSTÈME D'EXPLOITATION

- Quel système choisir ?
- Quelle configuration réaliser ?

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

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

1.3.1 Choix du système d'exploitation

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

PostgreSQL est écrit pour être le plus portable possible. Un grand nombre de choix dans son architecture a été fait en fonction de cette portabilité. La liste des plate-formes officiellement supportées[5]) comprend donc Linux, FreeBSD, OpenBSD, macOS, Windows, Solaris, etc. Cette portabilité est vérifiée en permanence avec la ferme de construction (*BuildFarm*[6]), qui comprend même de vieilles versions de ces systèmes sur différentes architectures avec plusieurs compilateurs.

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

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

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

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

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

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

1.3.2 Choix du noyau

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

Il est préférable de ne pas fonctionner avec une très ancienne version du noyau Linux. Les dernières versions sont les plus stables, les plus performantes, les plus compatibles avec les derniers matériels. 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.

[7] http://support.microsoft.com/kb/184802
[8] https://techcommunity.microsoft.com/t5/ask-the-performance-team/sessions-desktops-and-windows-stations/bap/372473
[9] https://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows#I_cannot_run_with_more_than_about_125_connections_at_once.2C_despite_having_capable_hardware

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

1.3.3 Configuration du noyau

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

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

1.3.4 Contrôle du cache disque système

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

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

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

alors un encombrement de la bande passante des disques dans les situations de forte charge en écriture, surtout lors des opérations provoquant des synchronisations de données modifiées en cache sur le disque, comme l'appel à `fsync`. Celui-ci est utilisé par PostgreSQL lors des *checkpoints*, ce qui peut provoquer des latences supplémentaires à ces moments-là.

Sur les versions de PostgreSQL précédant la 9.6, pour réduire les conséquences de ce phénomène, il était systématiquement conseillé d'abaisser `vm.dirty_ratio` à 10 et `vm.dirty_background_ratio` à 5. Ainsi, lors de fortes charges en écriture, le noyau reporte plus fréquemment son cache disque sur l'espace de stockage, mais pour une volumétrie plus faible, et l'encombrement de la bande passante vers les disques est moins long si ceux-ci ne sont pas capables d'absorber ces écritures rapidement. Dans les situations où la quantité de mémoire physique est importante, ces paramètres peuvent même être encore abaissés à 2 et 1 respectivement. Avec 32 Go de RAM, cela donne encore 640 Mo et 320 Mo de données à synchroniser, ce qui peut nécessiter plusieurs secondes d'écritures en fonction de la configuration disque utilisée.

Dans les cas plus extrêmes, 1 % de la mémoire représente une volumétrie trop importante (par exemple, 1,3 Go pour 128 Go de mémoire physique). Les paramètres `vm.dirty_bytes` et `vm.dirty_background_bytes` permettent alors de contrôler ces mêmes comportements, mais en fonction d'une quantité de *dirty pages* exprimée en octets et non plus en pourcentage de la mémoire disponible. Notez que ces paramètres ne sont pas complémentaires entre eux. Le dernier paramètre ayant été positionné prend le pas sur le précédent.

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

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

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

Mais ces derniers paramètres ne concernent que les processus de PostgreSQL. Or PostgreSQL n'est pas seul à écrire de gros fichiers (exports `pg_dump`, processus d'archivage, copies de fichiers…). Le paramétrage des `vm.dirty_bytes` conserve donc un intérêt.

1.3.5 Configuration du swap

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

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

```
vm.swappiness = 10
```

Le *swap* n'est plus que rarement utilisé sur un système moderne, et 2 Go suffisent amplement en temps normal. Avoir trop de *swap* a tendance à aggraver la situation dans un contexte où la mémoire devient rare : le système finit par s'effondrer à force de swapper et dé-swapper un nombre de processus trop élevé par rapport à ce qu'il est capable de gérer.

Il est utile d'en conserver un peu pour swapper des processus inactifs, ou le contenu de systèmes de fichiers `tmpfs` (classiquement, `/var/run`) et les journaux de `systemd-journald` selon la configuration de celui-ci. Ne pas avoir de *swap* amène encore un autre problème : cela ne permet pas de s'apercevoir d'une surconsommation de mémoire. Il convient donc de créer un espace de *swap* de 2 Go au plus sur la machine.

Le paramètre `vm.swappiness` contrôle le comportement du noyau vis-à-vis de l'utilisation du *swap*. Plus ce pourcentage est élevé, plus le système a tendance à swapper facilement. Un système hébergeant une base de données ne doit swapper qu'en dernière extrémité. La valeur par défaut (30 ou 60 suivant les distributions) doit donc être abaissée à 10 pour éviter l'utilisation du *swap* dans la majorité des cas.

1.3.6 Configuration de la sur-réservation mémoire

Le noyau peut autoriser trop de réservation mémoire.

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

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

Danger de l'overcommit :

Certaines applications réservent (*commit*) auprès du noyau plus de mémoire que nécessaire. Plusieurs optimisations noyau permettent aussi d'économiser de l'espace mémoire. Ainsi, par défaut, le noyau Linux s'autorise à réserver aux processus plus de mémoire qu'il n'en dispose réellement, le risque de réellement utiliser cette mémoire étant faible. On appelle ce comportement l'*Overcommit Memory*. Ce peut être intéressant dans certains cas d'utilisation, mais peut devenir dangereux dans le cadre d'un serveur PostgreSQL dédié, qui va réellement allouer (utiliser) toute la mémoire qu'il réservera. Typiquement, cela arrive lors de tris en mémoire trop gros, ou trop nombreux au même moment.

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

Configuration de vm.overcommit_memory et vm.overcommit_ratio :

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

Pour désactiver complètement l'*overcommit memory* :

```
vm.overcommit_memory = 2
```

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

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

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

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

```
vm.overcommit_ratio = 75
```

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

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

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

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

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

Exemple :

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

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

Extrait de la configuration mémoire résultante :

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

dalibo@srv-psql-02:~$ cat /proc/meminfo
MemTotal:       16473548 kB
MemFree:          178432 kB
Buffers:           67260 kB
…
SwapTotal:       1740796 kB
SwapFree:        1740696 kB
…
CommitLimit:    15743308 kB
Committed_AS:    6436004 kB
…
```

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

Avec des *huge pages* :

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

Exemple de saturation mémoire :

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

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

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

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

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

Sans paramétrage, ce serait plus brutal :

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

et les traces indiqueraient ceci avant le redémarrage :

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

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

Pour plus de détails :

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

1.3.7 Huge pages

Pages mémoire de 2 Mo au lieu de 4 ko :

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

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

- Noyau :

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

- *Transparent Huge Pages* : à désactiver

Principe des *huge pages* :

Les systèmes d'exploitation utilisent un système de mémoire virtuelle : chaque contexte d'exécution (comme un processus) utilise un plan d'adressage virtuel, et c'est le processeur qui s'occupe de réaliser la correspondance entre l'adressage virtuel et l'adressage réel. Chaque processus fournit donc la

correspondance entre les deux plans d'adressage, dans ce qu'on appelle une « table de pagination ». Les processeurs modernes permettent d'utiliser plusieurs tailles de page mémoire simultanément. Pour les processeurs Intel/AMD, les tailles de page possibles sont 4 ko, 2 Mo et 1 Go.

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

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

```
PageTables:      1193040 kB
```

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

Paramétrer PostgreSQL pour les *huge pages* :

Dans `postgresql.conf`, le défaut convient :

```
huge_pages = try
```

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

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

Paramétrer les *huge pages* au niveau noyau :

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

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

En conséquence, si `shared_buffers` vaut 8 Go :

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

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

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

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

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

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

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

Désactivation des Transparent Huge Pages

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

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

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

ou encore dans la configuration de `grub` :

```
transparent_hugepage=never
```

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

1.3.8 Configuration de l'affinité processeur / mémoire

- Pour architecture NUMA (multisocket)
- Chaque socket travaille plus efficacement avec une zone mémoire allouée
- Peut pénaliser le cache disque système
  - `vm.zone_reclaim_mode` : passer à 0

Attention, ne pas confondre multicœur et multisocket ! Chaque processeur physique occupe un socket et peut contenir plusieurs cœurs. Le nombre de processeurs physiques peut être trouvé grâce au

nombre d'identifiants dans le label `physical id` du fichier `/proc/cpuinfo`. Par exemple, sur un serveur bi-processeur :

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

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

```
root@serveur:~# lscpu | grep -i socket
Cœur(s) par socket :                   2
Socket(s) :                            1
```

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

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

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

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

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

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

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

Pour plus de détails, voir :

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

1.3.9 Configuration de l’ordonnanceur

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

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

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

Sans regroupement de processus :

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

Avec regroupement de processus :

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

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

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

1.3.10 Comment les configurer

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

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

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

```
# sysctl --system
```

de consulter les valeurs avec :

```
# sysctl -a
```

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

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

1.3.11 Choix du système de fichiers

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

Quel que soit le système d'exploitation, les systèmes de fichiers ne manquent pas. Linux en est la preuve avec pas moins d'une dizaine de systèmes de fichiers. Le choix peut paraître compliqué mais il se révèle fort simple : il est préférable d'utiliser le système de fichiers préconisé par votre distribution Linux. Ce système est à la base de tous les tests des développeurs de la distribution : il a donc plus de chances d'avoir moins de bugs, tout en proposant plus de performances. Les instances de production PostgreSQL utilisent de fait soit ext4, soit XFS, qui sont donc les systèmes de fichiers recommandés.

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

Autrefois réservé à Solaris, ZFS est un système très intéressant grâce à son panel fonctionnel et son mécanisme de *Copy On Write* permettant de faire une copie des fichiers sans arrêter PostgreSQL (*snapshot*). OpenZFS[13], son portage sous Linux/FreeBSD, entre autres, est un système de fichiers proposant un panel impressionnant de fonctionnalités (dont : checksum, compression, gestion de snapshot), les performances en écriture sont cependant bien moins bonnes qu'avec ext4 ou XFS. De plus, il est plus complexe à mettre en place et à administrer. Btrfs[14] est relativement répandu et bien intégré à Linux, et offre une partie des fonctionnalités de ZFS ; mais il est également peu performant avec PostgreSQL.

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

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

Par contre, NFS est totalement déconseillé dans les environnements critiques avec PostgreSQL. Greg Smith, contributeur très connu, spécialisé dans l'optimisation de PostgreSQL, parle plus longuement des soucis de NFS avec PostgreSQL[18]. En fait, il y a des dizaines d'exemples de gens ayant eu des problèmes avec NFS. Les problèmes de performance sont quasi-systématiques, et ceux de fiabilité

[12] https://fr.slideshare.net/fuzzycz/postgresql-na-ext4-xfs-btrfs-a-zfs-fosdem-pgday-2016
[13] https://en.wikipedia.org/wiki/OpenZFS
[14] https://en.wikipedia.org/wiki/Btrfs
[15] https://fr.wikipedia.org/wiki/Gestion_par_volumes_logiques
[16] https://kb.dalibo.com/nfs
[17] https://docs.postgresql.fr/current/creating-cluster.html#creating-cluster-nfs
[18] https://www.postgresql.org/message-id/4D2285CF.3050304@2ndquadrant.com

fréquents, et compliqués à diagnostiquer (comme illustré dans ce mail[19], où le problème venait du noyau Linux).

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

1.3.12 Configuration du système de fichiers

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

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

Certaines options sont intéressantes en termes de performances. Ainsi, `noatime` évite l'écriture de l'horodatage du dernier accès au fichier. `nodiratime` fait de même au niveau du répertoire. Depuis plusieurs années maintenant, `nodiratime` est inclus dans `noatime`.

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

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

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

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

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

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

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

[19] https://www.postgresql.org/message-id/4D40DDB7.1010000@credativ.com

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

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

1.3.13 Configuration de l'antivirus

Pas d'antivirus

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

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

1.4 SERVEUR DE BASES DE DONNÉES

- Version
- Configuration
- Emplacement des fichiers

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

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

1.4.1 Version

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

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

Voici quelques exemples frappants :

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

- Le partitionnement déclaratif (introduit en version 10) peut mener à manipuler beaucoup de tables-partitions. Le planificateur gère cela de mieux en mieux dans les dernières versions.

Compiler soi-même PostgreSQL ne permet pas de gagner réellement en performance. Même s'il peut y avoir un gain, ce dernier ne peut être que mineur et difficilement identifiable.

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

- la taille des fichiers recompilés est nettement plus grande ;
- la compilation avec `icc` est moins documentée et moins testée qu'avec `gcc`.

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

1.4.2 Configuration - mémoire partagée

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

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

shared_buffers :

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

Ce cache est en mémoire partagée, et donc commun à tous les processus PostgreSQL. Généralement, il faut lui donner une grande taille, tout en conservant malgré tout la majorité de la mémoire pour le cache disque du système, à priori plus efficace pour de grosses quantités de données.

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

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

```
shared_buffers = 8GB
```

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

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

Le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est moins grave de sous-dimensionner un peu `shared_buffers` que de le sur-dimensionner.

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

Modifier `shared_buffers` impose de redémarrer l'instance.

wal_buffers :

PostgreSQL dispose d'un autre cache disque. Ce dernier concerne les journaux de transactions. Il est généralement bien plus petit que `shared_buffers` mais, si le serveur est multiprocesseur et qu'il y a de nombreuses connexions simultanées au serveur PostgreSQL, il est important de l'augmenter. Le paramètre en question s'appelle `wal_buffers`. Plus cette mémoire est importante, plus les transactions seront conservées en mémoire avant le `COMMIT`. À partir du moment où le `COMMIT` d'une transaction arrive, toutes les modifications effectuées dans ce cache par cette transaction sont enregistrées dans le fichier du journal de transactions.

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

1.4.3 Configuration - mémoire des processus

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

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

- Le paramètre le plus important est `work_mem`, qui définit la taille maximale de la mémoire de travail d'un `ORDER BY`, de certaines jointures, pour la déduplication… que peut utiliser un processus sur un nœud de requête, principalement lors d'opérations de tri ou regroupement.
- Autre paramètre capital, `maintenance_work_mem` qui est la mémoire utilisable pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, `REINDEX`, ajouts de clé étrangère…

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

- Il existe aussi `logical_decoding_work_mem` (défaut : 64 Mo), utilisable pour chacun des flux de réplication logique (s'il y en a, ils sont rarement nombreux).

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go sur les machines récentes, car il concerne des opérations lourdes. Leurs consommations de RAM s'additionnent, mais en pratique ces opérations sont rarement exécutées plusieurs fois simultanément.

Monter au-delà de 1 Go n'a d'intérêt que pour la création ou la réindexation de très gros index.

Paramétrage de work_mem :

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

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

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

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

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

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

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

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

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

> `work_mem` = mémoire / `max_connections`

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

```
work_mem = 80MB
```

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

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

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

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

```
work_mem = 240MB
```

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

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

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

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

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

hash_mem_multiplier :

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

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

1.4.4 Configuration - planificateur

- `effective_cache_size`
- `random_page_cost`

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

Le premier permet d'indiquer la taille totale du cache disque disponible pour une requête. Pour le configurer, il faut prendre en compte le cache de PostgreSQL (`shared_buffers`) et celui du système d'exploitation. Ce n'est donc pas une mémoire que PostgreSQL va allouer, mais plutôt une simple

indication de ce qui est disponible. Le planificateur se base sur ce paramètre pour évaluer les chances de trouver des pages de données en mémoire. Une valeur plus importante aura tendance à faire en sorte que le planificateur privilégie l'utilisation des index, alors qu'une valeur plus petite aura l'effet inverse. Généralement, il se positionne à 2/3 de la mémoire d'un serveur pour un serveur dédié.

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

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

Soit 789 116 Kio, résultat de l'addition de 190 580 (colonne `buffers`) et 598 536 (colonne `cached`). Il faut ensuite ajouter `shared_buffers` à cette valeur.

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

1.4.5 Configuration - parallélisation : principe

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

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

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

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

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

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

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

1.4.6 Configuration - parallélisation : paramètres

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

Paramètre principaux :

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

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

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

Impact de la volumétrie :

Le volume déclencheur dépend pour les tables de la valeur du paramètre `min_parallel_table_scan_size` (8 Mo par défaut) et de celle de `min_parallel_index_scan_size` (512 ko par défaut) pour les index.

Ces paramètres sont rarement modifiés. Le moteur détermine ensuite ainsi le nombre de *workers* à lancer :

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

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

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

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

Exemple :

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

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

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

L'option `VERBOSE` donne plus de détails :

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather (actual time=1983.902..2124.019 rows=100000 loops=1)
   Output: aid, bid, abalance, filler
```

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

Mémoire :

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

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

Paramétrage :

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

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

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

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

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

Création et maintenance d'index :

La création d'index B-tree peut être aussi être parallélisée depuis la version 11. Le paramètre `max_parallel_maintenance_workers`, par défaut à 2, indique le nombre de *workers* utilisables lors de la création d'un index, mais aussi de son nettoyage lors d'un `VACUUM`. Le gain de temps peut être

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

appréciable. Cette opération étant assez rare, le paramètre peut être monté assez haut. Les limites de `max_worker_processes` et `max_parallel_workers` s'appliquent là encore.

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

Référence :

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

1.4.7 Configuration - WAL

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

`fsync` est le paramètre qui assure que les données sont non seulement écrites mais aussi forcées sur disque. En fait, quand PostgreSQL écrit dans des fichiers, cela passe par des appels système pour le noyau qui, pour des raisons de performances, conserve dans un premier temps les données dans un cache. En cas de coupure de courant, si ce cache n'est pas vidé sur disque, il est possible que des données enregistrées par un `COMMIT` implicite ou explicite n'aient pas atteint le disque et soient donc perdues une fois le serveur redémarré, ou pire, que des données aient été modifiées dans des fichiers de données, sans avoir été auparavant écrites dans le journal. Cela entraînera des incohérences dans les fichiers de données au redémarrage. Il est donc essentiel que les données enregistrées dans les journaux de transactions soient non seulement écrites, mais que le noyau soit forcé de les écrire réellement sur disque. Cette opération s'appelle `fsync`, et est activé par défaut (`on`). C'est essentiel pour la fiabilité, même si cela impacte très négativement les performances en écriture en cas de nombreuses transactions. Il est donc obligatoire en production de conserver ce paramètre activé. Pour accélérer de très gros imports ou restaurations, on peut le passer exceptionnellement à `off`, et l'on n'oubliera pas de revenir à `on`.

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

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

Tout surplus d'activité doit aussi être géré. Un surplus d'activité engendrera des journaux de transactions supplémentaires. Le meilleur moyen dans ce cas est de préciser au bout de quelle quantité de journaux générés il faut lancer un `CHECKPOINT` :

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

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

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

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

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

1.4.8 Configuration - statistiques

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

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

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

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

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

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

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

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

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

1.4.9 Configuration - autovacuum

– autovacuum

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

Sa configuration est généralement trop basse pour être suffisamment efficace.

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

1.4.10 Tablespaces : principe

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

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

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

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

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

/HDD/tbl/froid:
PG_15_202209061

/HDD/tbl/froid/PG_15_202209061:
5

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

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

```
postgres=# \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
 tbl1       | postgres     | T:\TBL1
```

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

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

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

1.4.10.1 Utilité des tablespaces

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

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

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

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

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

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

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

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

1.4.10.2 Emplacement des tablespaces

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

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

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

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

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

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

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

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

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

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

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

1.4.11 Tablespaces : mise en place

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

CREATE DATABASE nom TABLESPACE 'chaud';

ALTER DATABASE nom SET default_tablespace TO 'chaud';

GRANT CREATE ON TABLESPACE chaud TO un_utilisateur ;

CREATE TABLE une_table (…) TABLESPACE chaud ;

ALTER TABLE une_table SET TABLESPACE chaud ;  -- verrou !

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

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

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

Les ordres SQL plus haut permettent de :

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

Quelques choses à savoir :

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

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

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

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

1.4.12 Tablespaces : configuration

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

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

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

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

1.4.12.1 Tablespaces de données

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

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

1.4.12.2 Tablespaces de tri

Les opérations de tri et les tables temporaires peuvent être déplacées vers un ou plusieurs tablespaces dédiés grâce au paramètre `temp_tablespaces`. Le premier intérêt est de dédier aux tris une partition

rapide (SSD, disque local…). Un autre est de ne plus risquer de saturer la partition du PGDATA en cas de fichiers temporaires énormes dans `base/pgsql_tmp/`.

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

Il faudra ouvrir les droits aux utilisateurs ainsi :

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

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

Paramètres de performances :

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

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

Notamment : `effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[29], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[30].

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

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

[29] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[30] https://docs.postgresql.fr/13/release.html

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

1.4.13 Emplacement des journaux de transactions

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

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

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

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

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

Si l'on souhaite modifier une instance existante, il est nécessaire d'arrêter PostgreSQL, de déplacer le répertoire des journaux de transactions, de créer un lien vers ce répertoire, et enfin de redémarrer PostgreSQL. Voici un exemple qui montre le déplacement dans `/montage/14/pgwal`.

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

1.4.14 Emplacement des fichiers statistiques

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

PostgreSQL met à disposition différents compteurs statistiques via des vues.

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

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

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

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

Sur Rocky Linux 8, la procédure est :

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

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

- création et montage du système de fichiers :

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

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

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

- recharger la configuration de PostgreSQL :

```
SELECT pg_reload_conf() ;
```

```
SHOW stats_temp_directory ;
```

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

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

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

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

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

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

1.5 OUTILS

- pgtune
- pgbench
- postgresqltuner.pl

Nous allons discuter de trois outils.

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

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

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

1.5.1 Outil pgtune

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

Le site du projet en ruby[31] se trouve sur github.

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

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

```
max_connections = 200
shared_buffers = 2GB
effective_cache_size = 6GB
```

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

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

Une application web, c'est beaucoup d'utilisateurs qui exécutent de petites requêtes simples, très rapides, non consommatrices. Du coup, le nombre de connexions a été doublé par rapport à sa valeur par défaut. Le paramètre `work_mem` est augmenté mais raisonnablement par rapport à la mémoire totale et au nombre de connexions. Le paramètre `shared_buffers` se trouve au quart de la mémoire, alors que le paramètre `effective_cache_size` est au deux tiers évoqué précédemment. Le paramètre `wal_buffers` est aussi augmenté, il arrive à 16 Mo. Il peut y avoir beaucoup de transactions en même temps, mais elles seront généralement peu coûteuses en écriture, d'où le fait que les paramètres `min_wal_size`, `max_wal_size` et `checkpoint_completion_target` sont augmentés mais là aussi très raisonnablement. La parallélisation est activée, sans excès.

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

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

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

deux tiers de la mémoire).

Et enfin avec un profil entrepôt de données (*datawarehouse*) sur une machine moins modeste avec 32 Go de RAM, 8 cœurs, et un disque SSD :

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

Pour un entrepôt de données, il y a généralement peu d'utilisateurs à un instant t, mais qui exécutent des requêtes complexes sur une grosse volumétrie. Du coup, la configuration change en profondeur cette fois. Le paramètre `max_connections` est diminué très fortement. Cela permet d'allouer beaucoup de mémoire aux tris et hachages (paramètre `work_mem` à 26 Mo). `shared_buffers` et `effective_cache_size` suivent les règles habituelles. Les entrepôts de données ont souvent des scripts d'import de données (batchs) : cela nécessite de pouvoir écrire rapidement de grosses quantités de données, autrement dit une augmentation conséquente du paramètre `wal_buffers` et des `min_wal_size` / `max_wal_size`. Du fait de la grosse volumétrie des bases dans ce contexte, une valeur importante pour le `maintenance_work_mem` est essentielle pour que les créations d'index et les `VACUUM` se fassent rapidement. De même, la valeur du `default_statistics_target` est sérieusement augmentée car le nombre de lignes des tables est conséquent et nécessite un échantillon plus important pour avoir des statistiques précises sur les données des tables. `random_page_cost`, auparavant à sa valeur par défaut, descend à 1.1 pour tenir compte des performances d'un SSD. C'est le cas aussi pour `effective_io_concurrency`. Enfin, la configuration de la machine autorise une parallélisation importante, généralement bienvenue pour du décisionnel.

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

1.5.2 Outil pgbench

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

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

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

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

1.5.3 Types de tests avec pgbench

- On peut faire varier différents paramètres, tel que :
  - le nombre de clients
  - le nombre de transactions par client
  - faire un test de performance en `SELECT only`, `UPDATE only` ou `TPC-B`
  - faire un test de performance dans son contexte applicatif
  - exécuter le plus de requêtes possible sur une période de temps donné
  - etc.

1.5.4 Environnement de test avec pgbench

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

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[32].

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.
```

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

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 2
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 2.732 ms
tps = 366.049857 (including connections establishing)
tps = 396.322853 (excluding connections establishing)
```

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

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 200
number of transactions actually processed: 2000/2000
latency average = 19.716 ms
tps = 507.204902 (including connections establishing)
tps = 507.425131 (excluding connections establishing)
```

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

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 200
number of transactions actually processed: 2000/2000
latency average = 2.361 ms
tps = 4234.926931 (including connections establishing)
tps = 4272.412154 (excluding connections establishing)
```

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

```
$ pgbench -c 10 -T 10 benchs
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
duration: 10 s
number of transactions actually processed: 45349
latency average = 2.207 ms
tps = 4531.835068 (including connections establishing)
tps = 4534.070449 (excluding connections establishing)
```

1.5.5 Environnement réel avec pgbench

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

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

Pour effectuer de tels tests, il faut créer un ou plusieurs scripts SQL contenant les requêtes à exécuter sur la base de donnée. Avant la 9.6, chaque requête doit être écrite sur **UNE** seule ligne, sinon le point-virgule `;` habituel convient. Un script peut contenir plusieurs requêtes. Toutes les requêtes du fichier seront exécutées dans leur ordre d'apparition. Si plusieurs scripts SQL sont indiqués, chaque transaction sélectionne le fichier à exécuter de façon aléatoire. Enfin, il est possible d'utiliser des variables dans vos scripts SQL afin de faire varier le groupe de données manipulé dans vos tests. Ce dernier point est essentiel afin d'éviter les effets de cache ou encore pour simuler la charge lorsqu'un sous-ensemble des données de la base est utilisé en comparaison avec la totalité de la base (en utilisant un champ de date par exemple).

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

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

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

1.5.6 Outil postgresqltuner.pl

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

Ci-dessous figure un exemple de sortie en version 11 sur une machine virtuelle avec un SSD. Les conseils sont généralement pertinents. Il conseille notamment d'installer l'extension `pg_stat_statements`[33] et d'attendre un certain temps avant d'utiliser l'outil.

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

postgresqltuner.pl version 1.0.1
Checking if OS commands are available on /var/run/postgresql...
[OK]      OS command OK
Connecting to /var/run/postgresql:5432 database template1 with user postgres...
[OK]      User used for report has superuser rights
=====  OS information  =====
[INFO]    OS: linux Version: 3.10.0-693.17.1.el7.x86_64
          Arch: x86_64-linux-thread-multi
[INFO]    OS total memory: 3.70 GB
[OK]      vm.overcommit_memory is good: no memory overcommitment
[INFO]    Running in kvm hypervisor
[INFO]    Currently used I/O scheduler(s): mq-deadline
=====  General instance informations  =====
-----  Version  -----
[OK]      You are using latest major 11.5
-----  Uptime  -----
[INFO]    Service uptime:  44s
[WARN]    Uptime is less than 1 day. postgresqltuner.pl result may not be accurate
-----  Databases  -----
[INFO]    Database count (except templates): 2
[INFO]    Database list (except templates): postgres postgis
-----  Extensions  -----
[INFO]    Number of activated extensions: 1
[INFO]    Activated extensions: plpgsql
[WARN]    Extensions pg_stat_statements is disabled in database template1
-----  Users  -----
[OK]      No user account will expire in less than 7 days
[OK]      No user with password=username
[OK]      Password encryption is enabled
-----  Connection information  -----
[INFO]    max_connections: 100
[INFO]    current used connections: 6 (6.00%)
```

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

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

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

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

1.6 CONCLUSION

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

1.6.1 Questions

N'hésitez pas, c'est le moment !

1.7 QUIZ

https://dali.bo/j1_quiz

1.8 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

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

Elle ne dure que quelques minutes.

1.8.1 Sur Rocky Linux 8 ou 9

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

Installation du dépôt communautaire :

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

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

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

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

# dnf -qy module disable postgresql
```

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

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

Les outils clients et les librairies nécessaires seront automatiquement installés.

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

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

Création d'une première instance :

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

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

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

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

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

Chemins :

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

Configuration :

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

Commandes d'administration habituelles :

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

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

Test rapide de bon fonctionnement et connexion à psql :

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

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

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

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

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

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

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

Création d'autres instances :

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

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

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

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

- Création du fichier service de la deuxième instance :

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

- Modification de ce dernier fichier avec le nouveau chemin :

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

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

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

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

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

- Ouvrir le nouveau port dans le firewall au besoin.

1.8.2 Sur Debian / Ubuntu

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

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

Installation du dépôt communautaire :

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

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

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

Installation de PostgreSQL 16 :

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

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

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

Puis installer les paquets serveur et clients et leurs dépendances :

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

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

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

Chemins :

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

Configuration

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

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

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

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

Démarrage de l'instance avec le serveur :

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

Ouverture du firewall :

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

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

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

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

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

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

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

- créer une instance :

```
# pg_createcluster 16 infocentre
```

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

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

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

```
# pg_ctlcluster 16 infocentre start
```

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

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

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

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

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

```
local   all             postgres                                trust
```

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

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

Une authentification par mot de passe est plus sécurisée :

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

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

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

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

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

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

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

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

Ce fichier doit être protégé des autres utilisateurs :

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

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

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

Rappels :

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

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

root:~# pg_ctlcluster 16 main reload

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

1.9 INTRODUCTION À PGBENCH

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

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[34].

1.9.1 Installation

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

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

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

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

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

Le `pg_hba.conf` doit éventuellement être adapté.

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

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

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

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

1.9.2 Générer de l'activité

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

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

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

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

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

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

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

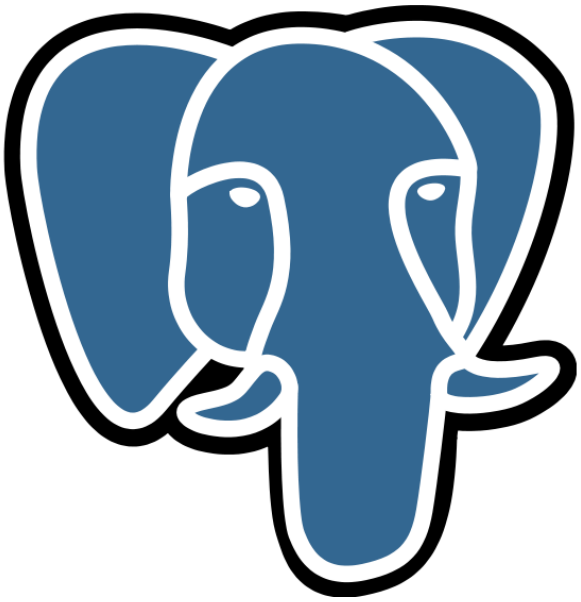
Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

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

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

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

2/ Introduction aux plans d’exécution



2.1 INTRODUCTION

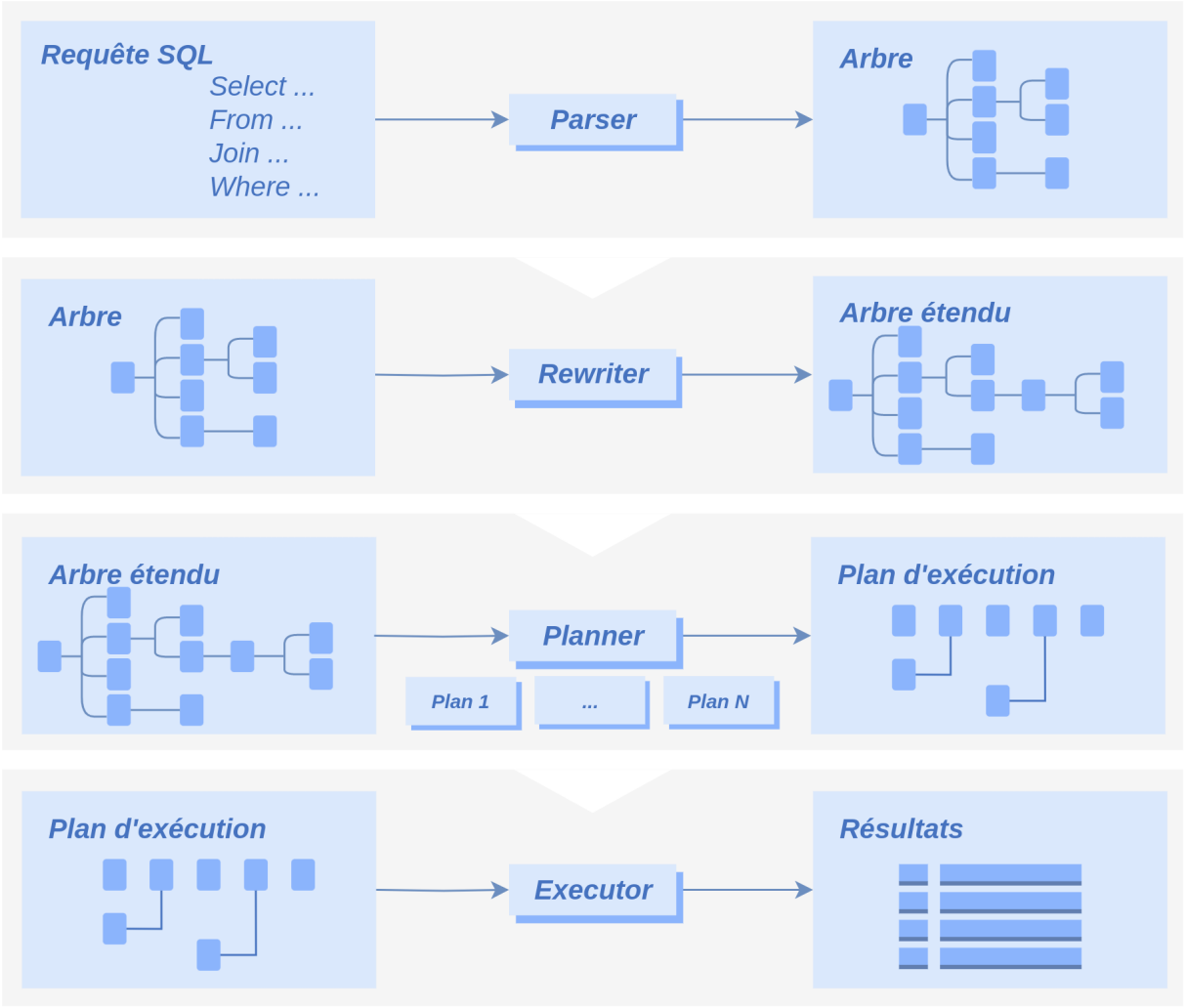
- Qu'est-ce qu'un plan d'exécution ?
- Quels outils peuvent aider

Ce module a pour but de faire une présentation très rapide de l'optimiseur et des plans d'exécution. Il contient surtout une introduction sur la commande `EXPLAIN` et sur différents outils en relation.

2.1.1 Au menu

- Exécution globale d'une requête
- Optimiseur
- `EXPLAIN`
- Nœuds d'un plan
- Outils

2.1.2 Niveau SGBD



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

Tout d'abord, le *parser* va réaliser une analyse syntaxique de la requête.

Puis le *rewriter* va réécrire, si nécessaire, la requête. Pour cela, il prend en compte les règles, les vues non matérialisées et les fonctions SQL.

Si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle.

Si une vue non matérialisée est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée. Il en est de même pour une fonction SQL intégrable.

Ensuite, 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.

Enfin, 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.

En général, le principal souci pour les performances sur ce type d'instructions est donc l'obtention des verrous et l'exécution réelle de la requête. Il existe quelques ordres (comme `TRUNCATE` ou `COPY`) exécutés beaucoup plus directement.

2.2 OPTIMISEUR

- SQL est un langage déclaratif
- Une requête décrit le résultat à obtenir
  - mais pas la façon pour l'obtenir
- C'est à l'optimiseur de déduire le moyen de parvenir au résultat demandé : comment ?

Les moteurs de base de données utilisent un langage SQL qui permet à l'utilisateur de décrire le résultat qu'il souhaite obtenir, mais pas la manière. C'est à la base de données de se débrouiller pour obtenir ce résultat le plus rapidement possible.

2.2.1 Principe de l'optimiseur

Le modèle vise à minimiser un coût :

- Énumérer tous les plans d'exécution
  - ou presque tous…
- Statistiques + configuration + règles → coût calculé
- Coût le plus bas = meilleur plan

Le but de l'optimiseur 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).

Pour calculer le « coût » d'un plan, PostgreSQL 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.

À la fin de l'énumération et du calcul de coût, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût.

Le coût d'un plan est une valeur calculée sans unité ni signification physique.

2.2.2 Exemple de requête et son résultat

```
SELECT nom, prenom, num_service
FROM employes
WHERE nom LIKE 'B%'
ORDER BY num_service;

    nom     |  prenom   | num_service
------------+-----------+-------------
 Berlicot   | Jules     |           2
 Brisebard  | Sylvie    |           3
 Barnier    | Germaine  |           4
```

La requête en exemple permet de récupérer des informations sur tous les employés dont le nom commence par la lettre B en triant les employés par leur service.

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

- faire un parcours séquentiel de la table `employes` en filtrant les enregistrements d'après leur nom, puis trier les données grâce à un algorithme ;
- faire un parcours d'index (s'il y en a un) sur la colonne `nom` pour trouver plus rapidement les enregistrements de la table `employes` satisfaisant le filtre `'B%'`, puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne `num_service` pour récupérer les enregistrements déjà triés par service, et ne retourner que ceux vérifiant le prédicat `nom like 'B%'`.

Et ce ne sont que quelques exemples, car il serait possible d'avoir un index utilisable à la fois 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'opérations de base. Ces opérations travaillent sur des ensembles de lignes, généralement un ou deux. Chaque opération renvoie un seul ensemble de lignes. Le planificateur peut combiner ces opérations suivant certaines règles. Une opération 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.

2.2.3 Décisions de l'optimiseur

- Comment accéder aux lignes ?
  - parcours de table, d'index, de fonction, etc.
- Comment joindre les tables ?
  - ordre
  - type
- Comment agréger ?
  - brut, tri, hachage…

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut utiliser un parcours de table (une lecture complète du fichier), un parcours d'index ou encore d'autres types de parcours. Ce sont généralement les premières opérations utilisées.

Pour joindre les tables, l'ordre dans lequel ce sera fait est très important. Pour la jointure elle-même, il existe plusieurs méthodes différentes. Il existe aussi plusieurs algorithmes d'agrégation de lignes. Un tri peut être nécessaire pour une jointure, une agrégation, ou pour un `ORDER BY`, et là encore il y a plusieurs algorithmes possibles, ou des techniques pour éviter de le faire.

2.3 MÉCANISME DE CALCUL DE COÛTS

- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne
  - appliquer un opérateur
  - …
- et généralement un paramètre associé

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

2.3.1 Statistiques

- Connaître le coût de traitement d'une ligne est bien
  - mais combien de lignes à traiter ?
- Statistiques sur les données
  - mises à jour : `ANALYZE`
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est une bonne chose, mais si on ne sait pas le nombre de lignes à traiter, on ne peut pas calculer le coût total. L'optimiseur a donc besoin de statistiques sur les données, comme par exemple le nombre de blocs et de lignes d'une table, les valeurs les plus fréquentes et leur fréquence pour chaque colonne de chaque table. Les statistiques sur les données sont calculées lors de l'exécution de la commande SQL `ANALYZE`. L'autovacuum exécute généralement cette opération en arrière-plan.

Des statistiques périmées ou pas assez fines sont une source fréquente de plans non optimaux !

2.3.2 Exemple - parcours d'index

```
CREATE TABLE t1 (c1 integer, c2 integer);
INSERT INTO t1 SELECT i, i FROM generate_series(1, 1000) i;
CREATE INDEX ON t1(c1);
ANALYZE t1;

EXPLAIN SELECT * FROM t1 WHERE c1=1 ;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using t1_c1_idx on t1 (cost=0.28..8.29 rows=1 width=8)
  Index Cond: (c1 = 1)
```

L'exemple crée une table et lui ajoute 1000 lignes. Chaque ligne a une valeur différente dans les colonnes `c1` et `c2` (de 1 à 1000).

```
SELECT * FROM t1 ;
```

```
  c1  |  c2
------+------
    1 |    1
    2 |    2
    3 |    3
    4 |    4
    5 |    5
    6 |    6
…
  996 |  996
  997 |  997
  998 |  998
  999 |  999
 1000 | 1000
(1000 lignes)
```

Dans cette requête :

```
EXPLAIN SELECT * FROM t1 WHERE c1=1 ;
```

nous savons qu'un `SELECT` filtrant sur la valeur 1 pour la colonne `c1` ne ramènera qu'une ligne. Grâce aux statistiques relevées par la commande `ANALYZE` exécutée juste avant, l'optimiseur estime lui aussi qu'une seule ligne sera récupérée. Une ligne sur 1000, c'est un bon ratio pour faire un parcours d'index. C'est donc ce que recommande l'optimiseur.

2.3.3 Exemple - parcours de table

```
UPDATE t1 SET c1=1 ;   /* 1000 lignes identiques */

ANALYZE t1 ;           /* ne pas oublier ! */

EXPLAIN SELECT * FROM t1 WHERE c1=1;

                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..21.50 rows=1000 width=8)
   Filter: (c1 = 1)
```

La même table, mais avec 1000 lignes ne contenant plus que la valeur 1. Un `SELECT` filtrant sur cette valeur 1 ramènera dans ce cas toutes les lignes. L'optimiseur s'en rend compte et décide qu'un parcours séquentiel de la table est préférable à un parcours d'index. C'est donc ce que recommande l'optimiseur.

Dans cet exemple, l'ordre `ANALYZE` garantit que les statistiques sont à jour (le démon autovacuum n'est pas forcément assez rapide).

2.3.4 Exemple - parcours d'index forcé

```
SET enable_seqscan TO off ;

EXPLAIN SELECT * FROM t1 WHERE c1=1;

                          QUERY PLAN
-------------------------------------------------------------
Index Scan using t1_c1_idx on t1 (cost=0.28..57.77 rows=1000 width=8)
  Index Cond: (c1 = 1)

RESET enable_seqscan ;
```

Le coût du parcours de table était de 21,5 pour la récupération des 1000 lignes, donc un coût bien supérieur au coût du parcours d'index, qui lui était de 8,29, mais pour une seule ligne. On pourrait se demander le coût du parcours d'index pour 1000 lignes. À titre expérimental, on peut désactiver (ou plus exactement désavantager) le parcours de table en configurant le paramètre `enable_seqscan` à `off`.

En faisant cela, on s'aperçoit que le plan passe finalement par un parcours d'index, tout comme le premier. Par contre, le coût n'est plus de 8,29, mais de 57,77, donc supérieur au coût du parcours

de table. C'est pourquoi l'optimiseur avait d'emblée choisi un parcours de table. Un index n'est pas forcément le chemin le plus court.

2.4 QU'EST-CE QU'UN 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

L'optimiseur transforme une grosse action (exécuter une requête) en plein de petites actions unitaires (trier un ensemble de données, lire une table, parcourir un index, joindre deux ensembles de données, etc). Ces petites actions sont liées les unes aux autres. Par exemple, pour exécuter cette requête :

```
SELECT * FROM une_table ORDER BY une_colonne;
```

peut se faire en deux actions :

- récupérer les enregistrements de la table ;
- trier les enregistrements provenant de la lecture de la table.

Mais ce n'est qu'une des possibilités.

2.4.1 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
  - le 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.

2.4.2 Récupérer un plan d'exécution

- Commande `EXPLAIN`
  - suivi de la requête complète
- Uniquement le plan finalement retenu

Pour récupérer le plan d'exécution d'une requête, il suffit d'utiliser la commande `EXPLAIN`. Cette commande est suivie de la requête pour laquelle on souhaite le plan d'exécution.

Seul le plan sélectionné est affichable. Les plans ignorés du fait de leur coût trop important ne sont pas récupérables. Ceci est dû au fait que les plans en question peuvent être abandonnés avant d'avoir été totalement développés si leur coût partiel est déjà supérieur à celui de plans déjà considérés.

2.4.3 Exemple de requête

```
EXPLAIN SELECT * FROM t1  WHERE c2<10 ORDER BY c1;
```

Cette requête va récupérer tous les enregistrements de t1 pour lesquels la valeur de la colonne c2 est inférieure à 10. Les enregistrements sont triés par rapport à la colonne c1.

2.4.4 Plan pour cette requête

```
                          QUERY PLAN
-------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Sort Key: c1
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
         Filter: (c2 < 10)
```

L'optimiseur envoie ce plan à l'exécuteur. Ce dernier voit qu'il a une opération de tri à effectuer (nœud `Sort`). Pour cela, il a besoin de données que le nœud suivant va lui donner. Il commence donc l'opération de lecture (nœud `SeqScan`). Il envoie chaque enregistrement valide au nœud `Sort` pour que ce dernier les trie.

Chaque nœud dispose d'un certain nombre d'informations placées soit sur la même ligne entre des parenthèses, soit sur la ou les lignes du dessous. La différence entre une ligne de nœud et une ligne d'informations est que la ligne de nœud contient une flèche au début (`->`). Par exemple, le nœud `Sort` contient des informations entre des parenthèses et une information supplémentaire sur la ligne suivante indiquant la clé de tri (la colonne `c1`). Par contre, la troisième ligne n'est pas une ligne d'informations du nœud `Sort` mais un nouveau nœud (`SeqScan`).

2.4.5 Informations sur la ligne nœud

```
->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
      Filter: (c2 < 10)
```

- `cost` : coûts de récupération
  - de la **première** ligne
  - de **toutes** les lignes
- `rows`
  - nombre de lignes en sortie du nœud
- `width`
  - largeur moyenne d'un enregistrement (octets)

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 (elle dépend essentiellement de la taille de la table lue, mais aussi d'opération de filtrage). `rows` correspond au nombre de lignes que le planificateur pense récupérer à la sortie de ce nœud. Dans le cas d'une nouvelle table traitée par `ANALYZE`, les versions antérieures à la version 14 calculaient une valeur probable du nombre de lignes en se basant sur la taille moyenne d'une ligne et sur une table faisant 10 blocs. La version 14 corrige cela en ayant une meilleure idée du nombre de lignes d'une nouvelle table. `width` est la largeur en octets de la ligne.

2.4.6 Informations sur les lignes suivantes

```
Sort  (cost=21.64..21.67 rows=9 width=8)
  Sort Key: c1
Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
  Filter: (c2 < 10)
```

- `Sort`
  - `Sort Key` : clé de tri
- `Seq Scan`
  - `Filter` : filtre (si besoin)
- Dépend
  - du type de nœud
  - des options de `EXPLAIN`
  - des paramètres de configuration
  - de la version de PostgreSQL

Les informations supplémentaires dépendent de beaucoup d'éléments. Elles peuvent différer suivant le type de nœud, les options de la commande `EXPLAIN`, et certains paramètres de configuration. De même la version de PostgreSQL joue un rôle majeur : les nouvelles versions peuvent apporter des informations supplémentaires pour que le plan soit plus lisible et que l'utilisateur soit mieux informé.

2.4.7 Option ANALYZE

```
EXPLAIN (ANALYZE)  /* exécution !! */
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
       (actual time=0.493..0.498 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
                 (actual time=0.061..0.469 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
 Planning Time: 0.239 ms
 Execution Time: 0.606 ms
```

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

Avec `ANALYZE`, la requête est réellement exécutée ! Attention donc aux `INSERT` / `UPDATE` / `DELETE`. N'oubliez pas non plus qu'un `SELECT` peut appeler des fonctions qui écrivent dans la base. Dans le doute, pensez à englober l'appel dans une transaction que vous annulerez après coup.

Quatre nouvelles informations apparaissent dans un nouveau bloc de parenthèses. Elles sont 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'exécutions de ce nœud, soit dans le cadre d'une jointure, soit dans le cadre d'une requête parallélisée.

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).

2.4.8 Option BUFFERS

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=17.64..17.67 rows=9 width=8)
       (actual time=0.126..0.127 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   Buffers: shared hit=3 read=5
   ->  Seq Scan on t1  (cost=0.00..17.50 rows=9 width=8)
                       (actual time=0.017..0.106 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
         Buffers: shared read=5
```

`BUFFERS` fait apparaître le nombre de blocs (*buffers*) impactés par chaque nœud du plan d'exécution, en lecture comme en écriture.

`shared read=5` en bas signifie que 5 blocs ont été trouvés et lus **hors** du cache de PostgreSQL (*shared buffers*). 5 blocs est ici la taille de `t1` sur le disque. Le cache de l'OS est peut-être intervenu, ce n'est pas visible ici. Un peu plus haut, `shared hit=3 read=5` indique que 3 blocs ont été lus dans ce cache, et 5 autres toujours hors du cache. Les valeurs exactes dépendent donc de l'état du cache. Si on relance la requête, pour une telle petite table, les relectures se feront uniquement en `shared hit`.

`BUFFERS` compte aussi les blocs de fichiers ou tables temporaires (`temp` ou `local`), ou les blocs écrits sur disque (`written`).

`EXPLAIN (ANALYZE, BUFFERS)` n'affiche que des données réelles, pas des estimations. Depuis PostgreSQL 13, `EXPLAIN (BUFFERS)` sans `ANALYZE` peut être utilisé, mais il ne montre que les quelques blocs utilisés par la planification, plutôt que tous ceux auxquels la requête accéderait réellement.

2.4.9 Option SETTINGS

```
SET enable_seqscan TO off ;
SET work_mem TO '100MB';

EXPLAIN (SETTINGS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using t1_c1_idx on t1  (cost=0.28..57.77 rows=9 width=8)
  Filter: (c2 < 10)
Settings: enable_seqscan = 'off', work_mem = '100MB'

RESET ALL ;
```

Désactivée par défaut, l'option `SETTINGS` permet d'obtenir les valeurs des paramètres qui ne sont pas à leur valeur par défaut dans la session de la requête. Elle est pratique quand il faut transmettre le plan à un collègue ou un prestataire qui n'a pas forcément accès à la machine.

2.4.10 Option WAL

```
EXPLAIN (ANALYZE, WAL)
INSERT INTO t1 SELECT i, i FROM generate_series(1,1000) i ;

                        QUERY PLAN
-----------------------------------------------------
 Insert on t1  (cost=0.00..10.00 rows=1000 width=8)
          (actual time=8.078..8.079 rows=0 loops=1)
   WAL: records=2017 fpi=3 bytes=162673
   ->  Function Scan on generate_series i
       (cost=0.00..10.00 rows=1000 width=8)
       (actual time=0.222..0.522 rows=1000 loops=1)
 Planning Time: 0.076 ms
 Execution Time: 8.141 ms
```

Désactivée par défaut et nécessitant l'option `ANALYZE`, l'option `WAL` permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions. (Rappelons que les écritures dans les fichiers de données se font généralement plus tard, en arrière-plan.)

2.4.11 Option GENERIC_PLAN

Quel plan générique pour les requêtes préparées ?

```
EXPLAIN (GENERIC_PLAN)
SELECT * FROM t1 WHERE c1 < $1 ;
```

- PostgreSQL 16

L'option `GENERIC_PLAN` n'est malheureusement pas disponible avant PostgreSQL 16. Elle est pourtant très pratique quand on cherche le plan d'une requête préparée sans connaître ses paramètres, ou pour savoir quel est le plan générique que prévoit PostgreSQL pour une requête préparée.

En effet, les plans des requêtes préparées ne sont pas forcément recalculés à chaque appel avec les paramètres exacts (le système est assez complexe et dépend du paramètre `plan_cache_mode`). La requête ne peut être exécutée sans vraie valeur de paramètre, donc l'option `ANALYZE` est inutilisable, mais en activant `GENERIC_PLAN` on peut tout de même voir le plan générique que PostgreSQL peut choisir (`SUMMARY ON` affiche en plus le temps de planification) :

```
EXPLAIN (GENERIC_PLAN, SUMMARY ON)
SELECT * FROM t1 WHERE c1 < $1 ;

                              QUERY PLAN
----------------------------------------------------------------------
```

```
 Index Scan using t1_c1_idx on t1  (cost=0.15..14.98 rows=333 width=8)
   Index Cond: (c1 < $1)
 Planning Time: 0.195 ms
```

C'est effectivement le plan qui serait optimal pour `$1` =1. Mais pour la valeur 1000, qui ramène toute la table, un *Seq Scan* serait plus pertinent.

2.4.12 Autres options

- `COSTS OFF`
  - masquer les coûts
- `TIMING OFF`
  - désactiver le chronométrage & des informations vues/calculées par l'optimiseur
- `VERBOSE`
  - affichage verbeux : schémas, colonnes, workers
- `SUMMARY`
  - affichage du temps de planification et exécution (si applicable)
- `FORMAT`
  - sortie en texte, JSON, XML, YAML

Ces options sont moins utilisées, mais certaines restent intéressantes dans des cas précis.

Option COSTS

Cette option est activée par défaut. Il peut être intéressant de la désactiver pour n'avoir que le plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
```

```
        QUERY PLAN
--------------------------
 Sort
   Sort Key: c1
   ->  Seq Scan on t1
         Filter: (c2 < 10)
```

Option TIMING

Cette option est activée par défaut. Il peut être intéressant de le désactiver sur les systèmes où le chronométrage prend beaucoup de temps et allonge inutilement la durée d'exécution de la requête. Mais de ce fait, le résultat devient beaucoup moins intéressant.

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8) (actual rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8) (actual rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
 Planning Time: 0.155 ms
 Execution Time: 0.381 ms
```

Option VERBOSE

Désactivée par défaut, l'option `VERBOSE` permet d'afficher des informations supplémentaires comme :

- la liste des colonnes en sortie ;
- le nom des objets qualifiés par le nom du schéma ;
- des statistiques sur les workers (pour les requêtes parallélisées) ;
- le code SQL envoyé à un serveur distant (pour les tables distantes avec `postgres_fdw` notamment).

Dans l'exemple suivant, le nom du schéma est ajouté au nom de la table. La nouvelle ligne `Output` indique la liste des colonnes de l'ensemble de données en sortie du nœud.

```
EXPLAIN (VERBOSE) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
                           QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Output: c1, c2
   Sort Key: t1.c1
   ->  Seq Scan on public.t1  (cost=0.00..21.50 rows=9 width=8)
         Output: c1, c2
         Filter: (t1.c2 < 10)
```

Option SUMMARY

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 (la durée de planification est pourtant parfois importante). Par contre, un `EXPLAIN ANALYZE` l'affiche par défaut.

```
EXPLAIN (SUMMARY ON) SELECT * FROM t1 WHERE c2<10 ORDER BY c1;
                       QUERY PLAN
---------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Sort Key: c1
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
         Filter: (c2 < 10)
 Planning Time: 0.185 ms

EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1;
```

```
                     QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
       (actual time=0.343..0.346 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
                 (actual time=0.031..0.331 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
```

Option FORMAT

L'option `FORMAT` permet de préciser le format du texte en sortie. Par défaut, il s'agit du format texte habituel, mais il est possible de choisir un format semi-structuré parmi JSON, XML et YAML. Les formats semi-structurés sont utilisés principalement par des outils d'analyse comme explain.dalibo.com[1], car le contenu est plus facile à analyser, et même un peu plus complet. Voici ce que donne la commande `EXPLAIN` avec le format JSON :

```
psql -X -AtX \
-c 'EXPLAIN (FORMAT JSON) SELECT * FROM t1 WHERE c2<10 ORDER BY c1' | jq '.[]'
```

```
{
  "Plan": {
    "Node Type": "Sort",
    "Parallel Aware": false,
    "Async Capable": false,
    "Startup Cost": 34.38,
    "Total Cost": 34.42,
    "Plan Rows": 18,
    "Plan Width": 8,
    "Sort Key": [
      "c1"
    ],
    "Plans": [
      {
        "Node Type": "Seq Scan",
        "Parent Relationship": "Outer",
        "Parallel Aware": false,
        "Async Capable": false,
        "Relation Name": "t1",
        "Alias": "t1",
        "Startup Cost": 0,
        "Total Cost": 34,
        "Plan Rows": 18,
        "Plan Width": 8,
        "Filter": "(c2 < 10)"
      }
    ]
  }
}
```

[1] https://explain.dalibo.com

2.4.13 Paramètre track_io_timing

```
SET track_io_timing TO on;
EXPLAIN (ANALYZE, BUFFERS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1 ;

                              QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=52.14..52.21 rows=27 width=8) (actual time=1.359..1.366
↪  rows=27 loops=1)
   …
   Buffers: shared hit=3 read=14
   I/O Timings: read=0.388
   ->  Seq Scan on t1  (cost=0.00..51.50 rows=27 width=8) (actual
↪  time=0.086..1.233 rows=27 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 2973
         Buffers: shared read=14
         I/O Timings: read=0.388
 Planning:
   Buffers: shared hit=43 read=14
   I/O Timings: read=0.469
 Planning Time: 1.387 ms
 Execution Time: 1.470 ms
```

La configuration du paramètre `track_io_timing` permet de demander le chronométrage des opérations d'entrée/sortie disque. Sur ce plan, nous pouvons voir que 14 blocs ont été lus en dehors du cache de PostgreSQL et que cela a pris 0,388 ms pour les lire (ils étaient certainement dans le cache du système d'exploitation).

Cette information permet de voir si le temps d'exécution de la requête est dépensé surtout dans la demande de blocs au système d'exploitation (donc hors du cache de PostgreSQL) ou dans l'exécution même de la requête (donc interne à PostgreSQL).

2.4.14 Détecter les problèmes

- Temps d'exécution de chaque opération
- Différence entre l'estimation du nombre de lignes et la réalité
- Boucles
  - appels, même rapides, nombreux
- Opérations utilisant beaucoup de blocs (`BUFFERS`)
- Opérations lentes de lecture/écriture (`track_io_timing`)

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.

Par exemple, une différence importante entre le nombre estimé de lignes et le nombre réel de lignes 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.

Les boucles sont à surveiller. Par exemple, un accès à une ligne par un index est généralement très rapide, mais répété des millions de fois à cause d'une boucle, le total est parfois plus long qu'une lecture complète de la table indexée. C'est notamment l'enjeu du réglage entre `seq_page_cost` et `random_page_cost`.

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 du cache de PostgreSQL. Sachant qu'un bloc fait généralement 8 kilo-octets, il est aisé de déterminer le volume de données manipulé par une requête.

2.5 NŒUDS D'EXÉCUTION LES PLUS COURANTS (INTRODUCTION)

- Parcours
- Jointures
- Agrégats
- Tri

Nous n'allons pas détailler tous les nœuds existants, mais évoquer simplement les plus importants. Une analyse plus poussée des nœuds et une référence complète sont disponibles dans les modules J2[2] et J6[3].

2.5.1 Parcours

- Table
  - *Seq Scan*, *Parallel Seq Scan*
- Index
  - *Index Scan*, *Bitmap Scan*, *Index Only Scan*
  - et les variantes parallélisées
- Autres
  - *Function Scan*, *Values Scan*

Plusieurs types d'objets peuvent être parcourus. Pour chacun, l'optimiseur peut choisir entre plusieurs types de parcours.

Les tables passent par un *Seq Scan* qui est une lecture simple de la table, bloc par bloc, ligne par ligne. Ce parcours peut filtrer les données mais ne les triera pas. Une variante parallélisée existe sous le nom de *Parallel Seq Scan*.

Les index disposent de plusieurs parcours, principalement suivant la quantité d'enregistrements à récupérer :

- *Index Scan* quand il y a très peu d'enregistrements à récupérer ;
- *Bitmap Scan* quand il y en a un peu plus ou quand on veut lire plusieurs index d'une même table pour satisfaire plusieurs conditions de filtre ;

[2] https://dali.bo/j2_html
[3] https://dali.bo/j6_html

- *Index Only Scan* quand les champs de la requête correspondent aux colonnes de l'index (ce qui permet d'éviter la lecture de tout ou partie de la table).

Ces différents parcours sont parallélisables. Ils ont dans ce cas le mot *Parallel* ajouté en début du nom du nœud.

Enfin, il existe des parcours moins fréquents, comme les parcours de fonction (*Function Scan*) ou de valeurs (*Values Scan*).

2.5.2 Jointures

- Algorithmes
  - *Nested Loop*
  - *Hash Join*
  - *Merge Join*
- Parallélisation possible
- Pour `EXISTS`, `IN` et certaines jointures externes
  - *Semi Join*
  - *Anti Join*

Trois nœuds existent pour les jointures.

Le *Nested Loop* est utilisé pour toutes les conditions de jointure n'utilisant pas l'opérateur d'égalité. Il est aussi utilisé quand un des deux ensembles de données renvoie très peu de données.

Le *Hash Join* est certainement le nœud le plus commun. Il est utilisé un peu dans tous les cas, sauf si les deux ensembles de données arrivent déjà triés. Dans ce cas, il est préférable de passer par un *Merge Join* qui réclame deux ensembles de données déjà triés.

Les *Semi Join* et *Anti Join* sont utilisés dans des cas très particuliers et peu fréquents.

2.5.3 Agrégats

- Un résultat au total
  - *Aggregate*
- Un résultat par regroupement
  - *Hash Aggregate*
  - *Group Aggregate*
  - *Mixed Aggregate*
- Parallélisation
  - *Partial Aggregate*
  - *Finalize Aggregate*

De même il existe plusieurs algorithmes d'agrégation qui s'occupent des sommes, des moyennes, des regroupements divers, etc. Ils sont souvent parallélisables.

2.5.4 Opérations unitaires

- *Sort*
- *Incremental Sort*
- *Limit*
- *Unique* (`DISTINCT`)
- *Append* (`UNION ALL`), *Except*, *Intersect*
- *Gather* (parallélisme)
- *Memoize* (14+)

Un grand nombre de petites opérations ont leur propre nœud, comme le tri avec *Sort* et *Incremental Sort*, la limite de lignes (`LIMIT`) avec *Limit*, la clause `DISTINCT` avec *Unique*), etc. Elles prennent généralement un ensemble de données et renvoient un autre ensemble de données issu du traitement du premier.

Le groupe des nœuds *Append*, *Except* et *Intersect* ne se comporte pas ainsi. Notamment, *Append* est le seul nœud à prendre potentiellement plus de deux ensembles de données en entrée.

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois

de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop*.

2.6 OUTILS GRAPHIQUES

- pgAdmin
- explain.depesz.com
- explain.dalibo.com

L'analyse de plans complexes devient très vite fastidieuse. Nous n'avons vu ici que des plans d'une dizaine de lignes au maximum, mais les plans de requêtes réellement problématiques peuvent faire plusieurs centaines, voire milliers de lignes. L'analyse manuelle devient impossible. Des outils ont été créés pour mieux visualiser les parties intéressantes des plans.

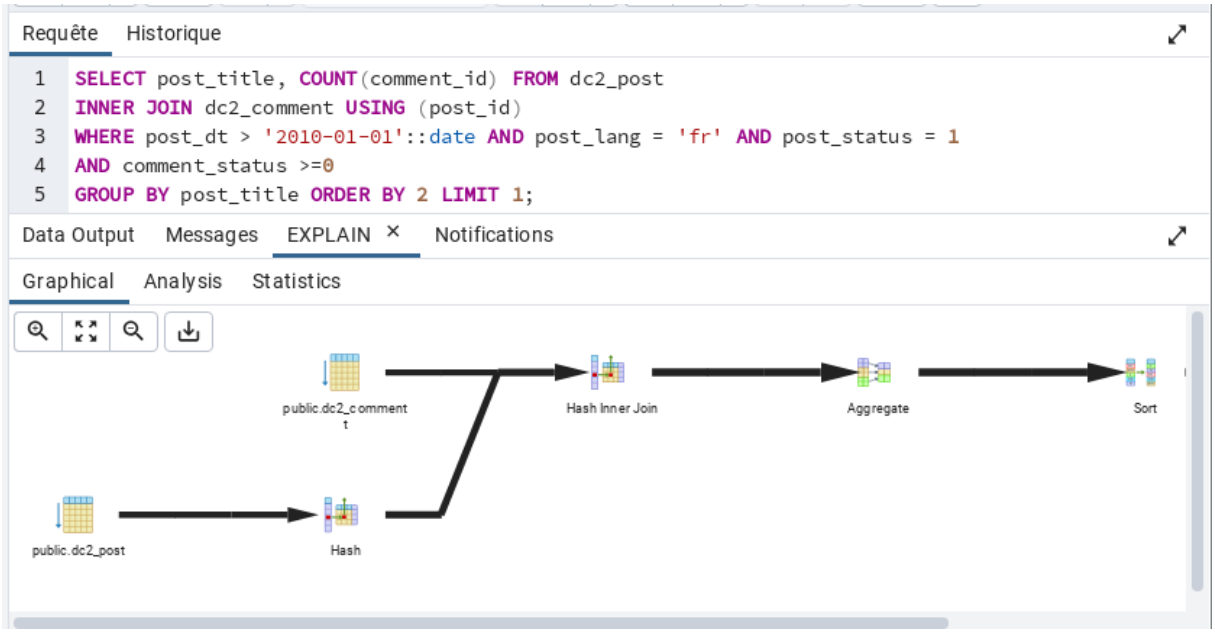
2.6.1 pgAdmin

- 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.

2.6.2 pgAdmin - copie d'écran



Voici un exemple d'un `EXPLAIN` graphique réalisé par pgAdmin 4. En cliquant sur un nœud, un message affiche les informations statistiques sur le nœud.

2.6.3 explain.depesz.com

- Site web avec affichage amélioré du `EXPLAIN ANALYZE`
- Lignes colorées pour indiquer les problèmes
- 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. Cependant, il est aussi connu pour avoir créé un site web d'analyse des plans d'exécution. Ce site web est disponible sur https://explain.depesz.com/

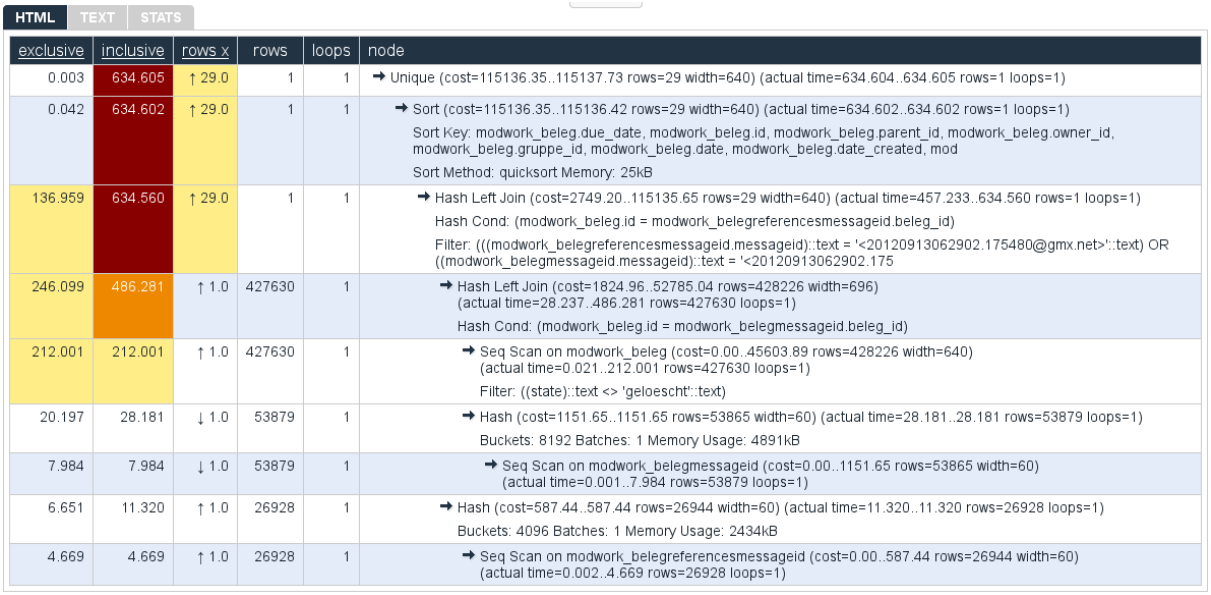
Il suffit d'aller sur ce site, de coller le résultat d'un `EXPLAIN ANALYZE`, et le site affichera le plan d'exécution avec des codes couleurs pour bien distinguer les nœuds performants des autres.

Le code couleur est simple : blanc indique que tout va bien, jaune est inquiétant, marron est plus inquiétant, et rouge très inquiétant.

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

- le module explain en Perl[4]
- la partie site web[5]

2.6.4 explain.depesz.com - exemple



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 l'affichage d'un plan sur le site explain.depesz.com[6].

Voici la signification des différentes colonnes :

- *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.

[4] https://gitlab.com/depesz/Pg--Explain
[5] https://gitlab.com/depesz/explain.depesz.com
[6] https://explain.depesz.com/

2.6.5 explain.dalibo.com

- Reprise de **pev** d'Alex Tatiyants, par Pierre Giraud (Dalibo)
- Page web avec affichage graphique d'un `EXPLAIN [ANALYZE]`
- Repérage des nœuds longs, lourds…
- Affichage flexible
- explain.dalibo.com
- Installable en local

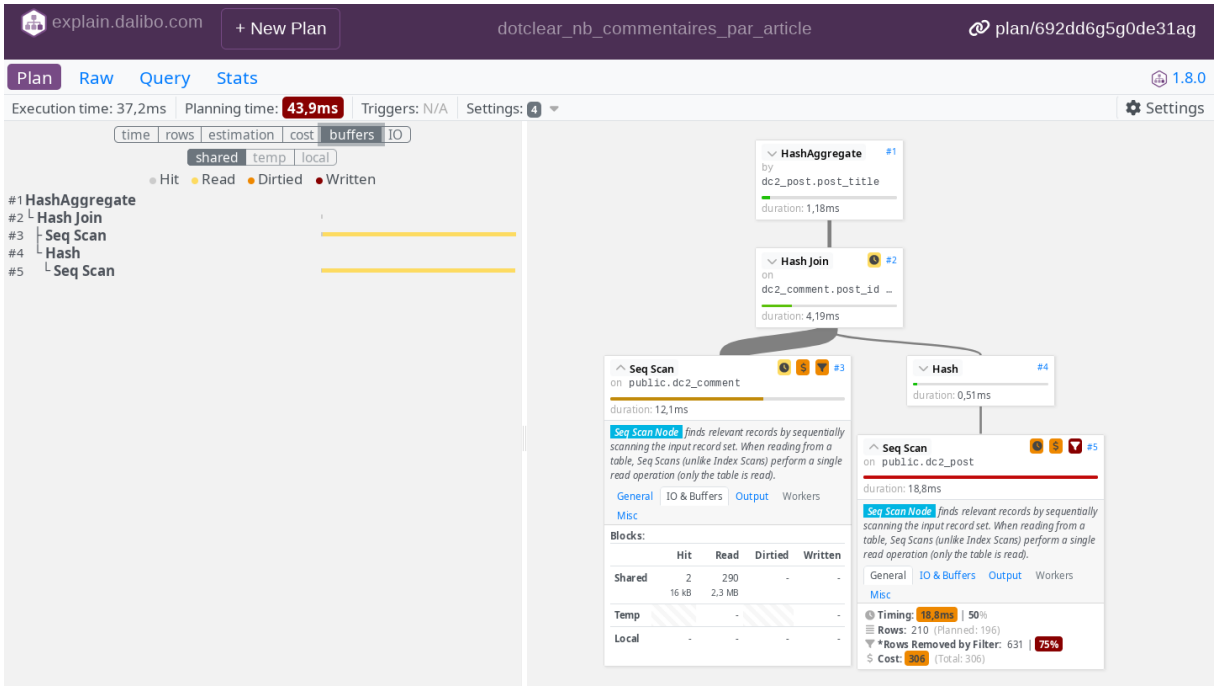
À l'origine, pev (*PostgreSQL Explain Visualizer*) est un outil libre[7] offrant un affichage graphique du plan d'exécution et pointant le nœud le plus coûteux, le plus long, le plus volumineux, etc. Utilisable en ligne[8], il n'est hélas plus maintenu depuis plusieurs années.

explain.dalibo.com[9] en est un *fork*, très étendu et activement maintenu par Pierre Giraud de Dalibo. Les plans au format texte comme JSON sont acceptés. Les versions récentes de PostgreSQL sont supportées, avec leurs spécificités : nouvelles options d'`EXPLAIN`, nouveaux types de nœuds… Tout se passe en ligne. Les plans peuvent être partagés. Si vous ne souhaitez pas qu'ils soient stockés chez Dalibo, utilisez la version strictement locale de pev2[10].

Le code[11] est sous licence PostgreSQL. Techniquement, c'est un composant VueJS qui peut être intégré à vos propres outils.

[7] https://github.com/AlexTatiyants/pev
[8] https://tatiyants.com/pev/#/plans
[9] https://explain.dalibo.com
[10] https://www.github.com/dalibo/pev2/releases/latest/download/index.html
[11] https://github.com/dalibo/pev2

2.6.6 explain.dalibo.com - exemple



explain.dalibo.com permet de repérer d'un coup d'œil les parties les plus longues du plan, celles utilisant le plus de lignes, les écarts d'estimation, les dérives du temps de planification… Les nœuds peuvent être repliés. Plusieurs modes d'affichage sont disponibles.

Un grand nombre de plans d'exemple sont disponibles sur le site.

2.7 CONCLUSION

- Un optimiseur très avancé
- Ne vous croyez pas plus malin que lui
- Mais il est important de savoir comment il fonctionne

Cette introduction à l'optimiseur de PostgreSQL permet de comprendre comment il fonctionne et sur quoi il se base. Cela permet de pointer certains des problèmes. C'est aussi un prérequis indispensable pour voir plus tard l'intérêt des différents index et nœuds d'exécution de PostgreSQL.

2.7.1 Questions

N'hésitez pas, c'est le moment !

2.8 QUIZ

https://dali.bo/j0_quiz

3/ Techniques d’indexation

Photo de Maksym Kaharlytskyi[1], Unsplash licence

[1] https://unsplash.com/@qwitka

3.1 INTRODUCTION

- Qu'est-ce qu'un index ?
- Comment indexer une base ?
- Les index B-tree dans PostgreSQL

3.1.1 Objectifs

- Comprendre ce qu'est un index
- Maîtriser le processus de création d'index
- Connaître les différents types d'index B-tree et leurs cas d'usages

3.1.2 Introduction aux index

- Uniquement destinés à l'optimisation
- À gérer d'abord par le développeur
    - **Markus Winand** : *SQL Performance Explained*

Les index ne sont pas des objets qui font partie de la théorie relationnelle. Ils sont des objets physiques qui permettent d'accélérer l'accès aux données. Et comme ils ne sont que des moyens d'optimisation des accès, les index ne font pas non plus partie de la norme SQL. C'est d'ailleurs pour cette raison que la syntaxe de création d'index est si différente d'une base de données à une autre.

La création des index est à la charge du développeur ou du DBA, leur création n'est pas automatique, sauf exception.

Pour Markus Winand, c'est d'abord au développeur de poser les index, car c'est lui qui sait comment ses données sont utilisées. Un DBA d'exploitation n'a pas cette connaissance, mais il connaît généralement mieux les différents types d'index et leurs subtilités, et voit comment les requêtes réagissent en production. Développeur et DBA sont complémentaires dans l'analyse d'un problème de performance.

Le site de Markus Winand, Use the index, Luke[2], propose une version en ligne de son livre *SQL Performance Explained*, centré sur les index B-tree (les plus courants). Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

[2] https://use-the-index-luke.com

3.1.3 Utilité d'un index

- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire tout récupérer dans l'index (*Index Only Scan*)
- Un index facilite :
  - certains tris
  - certains agrégats
- Obligatoires et automatique pour clés primaires & unicité
  - conseillé pour clés étrangères (FK)

Les index ne changent pas le résultat d'une requête, mais l'accélèrent. L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.

Le cas le plus favorable est l'*Index Only Scan* : toutes les données nécessaires sont contenues dans l'index, lui seul sera lu et PostgreSQL ne lira pas la table elle-même.

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause `WHERE` ;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table ;
- index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table, et obtenir un *Index Only Scan*.

La création des index est à la charge du développeur. Seules exceptions : ceux créés automatiquement quand on déclare des contraintes de clé primaire ou d'unicité. La création est alors automatique.

Les contraintes de clé étrangère imposent qu'il existe déjà une clé primaire sur la table pointée, mais ne crée pas d'index sur la table portant la clé.

3.1.4 Index et lectures

Un index améliore les `SELECT`

- Sans index :

```
=# SELECT * FROM test WHERE id = 10000;
Temps : 1760,017 ms
```

- Avec index :

```
=# CREATE INDEX idx_test_id ON test (id);

=# SELECT * FROM test WHERE id = 10000;
Temps : 27,711 ms
```

L'index est une structure de données qui permet d'accéder rapidement à l'information recherchée. À l'image de l'index d'un livre, pour retrouver un thème rapidement, on préférera utiliser l'index du livre plutôt que lire l'intégralité du livre jusqu'à trouver le passage qui nous intéresse. Dans une base de données, l'index a un rôle équivalent. Plutôt que de lire une table dans son intégralité, la base de données utilisera l'index pour ne lire qu'une faible portion de la table pour retrouver les données recherchées.

Pour la requête d'exemple (avec une table de 20 millions de lignes), on remarque que l'optimiseur n'utilise pas le même chemin selon que l'index soit présent ou non. Sans index, PostgreSQL réalise un parcours séquentiel de la table :

```
EXPLAIN SELECT * FROM test WHERE id = 10000;

                            QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..193661.66 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on test  (cost=0.00..192661.56 rows=1 width=4)
         Filter: (id = 10000)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupérera qu'une seule ligne sur les 20 millions que compte la table :

```
EXPLAIN SELECT * FROM test WHERE id = 10000;

                             QUERY PLAN
---------------------------------------------------------------------
 Index Only Scan using idx_test_id on test  (cost=0.44..8.46 rows=1 width=4)
   Index Cond: (id = 10000)
```

Mais l'index n'accélère pas seulement la simple lecture de données, il permet également d'accélérer les tris et les agrégations, comme le montre l'exemple suivant sur un tri :

```
EXPLAIN SELECT id FROM test
        WHERE id BETWEEN 1000 AND 1200 ORDER BY id DESC;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan Backward using idx_test_id on test
                                                (cost=0.44..12.26 rows=191 width=4)
   Index Cond: ((id >= 1000) AND (id <= 1200))
```

3.1.5 Index : inconvénients

- L'index n'est pas gratuit !
- Ralentit les écritures
  - maintenance
- Place disque
- Compromis à trouver

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table.

Les index dégradent surtout les temps de réponse des insertions. Les mises à jour et les suppressions (`UPDATE` et `DELETE`) tirent en général parti des index pour retrouver les lignes concernées par les modifications. Le coût de maintenance de l'index est secondaire par rapport au coût de l'accès aux données.

Soit une table `test2` telle que :

```
CREATE TABLE test2 (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);
```

La table est chargée avec pour seul index présent celui sur la clé primaire :

```
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 35253,228 ms (00:35,253)
```

Un index supplémentaire est créé sur une colonne de type entier :

```
CREATE INDEX idx_test2_valeur ON test2 (valeur);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 44410,775 ms (00:44,411)
```

Un index supplémentaire est encore créé, mais cette fois sur une colonne de type texte :

```
CREATE INDEX idx_test2_commentaire ON test2 (commentaire);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 207075,335 ms (03:27,075)
```

On peut comparer ces temps à l'insertion dans une table similaire dépourvue d'index :

```
CREATE TABLE test3 AS SELECT * FROM test2;
INSERT INTO test3 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 14758,503 ms (00:14,759)
```

La table `test2` a été vidée préalablement pour chaque test.

Enfin, la place disque utilisée par ces index n'est pas négligeable :

```
\di+ *test2*
                                 Liste des relations
 Schéma |          Nom           | Type  | Propriétaire | Table | Taille |  …
--------+------------------------+-------+--------------+-------+--------+-
 public | idx_test2_commentaire  | index | postgres     | test2 | 387 MB |
 public | idx_test2_valeur       | index | postgres     | test2 | 214 MB |
 public | test2_pkey             | index | postgres     | test2 | 214 MB |
```

```
SELECT pg_size_pretty(pg_relation_size('test2')),
       pg_size_pretty(pg_indexes_size('test2')) ;
```

```
 pg_size_pretty | pg_size_pretty
----------------+----------------
 574 MB         | 816 MB
```

Pour ces raisons, on ne posera pas des index systématiquement avant de se demander s'ils seront utilisés. L'idéal est d'étudier les plans de ses requêtes et de chercher à optimiser.

3.1.6 Index : contraintes pratiques à la création

- Lourd…

```
-- bloque les écritures !
CREATE INDEX ON matable ( macolonne ) ;
-- ne bloque pas, peut échouer
CREATE INDEX CONCURRENTLY ON matable ( macolonne ) ;
```

- Si fragmentation :

```
REINDEX INDEX nomindex ;
REINDEX TABLE CONCURRENTLY nomtable ;
```

- Paramètres :
  - `maintenance_work_mem` (sinon : fichier temporaire !)
  - `max_parallel_maintenance_workers`

Création d'un index :

Bien sûr, la durée de création de l'index dépend fortement de la taille de la table. PostgreSQL va lire toutes les lignes et trier les valeurs rencontrées. Ce peut être lourd et impliquer la création de fichiers temporaires.

Si l'on utilise la syntaxe classique, toutes les écritures sur la table sont bloquées (mises en attente) pendant la durée de la création de l'index (verrou *ShareLock*). Les lectures restent possibles, mais cette contrainte est parfois rédhibitoire pour les grosses tables.

Clause CONCURRENTLY :

Ajouter le mot clé `CONCURRENTLY` permet de rendre la table accessible en écriture. Malheureusement, cela nécessite au minimum deux parcours de la table, et donc alourdit et ralentit la construction de l'index. Dans quelques cas défavorables (entre autres l'interruption de la création de l'index), la création échoue et l'index existe mais est invalide :

```
pgbench=# \d pgbench_accounts
                Table « public.pgbench_accounts »
 Colonne  |     Type      | Collationnement | NULL-able | Par défaut
----------+---------------+-----------------+-----------+------------
 aid      | integer       |                 | not null  |
 bid      | integer       |                 |           |
 abalance | integer       |                 |           |
 filler   | character(84) |                 |           |
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
    "pgbench_accounts_bid_idx" btree (bid) INVALID
```

L'index est inutilisable et doit être supprimé et recréé, ou bien réindexé. Pour les détails, voir la documentation officielle[3].

Une supervision peut détecter des index invalides avec cette requête, qui ne doit jamais rien ramener :

```
SELECT indexrelid::regclass AS index, indrelid::regclass AS table
FROM pg_index
WHERE indisvalid = false ;
```

Réindexation :

Comme les tables, les index sont soumis à la fragmentation. Celle-ci peut cependant monter assez haut sans grande conséquence pour les performances. De plus, le nettoyage des index est une des étapes des opérations de VACUUM[4].

Une reconstruction de l'index est automatique lors d'un `VACUUM FULL` de la table.

Certaines charges provoquent une fragmentation assez élevée, typiquement les tables gérant des files d'attente. Une réindexation reconstruit totalement l'index. Voici quelques variantes de l'ordre :

```
REINDEX INDEX pgbench_accounts_bid_idx ;  -- un seul index
REINDEX TABLE pgbench_accounts ;          -- tous les index de la table
REINDEX (VERBOSE) DATABASE pgbench ;      -- tous ceux de la base, avec détails
```

Il existe là aussi une clause `CONCURRENTLY` :

```
REINDEX (VERBOSE) INDEX CONCURRENTLY pgbench_accounts_bid_idx ;
```

(En cas d'échec, on trouvera là aussi des index invalides, suffixés avec `_ccnew`, à côté des index préexistants toujours fonctionnels et que PostgreSQL n'a pas détruits.)

Paramètres :

La rapidité de création d'un index dépend essentiellement de la mémoire accordée, définie dans `maintenance_work_mem`. Si elle ne suffit pas, le tri se fera dans des fichiers temporaires plus lents. Sur les serveurs modernes, le défaut de 64 Mo est ridicule, et on peut monter aisément à :

```
SET maintenance_work_mem = '2GB' ;
```

Attention de ne pas saturer la mémoire en cas de création simultanée de nombreux gros index (lors d'une restauration avec `pg_restore` notamment).

Si le serveur est bien doté en CPU, la parallélisation de la création d'index peut apporter un gain en temps appréciable. La valeur par défaut est :

```
SET max_parallel_maintenance_workers = 2 ;
```

et devrait même être baissée sur les plus petites configurations.

[3] https://docs.postgresql.fr/current/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY
[4] https://dali.bo/m5_html#fonctionnement-de-vacuum

3.1.7 Types d'index dans PostgreSQL

- Défaut : B-tree classique (équilibré)
- `UNIQUE` (préférer la contrainte)
- Mais aussi multicolonne, fonctionnel, partiel, couvrant
- Index spécialisés : hash, GiST, GIN, BRIN, HNSW….

Par défaut un `CREATE INDEX` créera un index de type B-tree, de loin le plus courant. Il est stocké sous forme d'arbre équilibré, avec de nombreux avantages :

- les performances se dégradent peu avec la taille de l'arbre (les temps de recherche sont en O(log(n)), donc fonction du logarithme du nombre d'enregistrements dans l'index) ;
- l'accès concurrent est excellent, avec très peu de contention entre processus qui insèrent simultanément.

Toutefois les B-tree ne permettent de répondre qu'à des questions très simples, portant sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre tout de même la majorité des cas.

Contrainte d'unicité et index :

Un index peut être déclaré `UNIQUE` pour provoquer une erreur en cas d'insertion de doublons. Mais on préférera généralement déclarer une *contrainte* d'unicité (notion fonctionnelle), qui techniquement, entraînera la création d'un index.

Par exemple, sur cette table `personne` :

```
$ CREATE TABLE personne (id int, nom text);

$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
```

on peut créer un index unique :

```
$ CREATE UNIQUE INDEX ON personne (id);

$ \d personne
                 Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "personne_id_idx" UNIQUE, btree (id)
```

La contrainte d'unicité est alors implicite. La suppression de l'index se fait sans bruit :

```
DROP INDEX personne_id_idx;
```

Définissons une contrainte d'unicité sur la colonne plutôt qu'un index :

```
ALTER TABLE personne ADD CONSTRAINT unique_id UNIQUE (id);
```

```
$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "unique_id" UNIQUE CONSTRAINT, btree (id)
```

Un index est également créé. La contrainte empêche sa suppression :

```
DROP INDEX unique_id ;
```

```
ERREUR:  n'a pas pu supprimer index unique_id car il est requis par contrainte
 unique_id sur table personne
ASTUCE : Vous pouvez supprimer contrainte unique_id sur table personne à la
 place.
```

Le principe est le même pour les clés primaires.

Indexation avancée :

Il faut aussi savoir que PostgreSQL permet de créer des index B-tree :

- sur plusieurs colonnes ;
- sur des résultats de fonction ;
- sur une partie des valeurs indexées ;
- intégrant des champs non indexés mais souvent récupérés avec les champs indexés (index couvrants).

D'autres types d'index que B-tree existent, destinés à certains types de données ou certains cas d'optimisation précis.

3.2 FONCTIONNEMENT D'UN INDEX

3.2.1 Structure d'un index

- Structure associant des clés (termes) à des localisations (pages)
- Structure de données spécialisée, plusieurs types
- Séparée de la table
- Analogies :
  - fiches en carton des bibliothèques avant l'informatique (B-tree)
  - index d'un livre technique (GIN)

Les fiches en carton des anciennes bibliothèques sont un bon équivalent du type d'index le plus courant utilisé par les bases de données en général et PostgreSQL en particulier : le B-tree.

Lorsque l'on recherche des ouvrages dans la bibliothèque, il est possible de parcourir l'intégralité du bâtiment pour chercher les livres qui nous intéressent. Ceci prend énormément de temps. La bibliothèque peut être triée, mais ce tri ne permet pas forcément de trouver facilement le livre. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (*Seq Scan*).

Une deuxième méthode pour localiser l'ouvrage consiste à utiliser un index. Sur fiche carton ou sous forme informatique, cet index associe par exemple le nom d'auteur à un ensemble de références (emplacements dans les rayonnages) où celui-ci est présent. Ainsi, pour trouver les œuvres de Proust avec l'index en carton, il suffit de parcourir les fiches, dont l'intégralité tient devant l'utilisateur. La fiche indique des références dans plusieurs rayons et il faudra aller se déplacer pour trouver les œuvres, en allant directement aux bons rayons.

Dans une base de données, le fonctionnement d'un index est très similaire. En effet, comme dans une bibliothèque, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est principalement utilisée pour la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données entraîne des modifications de l'index afin de le maintenir à jour. Et un index qui n'est pas à jour peut provoquer de gros problèmes. Dans le doute, on peut jeter l'index et le recréer de zéro sans problème d'intégrité des données originales.

Il peut y avoir plusieurs index suivant les besoins. L'index trié par auteur ne permet pas de trouver un livre dont on ne connaît que le titre (sauf à lire toutes les fiches). Il faut alors un autre index classé par titre.

Pour filer l'analogie : un index peut être multicolonne (les fiches en carton triées par auteur le sont car elles contiennent le titre, et pas que la référence dans les rayons). L'index peut répondre à une demande à lui seul : il suffit pour compter le nombre de livres de Marcel Proust (c'est le principe des *Index Only Scans*). Une fiche d'un index peut contenir des informations supplémentaires (dates de publication, éditeur…) pour faciliter d'autres recherches sans aller dans les rayons (index « couvrant »).

Dans la réalité comme dans une base de données, il y a un dilemme quand il faut récupérer de très nombreuses données : soit aller chercher de nombreux livres un par un dans les rayons, soit balayer tous les livres systématiquement dans l'ordre où ils viennent pour éviter trop d'allers-retours.

Autres types d'index non informatiques similaires au B-tree :

- les tables décennales de l'État civil, qui pointent vers un endroit précis des registres des actes de naissance, mariage ou décès d'une commune ;
- l'index d'un catalogue papier.

L'index d'un livre technique ou d'un livre de recettes cible des parties des données et non les données elles-mêmes (comme le titre). Il s'approche plus d'un autre type d'index, le GIN, qui existe aussi dans PostgreSQL.

Un annuaire téléphonique papier présente les données sous un mode strictement ordonné. Cette intégration entre table et index n'a pas d'équivalent sous PostgreSQL mais existe dans d'autres moteurs de bases de données.

3.2.2 Un index n'est pas magique

- Un index ne résout pas tout
- Importance de la conception du schéma de données
- Importance de l'écriture de requêtes SQL correctes

Bien souvent, la création d'index est vue comme le remède à tous les maux de performance subis par une application. Il ne faut pas perdre de vue que les facteurs principaux affectant les performances vont être liés à la conception du schéma de données, et à l'écriture des requêtes SQL.

Pour prendre un exemple caricatural, un schéma EAV (*Entity-Attribute-Value*, ou *entité-clé-valeur*) ne pourra jamais être performant, de part sa conception. Bien sûr, dans certains cas, une méthodologie pertinente d'indexation permettra d'améliorer un peu les performances, mais le problème réside là dans la conception même du schéma. Il est donc important dans cette phase de considérer la manière dont le modèle va influer sur les méthodes d'accès aux données, et les implications sur les performances.

De même, l'écriture des requêtes elles-mêmes conditionnera en grande partie les performances observées sur l'application. Par exemple, la mauvaise pratique (souvent mise en œuvre accidentellement via un ORM) dite du « N+1 » ne pourra être corrigée par une indexation correcte : celle-ci consiste à récupérer une collection d'enregistrement (une requête) puis d'effectuer une requête pour chaque enregistrement afin de récupérer les enregistrements liés (N requêtes). Dans ce type de cas, une jointure est bien plus performante. Ce type de comportement doit encore une fois être connu de l'équipe de développement, car il est plutôt difficile à détecter par une équipe d'exploitation.

De manière générale, avant d'envisager la création d'index supplémentaires, il convient de s'interroger sur les possibilités de réécriture des requêtes, voire du schéma.

3.2.3 Index B-tree

- Type d'index le plus courant
  - et le plus simple
- Utilisable pour les contraintes d'unicité
- Supporte les opérateurs : `<`, `<=`, `=`, `>=`, `>`
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2,6 ko

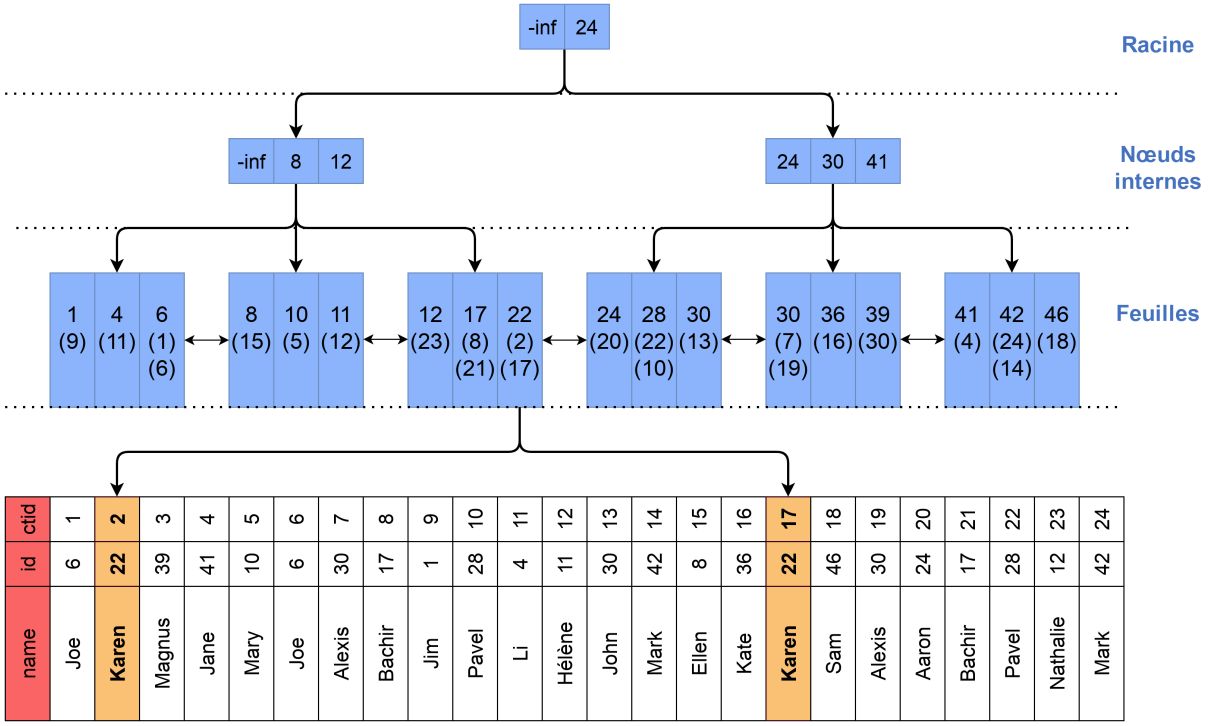
L'index B-tree est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index B-tree est par définition équilibré : ainsi, quelle que soit la valeur recherchée, le coût est le même lors du

parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps ! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'⅓ de bloc, soit environ 2,6 ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera renvoyée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Ces champs sont souvent des longs textes ou des champs composés dont on cherchera plutôt des parties, et un index B-tree n'est de toute façon pas adapté. Si un index de type B-tree est tout de même nécessaire sur les colonnes en question, pour des recherches sur l'intégralité de la ligne, les index de type hash sont plus adaptés (mais ils ne supportent que l'opérateur `=`).

3.2.4 Exemple de structure d'index

```
SELECT name FROM ma_table WHERE id = 22
```



Ce schéma présente une vue très simplifiée d'une table (en blanc, avec ses champs `id` et `name`) et d'un index B-tree sur `id` (en bleu), tel que le créerait :

```
CREATE INDEX mon_index ON ma_table (id) ;
```

Un index B-tree peut contenir trois types de nœuds :

- la racine : elle est unique c'est la base de l'arbre ;
- des nœuds internes : il peut y en avoir plusieurs niveaux ;
- des feuilles : elles contiennent :
  - les valeurs indexées (triées !) ;
  - les valeurs incluses (si applicable) ;
  - les positions physiques (`ctid`), ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc) ;
  - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (`ctid`).

Lors de la création de l'index, il ne contient qu'une feuille. Lorsque cette feuille se remplit, elle se divise en deux et un nœud racine est créé au-dessus. Les feuilles se remplissent ensuite progressivement et se séparent en deux quand elles sont pleines. Ce processus remplit progressivement la racine. Lorsque la racine est pleine, elle se divise en deux nœuds internes, et une nouvelle racine est crée au-dessus. Ce processus permet de garder un arbre équilibré.

Recherchons le résultat de :

```
SELECT name FROM ma_table WHERE id = 22
```

en passant par l'index.

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles). On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Un arbre B-tree peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère. Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17. Lorsque la valeur recherchée est supérieure ou égale à la plus grande valeur du bloc, PostgreSQL va également lire le bloc suivant. Ce cas de figure peut se produire si PostgreSQL a divisé une feuille en deux avant ou même pendant la recherche que nous exécutons. Ce serait par exemple le cas si on cherchait la valeur 30.
- Pour trouver les valeurs de `name`, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index. D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table. (Il existe des cas où la recherche peut éviter cette dernière étape : ce sont les *Index Only Scan*.)

Même en parcourant les deux structures de données, si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera donc fortement réduit. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs classiques sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre. Le choix est fait par l'optimiseur.

Supposons désormais que nous souhaitions exécuter une requête sans filtre, mais exigeant un tri, du type :

```
SELECT id FROM ma_table ORDER BY id ;
```

L'index peut nous aider à répondre à cette requête. En effet, toutes les feuilles sont liées entre elles, et permettent ainsi un parcours ordonné. Il nous suffit donc de localiser la première feuille (la plus à gauche), et pour chaque clé, récupérer les lignes correspondantes. Une fois les clés de la feuille traitées, il suffit de suivre le pointeur vers la feuille suivante et de recommencer.

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peut être très coûteux, en mémoire comme en temps CPU. D'ailleurs,

de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut là aussi être utilisé. Par exemple, pour la requête suivante :

```
SELECT * FROM ma_table WHERE id <= 10 AND id >= 4 ;
```

Il suffit d'utiliser la propriété de tri de l'index pour parcourir les feuilles, en partant de la borne inférieure, jusqu'à la borne supérieure.

Dernière remarque : ce schéma ne montre qu'une entrée d'index pour 22, bien qu'il pointe vers deux lignes. En fait, il y avait bien deux entrées pour 22 avant PostgreSQL 13. Depuis cette version, PostgreSQL sait dédupliquer les entrées pour économiser de la place.

3.2.5 Index multicolonnes

- Possibilité d'indexer plusieurs colonnes :

```
CREATE INDEX ON ma_table (id, name) ;
```

- Ordre des colonnes **primordial**
  - accès direct aux premières colonnes de l'index
  - pour les autres, PostgreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée une table d'un million de lignes avec un index sur trois champs :

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 text);

INSERT INTO t1 (c1, c2, c3, c4)
SELECT i*10,j*5,k*20, 'text'||i||j||k
FROM generate_series (1,100) i
CROSS JOIN generate_series(1,100) j
CROSS JOIN generate_series(1,100) k ;

CREATE INDEX ON t1 (c1, c2, c3) ;

VACUUM ANALYZE t1 ;

-- Figer des paramètres pour l'exemple
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

L'index est optimal pour répondre aux requêtes portant sur les premières colonnes de l'index :

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 1000 and c2=500 and c3=2000 ;

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..8.45 rows=1 width=22)
   Index Cond: ((c1 = 1000) AND (c2 = 500) AND (c3 = 2000))
```

Et encore plus quand l'index permet de répondre intégralement au contenu de la requête :

```
EXPLAIN SELECT c1,c2,c3 FROM t1 WHERE c1 = 1000 and c2=500 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Only Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..6.33 rows=95 width=12)
   Index Cond: ((c1 = 1000) AND (c2 = 500))
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie.

Cela reste plus intéressant que parcourir toute la table, surtout si l'index est petit et contient toutes les données du `SELECT`. Mais le comportement dépend alors de nombreux paramètres, comme les statistiques, les estimations du nombre de lignes ramenées et les valeurs relatives de `seq_page_cost` et `random_page_cost` :

```
SET random_page_cost TO 0.1 ; SET seq_page_cost TO 0.1 ;  -- SSD

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (...) (...)
   Index Cond: (c3 = 2000)
   Buffers: shared hit=3899
 Planning:
   Buffers: shared hit=15
 Planning Time: 0.218 ms
 Execution Time: 67.081 ms
```

Noter que tout l'index a été lu.

Mais pour limiter les aller-retours entre index et table, PostgreSQL peut aussi décider d'ignorer l'index et de parcourir directement la table :

```
SET random_page_cost TO 4 ; SET seq_page_cost TO 1 ;  -- défaut (disque mécanique)

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
Seq Scan on t1  (cost=0.00..18871.00 rows=9600 width=22) (...)
   Filter: (c3 = 2000)
   Rows Removed by Filter: 990000
   Buffers: shared hit=6371
 Planning Time: 0.178 ms
 Execution Time: 114.572 ms
```

Concernant les *range scans* (requêtes impliquant des opérateurs d'inégalité, tels que `<`, `<=`, `>=`, `>`), celles-ci pourront être satisfaites par l'index de manière quasi optimale si les opérateurs d'inégalité sont appliqués sur la dernière colonne requêtée, et de manière sub-optimale s'ils portent sur les premières colonnes.

Cet index pourra être utilisé pour répondre aux requêtes suivantes de manière optimale :

```
SELECT * FROM t1 WHERE c1 = 20 ;
SELECT * FROM t1 WHERE c1 = 20 AND c2 = 50 AND c3 = 400 ;
SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

Il pourra aussi être utilisé, mais de manière bien moins efficace, pour les requêtes suivantes, qui bénéficieraient d'un index sur un ordre alternatif des colonnes :

```
SELECT * FROM t1 WHERE c1 = 100 AND c2 >= 80 AND c3 = 40 ;
SELECT * FROM t1 WHERE c1 < 100 AND c2 = 100 ;
```

Le plan de cette dernière requête est :

```
 Bitmap Heap Scan on t1  (cost=2275.98..4777.17 rows=919 width=22) (...)
   Recheck Cond: ((c1 < 100) AND (c2 = 100))
   Heap Blocks: exact=609
   Buffers: shared hit=956
   ->  Bitmap Index Scan on t1_c1_c2_c3_idx  (cost=0.00..2275.76 rows=919 width=0)
↪  (...)
         Index Cond: ((c1 < 100) AND (c2 = 100))
         Buffers: shared hit=347
 Planning Time: 0.227 ms
 Execution Time: 15.596 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Il n'y a pas besoin de trier (ce peut être très coûteux) puisque les données de l'index sont triées. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

```
SELECT * FROM t1 ORDER BY c1 ;
SELECT * FROM t1 ORDER BY c1, c2 ;
SELECT * FROM t1 ORDER BY c1, c2, c3 ;
```

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..55893.66 rows=1000000 width=22)
↪  (...)
   Buffers: shared hit=1003834
 Planning Time: 0.282 ms
 Execution Time: 425.520 ms
```

Il est donc nécessaire d'avoir une bonne connaissance de l'application (ou de passer du temps à observer les requêtes consommatrices) pour déterminer comment créer des index multicolonnes pertinents pour un nombre maximum de requêtes.

3.2.6 Nœuds des index

- *Index Scan*
- *Bitmap Scan*
- *Index Only Scan*
  - idéal pour les performances
- et les variantes parallélisées

L'optimiseur a le choix entre plusieurs parcours pour utiliser un index, principalement suivant la quantité d'enregistrements à récupérer :

3.2.6.1 Index Scan

Un *Index Scan* est optimal quand il y a peu d'enregistrements à récupérer. Noter qu'il comprend l'accès à l'index *et* celui à la table ensuite.

3.2.6.2 Bitmap Scan

Le *Bitmap Scan* est utile quand il y a plus de lignes, ou quand on veut lire plusieurs index d'une même table pour satisfaire plusieurs conditions de filtre.

Il se décompose en deux nœuds : un *Bitmap Index Scan* qui récupère des blocs d'index, et un *Bitmap Heap Scan* qui va chercher les blocs dans la table.

Typiquement, ce nœud servira pour des recherches de plages de valeurs ou de grandes quantités de lignes. Il est favorisé par une bonne corrélation des données avec leur emplacement physique.

3.2.6.3 Index Only Scan

L'*Index Only Scan* est utile quand les champs de la requête correspondent aux colonnes de l'index. Ce nœud permet d'éviter la lecture de tout ou partie de la table et est donc très performant.

Autre intérêt de l'*Index Only Scan* : les enregistrements cherchés sont contigus dans l'index (puisqu'il est trié), et le nombre d'accès disque est bien plus faible. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Si peu de champs de la table sont impliqués dans la requête, il faut penser à viser un *Index Only Scan*.

3.2.6.4 Parallélisation

Chacun de ses nœuds a une version parallélisable si l'index est assez grand et que l'optimiseur pense que paralléliser est utile. Il apparaît alors un nœud *Gather* pour rassembler les résultats des différents *workers*.

3.3 MÉTHODOLOGIE DE CRÉATION D'INDEX

- On indexe pour une requête
  - ou idéalement une collection de requêtes
- Et pas « une table »

La première chose à garder en tête est que l'on indexe pas le schéma de données, c'est-à-dire les tables, mais en fonction de la charge de travail supportée par la base, c'est-à-dire les requêtes. En effet, comme nous l'avons vu précédemment, tout index superflu a un coût global pour la base de données, notamment pour les opérations DML.

3.3.1 L'index ? Quel index ?

- Identifier les requêtes nécessitant un index
- Créer les index permettant de répondre à ces requêtes
- Valider le fonctionnement, en rejouant la requête avec :

```
EXPLAIN (ANALYZE, BUFFERS)
```

La méthodologie elle-même est assez simple. Selon le principe qu'un index sert à une (ou des) requête(s), la première chose à faire consiste à identifier celle(s)-ci. L'équipe de développement est dans une position idéale pour réaliser ce travail : elle seule peut connaître le fonctionnement global de l'application, et donc les colonnes qui vont être utilisées, ensemble ou non, comme cible de filtres ou de tris. Au delà de la connaissance de l'application, il est possible d'utiliser des outils tels que pgBadger, pg_stat_statements et PoWA pour identifier les requêtes particulièrement consommatrices, et qui pourraient donc potentiellement nécessiter un index. Ces outils seront présentés plus loin dans cette formation.

Une fois les requêtes identifiées, il est nécessaire de trouver les index permettant d'améliorer celles-ci. Ils peuvent être utilisés pour les opérations de filtrage (clause `WHERE`), de tri (clauses `ORDER BY`, `GROUP BY`) ou de jointures. Idéalement, l'étude portera sur l'ensemble des requêtes, afin notamment de pouvoir décider d'index multicolonnes pertinents pour le plus grand nombre de requêtes, et éviter ainsi de créer des index redondants.

3.3.2 Index et clés étrangères

- Indexation des colonnes faisant référence à une autre
- Performances des DML
- Performances des jointures

De manière générale, l'ensemble des colonnes étant la source d'une clé étrangère devraient être indexées, et ce pour deux raisons.

La première concerne les jointures. Généralement, lorsque deux tables sont liées par des clés étrangères, il existe au moins certaines requêtes dans l'application joignant ces tables. La colonne « cible » de la clé étrangère est nécessairement indexée, c'est un prérequis dû à la contrainte unique nécessaire à celle-ci. Il est donc possible de la parcourir de manière triée.

La colonne source devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme *Merge Join* (comme vu lors du module sur les plans d'exécution[5]), et donc d'être beaucoup plus rapide. Un tel index accélérera de la même manière les *Nested Loop*, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

De la même manière, pour les DML sur la table cible, cet index sera d'une grande aide : pour chaque ligne modifiée ou supprimée, il convient de vérifier, soit pour interdire soit pour « cascader » la modification, la présence de lignes faisant référence à celle touchée.

S'il n'y a qu'une règle à suivre aveuglément ou presque, c'est bien celle-ci : les colonnes faisant partie d'une clé étrangère doivent être indexées !

Deux exceptions : les champs ayant une cardinalité très faible et homogène (par exemple, un champ homme/femme dans une population équilibrée) ; et ceux dont on constate l'inutilité après un certain temps, par des valeurs à zéro dans `pg_stat_user_indexes`.

[5] https://dali.bo/j0_html

3.4 INDEX INUTILISÉ

C'est souvent tout à fait normal

- Utiliser l'index est-il rentable ?
- La requête est-elle compatible ?
- Bug de l'optimiseur : rare

C'est l'optimiseur SQL qui choisit si un index doit ou non être utilisé. Il est tout à fait possible que PostgreSQL décide qu'utiliser un index donné n'en vaut pas la peine par rapport à d'autres chemins. Il faut aussi savoir identifier les cas où l'index ne peut *pas* être utilisé.

L'optimiseur possède forcément quelques limitations. Certaines sont un compromis par rapport au temps que prendrait la recherche systématique de toutes les optimisations imaginables. Il y aussi le problème des estimations de volumétries, qui sont d'autant plus difficiles que la requête est complexe.

Quant à un vrai bug, si le cas peut être reproduit, il doit être remonté aux développeurs de PostgreSQL. D'expérience, c'est rarissime.

3.4.1 Index utilisable mais non utilisé

- L'optimiseur pense qu'il n'est pas rentable
  - sélectivité trop faible
  - meilleur chemin pour remplir d'autres critères
  - index redondant
  - *Index Only Scan* nécessite un `VACUUM` fréquent
- Les estimations de volumétries doivent être assez bonnes !
  - statistiques récentes, précises

Il existe plusieurs raisons pour que PostgreSQL néglige un index.

Sélectivité trop faible, trop de lignes :

Comme vu précédemment, le parcours d'un index implique à la fois des lectures sur l'index, et des lectures sur la table. Au contraire d'une lecture séquentielle de la table (*Seq Scan*), l'accès aux données via l'index nécessite des lectures aléatoires. Ainsi, si l'optimiseur estime que la requête nécessitera de parcourir une grande partie de la table, il peut décider de ne pas utiliser l'index : l'utilisation de celui-ci serait alors trop coûteux.

Autrement dit, l'index n'est pas assez discriminant pour que ce soit la peine de faire des allers-retours entre lui et la table. Le seuil dépend entre autres des volumétries de la table et de l'index et du rapport entre les paramètres `random_page_cost` et `seq_page_cost` (respectivement 4 et 1 pour un disque dur classique peu rapide, et souvent 1 et 1 pour du SSD, voire moins).

Il y a un meilleur chemin :

Un index sur un champ n'est qu'un chemin parmi d'autres, en aucun cas une obligation, et une requête contient souvent plusieurs critères sur des tables différentes. Par exemple, un index sur un filtre peut être ignoré si un autre index permet d'éviter un tri coûteux, ou si l'optimiseur juge que faire une jointure avant de filtrer le résultat est plus performant.

Index redondant :

Il existe un autre index doublant la fonctionnalité de celui considéré. PostgreSQL favorise naturellement un index plus petit, plus rapide à parcourir. À l'inverse, un index plus complet peut favoriser plusieurs filtres, des tris, devenir couvrant…

VACUUM trop ancien :

Dans le cas précis des *Index Only Scan*, si la table n'a pas été récemment nettoyée, il y aura trop d'allers-retours avec la table pour vérifier les informations de visibilité (*heap fetches*). Un `VACUUM` permet de mettre à jour la *Visibility Map* pour éviter cela.

Statistiques périmées :

Il peut arriver que l'optimiseur se trompe quand il ignore un index. Des statistiques périmées sont une cause fréquente. Pour les rafraîchir :

```
ANALYZE (VERBOSE) nom_table;
```

Si cela résout le problème, ce peut être un indice que l'autovacuum ne passe pas assez souvent (voir `pg_stat_user_tables.last_autoanalyze`). Il faudra peut-être ajuster les paramètres `autovacuum_analyze_scale_factor` ou `autovacuum_analyze_threshold` sur les tables.

Statistiques pas assez fines :

Les statistiques sur les données peuvent être trop imprécises. Le défaut est un histogramme de 100 valeurs, basé sur 300 fois plus de lignes. Pour les grosses tables, augmenter l'échantillonnage sur les champs aux valeurs peu homogènes est possible :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500 ;
```

La valeur 500 n'est qu'un exemple. Monter beaucoup plus haut peut pénaliser les temps de planification. Ce sera d'autant plus vrai si on applique cette nouvelle valeur globalement, donc à tous les champs de toutes les tables (ce qui est certes le plus facile).

Estimations de volumétries trompeuses :

Par exemple, une clause `WHERE` sur deux colonnes corrélées (ville et code postal par exemple), mène à une sous-estimation de la volumétrie résultante par l'optimiseur, car celui-ci ignore le lien entre

les deux champs. Vous pouvez demander à PostgreSQL de calculer cette corrélation avec l'ordre `CREATE STATISTICS` (voir le module de formation J2[6] ou la documentation officielle[7]).

Compatibilité :

Il faut toujours s'assurer que la requête est écrite correctement et permet l'utilisation de l'index.

Un index peut être inutilisable à cause d'une fonction plus ou moins explicite, ou encore d'un mauvais typage. Il arrive que le critère de filtrage ne peut remonter sur la table indexée à cause d'un CTE matérialisé (explicitement ou non), d'un `DISTINCT`, ou d'une vue complexe.

Nous allons voir quelques problèmes classiques.

3.4.2 Index inutilisable à cause d'une fonction

- Pas le bon type (`CAST` plus ou moins explicite)

```
EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;
```

- Utilisation de fonctions, comme :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

Voici quelques exemples d'index incompatible avec la clause `WHERE` :

Mauvais type :

Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre. Cela dépend des types et du sens de conversion. Dans les exemples suivants, le champ `client_id` est de type `bigint`. PostgreSQL réussit souvent à convertir, mais ce n'est pas toujours parfait.

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = 3)
```

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3::numeric;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on clients
   Filter: ((client_id)::numeric = '3'::numeric)
```

[6] https://dali.bo/j2_html
[7] https://docs.postgresql.fr/current/sql-createstatistics.html

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3::int;

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = 3)

EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = '003';

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = '3'::bigint)
```

De même, les conversions entre `date` et `timestamp`/`timestamptz` se passent généralement bien.

Autres exemples :

- Dans une jointure, si les deux champs joints n'ont pas le même type, il est possible que de simples index ne soient pas utilisables, ou un seul d'entre eux. Il faudra corriger l'incohérence, ou créer des index fonctionnels incluant le transtypage.
- Un index B-tree sur un tableau ou un JSON ne peut servir pour une recherche sur un de ses éléments. Il faudra s'orienter vers un index plus spécialisé, par exemple GIN ou GiST.

Utilisation de fonction :

Si une fonction est appliquée sur la colonne à indexer, comme dans cet exemple classique :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

alors PostgreSQL n'utilisera pas l'index sur `ma_date`. Il faut réécrire la requête ainsi :

```
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' AND ma_date<'2015-01-01' ;
```

Dans l'exemple suivant, on cherche les commandes dont la date tronquée au mois correspond au 1er janvier, c'est-à-dire aux commandes dont la date est entre le 1er et le 31 janvier. Pour un humain, la logique est évidente, mais l'optimiseur n'en a pas connaissance.

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_trunc('month', date_commande) = '2015-01-01';

                                QUERY PLAN
--------------------------------------------------------------------------
Gather  (cost=1000.00..8160.96 rows=5000 width=51)
        (actual time=17.282..192.131 rows=4882 loops=1)
 Workers Planned: 3
 Workers Launched: 3
 ->  Parallel Seq Scan on commandes (cost=0.00..6660.96 rows=1613 width=51)
                           (actual time=17.338..177.896 rows=1220 loops=4)
     Filter: (date_trunc('month'::text,
                                    (date_commande)::timestamp with time zone)
                           = '2015-01-01 00:00:00+01'::timestamp with time zone)
     Rows Removed by Filter: 248780
 Planning time: 0.215 ms
 Execution time: 196.930 ms
```

Il faut plutôt écrire :

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31' ;

                           QUERY PLAN
-----------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
                                     (cost=0.42..118.82 rows=5554 width=51)
                                (actual time=0.019..0.915 rows=4882 loops=1)
   Index Cond: ((date_commande >= '2015-01-01'::date)
           AND (date_commande <= '2015-01-31'::date))
 Planning time: 0.074 ms
 Execution time: 1.098 ms
```

Dans certains cas, la réécriture est impossible (fonction complexe, code non modifiable…). Nous verrons qu'un index fonctionnel peut parfois être la solution.

Ces exemples semblent évidents, mais il peut être plus compliqué de trouver dans l'urgence la cause du problème dans une grande requête d'un schéma mal connu.

3.4.3 Index inutilisable à cause d'un LIKE '…%'

```
SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';
```

- Solution :

```
CREATE INDEX idx1 ON ma_table (col_varchar varchar_pattern_ops) ;
```

Si vous avez un index « normal » sur une chaîne texte, certaines recherches de type `LIKE` n'utiliseront pas l'index. En effet, il faut bien garder à l'esprit qu'un index est basé sur un opérateur précis. Ceci est généralement indiqué correctement dans la documentation, mais pas forcément très intuitif.

Si un opérateur non supporté pour le critère de tri est utilisé, l'index ne servira à rien :

```
CREATE INDEX ON fournisseurs (commentaire);
EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';

                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on fournisseurs  (cost=0.00..225.00 rows=1 width=45)
                           (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
   Rows Removed by Filter: 9953
 Planning time: 0.085 ms
 Execution time: 1.509 ms
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente, et que `varchar_pattern_ops` est l'opérateur permettant d'indexer la requête précédente.

3.4.4 Index inutilisable car invalide

– `CREATE INDEX … CONCURRENTLY` peut échouer

Dans le cas où un index a été construit avec la clause `CONCURRENTLY`, nous avons vu qu'il peut arriver que l'opération échoue et l'index existe mais reste invalide, et donc inutilisable. Le problème ne se pose pas pour un échec de `REINDEX … CONCURRENTLY`, car l'ancienne version de l'index est toujours là et utilisable.

3.5 INDEXATION B-TREE AVANCÉE

De nombreuses possibilités d'indexation avancée :

- Index partiels
- Index fonctionnels
- Index couvrants
- Classes d'opérateur

3.5.1 Index partiels

- N'indexe qu'une partie des données :

```
CREATE INDEX on evenements (type) WHERE  traite IS FALSE ;
```

- Ne sert que si la clause est logiquement équivalente !
  - ou partie de la clause (inégalités, `IN`)
- Intérêt : index beaucoup plus petit

Un index partiel est un index ne couvrant qu'une partie des enregistrements. Ainsi, l'index est beaucoup plus petit. En contrepartie, il ne pourra être utilisé que si sa condition est définie dans la requête.

Pour prendre un exemple simple, imaginons un système de « queue », dans lequel des événements sont entrés, et qui disposent d'une colonne `traite` indiquant si oui ou non l'événement a été traité. Dans le fonctionnement normal de l'application, la plupart des requêtes ne s'intéressent qu'aux événements non traités :

```
CREATE TABLE evenements (
    id int primary key,
    traite bool NOT NULL,
    type text NOT NULL,
    payload text
);

-- 10 000 événements traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        true,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
```

```
    FROM generate_series(1, 10000) as i);

-- et 10  non encore traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        false,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(10001, 10010) as i);

\d evenements

                Table « public.evenements »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 | not null  |
 traite  | boolean |                 | not null  |
 type    | text    |                 | not null  |
 payload | text    |                 |           |
Index :
    "evenements_pkey" PRIMARY KEY, btree (id)
```

Typiquement, différents applicatifs vont être intéressés par des événements d'un certain type, mais les événements déjà traités ne sont quasiment jamais accédés, du moins via leur état (une requête portant sur `traite IS true` sera exceptionnelle et ramènera l'essentiel de la table : un index est inutile).

Ainsi, on peut souhaiter indexer le type d'événement, mais uniquement pour les événements non traités :

```
CREATE INDEX index_partiel on evenements (type) WHERE NOT traite ;
```

Si on recherche les événements dont le type est « FACTURATION », sans plus de précision, l'index ne peut évidemment pas être utilisé :

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' ;

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on evenements  (cost=0.00..183.12 rows=50 width=69)
   Filter: (type = 'FACTURATION'::text)
```

En revanche, si la condition sur l'état de l'événement est précisée, l'index sera utilisé :

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' AND NOT traite ;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on evenements  (cost=8.22..54.62 rows=25 width=69)
   Recheck Cond: ((type = 'FACTURATION'::text) AND (NOT traite))
   ->  Bitmap Index Scan on index_partiel  (cost=0.00..8.21 rows=25 width=0)
         Index Cond: (type = 'FACTURATION'::text)
```

Sur ce jeu de données, on peut comparer la taille de deux index, partiels ou non :

```
CREATE INDEX index_complet ON evenements (type);

SELECT idxname, pg_size_pretty(pg_total_relation_size(idxname::text))
FROM (VALUES ('index_complet'), ('index_partiel')) as a(idxname);

    idxname    | pg_size_pretty
---------------+----------------
 index_complet | 88 kB
 index_partiel | 16 kB
```

Un index composé sur `(is_traite,type)` serait efficace, mais inutilement gros.

Clauses de requête et clause d'index :

Attention ! Les clauses de l'index et du `WHERE` doivent être **logiquement équivalentes** ! (et de préférence identiques)

Par exemple, dans les requêtes précédentes, un critère `traite IS FALSE` à la place de `NOT traite` n'utilise pas l'index (en effet, il ne s'agit pas du même critère à cause de `NULL` : `NULL = false` renvoie `NULL`, mais `NULL IS false` renvoie `false`).

Par contre, des conditions mathématiquement plus restreintes que l'index permettent son utilisation :

```
CREATE INDEX commandes_recentes_idx
ON commandes (client_id) WHERE date_commande > '2015-01-01' ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE date_commande > '2016-01-01' AND client_id = 17 ;

                      QUERY PLAN
------------------------------------------------------
 Index Scan using commandes_recentes_idx on commandes
   Index Cond: (client_id = 17)
   Filter: (date_commande > '2016-01-01'::date)
```

Mais cet index partiel ne sera pas utilisé pour un critère précédant 2015.

De la même manière, si un index partiel contient une liste de valeurs, `IN ()` ou `NOT IN ()`, il est en principe utilisable :

```
CREATE INDEX commandes_1_3 ON commandes (numero_commande)
WHERE  mode_expedition IN (1,3);

EXPLAIN (COSTS OFF) SELECT * FROM commandes WHERE mode_expedition = 1 ;

                 QUERY PLAN
--------------------------------------------
 Index Scan using commandes_1_3 on commandes
   Filter: (mode_expedition = 1)

DROP INDEX commandes_1_3 ;
```

```
CREATE INDEX commandes_not34 ON commandes (numero_commande)
WHERE  mode_expedition NOT IN (3,4);

EXPLAIN (COSTS OFF) SELECT * FROM commandes WHERE mode_expedition = 1 ;

                  QUERY PLAN
-----------------------------------------------
 Index Scan using commandes_not34 on commandes
   Filter: (mode_expedition = 1)

DROP INDEX commandes_not34 ;
```

3.5.2 Index partiels : cas d'usage

- Données *chaudes* et *froides*
- Index dédié à une requête avec une condition fixe

Le cas typique d'utilisation d'un index partiel est celui de l'exemple précédent : une application avec des données *chaudes*, fréquemment accédées et traitées, et des données *froides*, qui sont plus destinées à de l'historisation ou de l'archivage. Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique.

De manière générale, tout système est susceptible de bénéficier des index partiels s'il doit gérer des données à état dont seul un sous-ensemble de ces états est activement exploité par les requêtes à optimiser. Par exemple, toujours sur cette même table, des requêtes visant à faire des statistiques sur les expéditions pourraient tirer parti de cet index :

```
CREATE INDEX index_partiel_expes ON evenements (id) WHERE type = 'EXPEDITION' ;

EXPLAIN SELECT count(id) FROM evenements WHERE type = 'EXPEDITION' ;

                              QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=106.68..106.69 rows=1 width=8)
   ->  Index Only Scan using index_partiel_expes on evenements  (cost=0.28..98.34
↪  rows=3337 width=4)
```

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. Donc, si une requête présente fréquemment des critères de ce type :

```
WHERE une_colonne = un_parametre_variable
  AND une_autre_colonne = une_valeur_fixe
```

alors il peut être intéressant de créer un index partiel pour les lignes satisfaisant le critère :

```
WHERE une_autre_colonne = une_valeur_fixe
```

Ces critères sont généralement très liés au fonctionnel de l'application : du point de vue de l'exploitation, il est souvent difficile d'identifier des requêtes dont une valeur est toujours fixe. Encore une fois, l'appropriation des techniques d'indexation par l'équipe de développement permet d'améliorer grandement les performances de l'application.

3.5.3 Index partiels : utilisation

- Éviter les index de type :

```
CREATE INDEX ON matable ( champ_filtre ) WHERE champ_filtre = …
```

- Préférer :

```
CREATE INDEX ON matable ( champ_resultat ) WHERE champ_filtre = …
```

En général, un index partiel doit indexer une colonne différente de celle qui est filtrée (et donc connue). Ainsi, dans l'exemple précédent, la colonne indexée (`type`) n'est pas celle de la clause `WHERE`. On pose un critère, mais on s'intéresse aux types d'événements ramenés. Un autre index partiel pourrait porter sur `id WHERE NOT traite` pour simplement récupérer une liste des identifiants non traités de tous types.

L'intérêt est d'obtenir un index très ciblé et compact, et aussi d'économiser la place disque et la charge CPU de maintenance. Il faut tout de même que les index partiels soient notablement plus petits que les index « génériques » (au moins de moitié). Avec des index partiels spécialisés, il est possible de « précalculer » certaines requêtes critiques en intégrant leurs critères de recherche exacts.

3.5.4 Index fonctionnels : principe

- Un index sur `a` est inutilisable pour :

```
SELECT … WHERE upper(a)='DUPOND'
```

- Indexer le résultat de la fonction :

```
CREATE INDEX mon_idx ON ma_table (upper(a)) ;
```

À partir du moment où une clause `WHERE` applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « Quels sont les mots dont la traduction en français est 'fenêtre' ? ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

L'exemple classique est l'indexation insensible à la casse : on crée un index sur `UPPER` (ou `LOWER`) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

3.5.5 Index fonctionnels : conditions

- Critère identique à la fonction dans l'index
- Fonction impérativement `IMMUTABLE` !
  - délicat avec les conversions de dates/heures
- Ne pas espérer d'*Index Only Scan*

Il est facile de créer involontairement des critères comportant des fonctions, notamment avec des conversions de type ou des manipulations de dates. Il a été vu plus haut qu'il vaut mieux placer la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction `extract` est appliquée à la colonne `date_commande` (type `date`) et l'index est inutilisable.

L'optimiseur ne peut donc pas utiliser un index :

```
CREATE INDEX ON commandes (date_commande) ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                            QUERY PLAN
------------------------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on commandes
         Filter: (EXTRACT(year FROM date_commande) = '2011'::numeric)
```

En réécrivant le prédicat, l'index est bien utilisé :

```
EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE date_commande BETWEEN '01-01-2011'::date AND '31-12-2011'::date;

                            QUERY PLAN
------------------------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
   Index Cond: ((date_commande >= '2011-01-01'::date) AND (date_commande <=
↪  '2011-12-31'::date))
```

C'est la solution la plus propre.

Mais dans d'autres cas, une telle réécriture de la requête sera impossible ou très délicate. On peut alors créer un index fonctionnel, dont la définition doit être **strictement** celle du `WHERE` :

```
CREATE INDEX annee_commandes_idx ON commandes( extract('year' from date_commande) ) ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on commandes
   Recheck Cond: (EXTRACT(year FROM date_commande) = '2011'::numeric)
   ->  Bitmap Index Scan on annee_commandes_idx
         Index Cond: (EXTRACT(year FROM date_commande) = '2011'::numeric)
```

Ceci fonctionne si `date_commande` est de type `date` ou `timestamp without timezone`.

Fonction immutable :

Cependant, n'importe quelle fonction d'indexation n'est pas utilisable, ou pas pour tous les types. La fonction d'indexation doit être notée `IMMUTABLE` : cette propriété indique à PostgreSQL que la fonction retournera toujours le **même résultat** quand elle est appelée avec les **mêmes arguments**.

En d'autres termes : le résultat de la fonction ne doit dépendre :

- ni du contenu de la base (pas de `SELECT` donc) ;
- ni de la configuration, ni de l'environnement (variables d'environnement, paramètres de session, fuseau horaire, formatage…) ;
- ni du temps (`now()` ou `clock_timestamp()` sont interdits, et indirectement les calculs d'âge) ;
- ni d'une autre fonction non-déterministe (comme `random()`) ou plus généralement non immutable.

Sans ces restrictions, l'endroit dans lequel la donnée est insérée dans l'index serait potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Pour revenir à l'exemple précédent : pour calculer l'année, on peut aussi imaginer un index avec la fonction `to_char`, une autre fonction hélas fréquemment utilisée pour les conversions de date. Au moment de la création d'un tel index, PostgreSQL renvoie l'erreur suivante :

```
CREATE INDEX annee_commandes_idx2
ON commandes ((to_char(date_commande,'YYYY')::int));

ERROR:  functions in index expression must be marked IMMUTABLE
```

En effet, `to_char()` n'est pas immutable, juste « stable » et cela dans toutes ses variantes :

```
magasin=# \df+ to_char
                                        Liste des fonctions
…  Nom   |…résultat|  Type de données des paramètres  |…|Volatibilité|…
+--------+---------+-----------------------------------+-+------------+-
…to_char | text    | bigint, text                      | | stable     |…
…to_char | text    | double precision, text            | | stable     |…
```

```
…to_char | text    | integer, text                    | | stable     |…
…to_char | text    | interval, text                   | | stable     |…
…to_char | text    | numeric, text                    | | stable     |…
…to_char | text    | real, text                       | | stable     |…
…to_char | text    | timestamp without time zone, text| | stable     |…
…to_char | text    | timestamp with time zone, text   | | stable     |…
(8 lignes)
```

La raison est que `to_date` accepte des paramètres de formatage qui dépendent de la session (nom du mois, virgule ou point décimal…). Ce n'est pas une très bonne fonction pour convertir une date ou heure en nombre.

La fonction `extract`, elle, est bien immutable quand il s'agit de convertir `commande.date_commande` de `date` vers une année, comme dans l'exemple plus haut.

```
 \sf extract (text, date)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, date)
 RETURNS numeric
 LANGUAGE internal
 IMMUTABLE PARALLEL SAFE STRICT
AS $function$extract_date$function$
```

De même, `extract` est immutable avec une entrée de type `timestamp without time zone`.

Les choses se compliquent si l'on manipule des heures avec fuseau horaire. En effet, il est conseillé de toujours privilégier la variante `timestamp with time zone`. Cette fois, l'index fonctionnel basé avec `extract` va poser problème :

```
DROP INDEX annee_commandes_idx ;
-- Nouvelle table d'exemple avec date_commande comme timestamp with time zone
-- La conversion introduit implicitement le fuseau horaire de la session
CREATE TABLE commandes2 (LIKE commandes INCLUDING ALL);
ALTER TABLE  commandes2 ALTER COLUMN date_commande TYPE timestamp with time zone ;
INSERT INTO  commandes2 SELECT * FROM commandes ;
-- Reprise de l'index fonctionnel précédent
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from date_commande) ) ;

ERROR:  functions in index expression must be marked IMMUTABLE
```

En effet la fonction `extract` n'est pas immutable pour le type `timestamp with time zone` :

```
magasin=# \sf extract (text, timestamp with time zone)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, timestamp with time zone)
 RETURNS numeric
 LANGUAGE internal
 STABLE PARALLEL SAFE STRICT
AS $function$extract_timestamptz$function$
```

Pour certains *timestamps* autour du Nouvel An, l'année retournée dépend du fuseau horaire. Le problème se poserait bien sûr aussi si l'on extrayait les jours ou les mois.

Il est possible de « tricher » en figeant le fuseau horaire dans une fonction pour obtenir un type intermédiaire `timestamp without time zone`, qui ne posera pas de problème :

```
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from (
  date_commande AT TIME ZONE 'Europe/Paris' )::timestamp
  )) ;
```

Ce contournement impose de modifier le critère de la requête. Tant qu'on y est, il peut être plus clair d'enrober l'appel dans une fonction que l'on définira immutable.

```
CREATE OR REPLACE FUNCTION annee_paris (t timestamptz)
RETURNS int
AS $$
    SELECT extract ('year' FROM (t AT TIME ZONE 'Europe/Paris')::timestamp) ;
$$ LANGUAGE sql
IMMUTABLE ;

CREATE INDEX annee_commandes2_paris_idx ON commandes2 (annee_paris (date_commande));
VACUUM ANALYZE commandes2 ;

EXPLAIN (COSTS OFF)
SELECT * FROM commandes2
WHERE annee_paris (date_commande) = 2021 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx on commandes2
   Index Cond: ((EXTRACT(year FROM (date_commande AT TIME ZONE
↪  'Europe/Paris'::text)))::integer = 2021)
```

Le nom de la fonction est aussi une indication pour les utilisateurs dans d'autres fuseaux.

Certes, on a ici modifié le code de la requête, mais il est parfois possible de contourner ce problème en passant par des vues qui masquent la fonction.

Signalons enfin la fonction `date_part` : c'est une alternative possible à `extract`, avec les mêmes soucis et contournement.

À partir de PostgreSQL 16, une autre possibilité existe avec `date_trunc` car la variante avec `timestamp without time zone` est devenue immutable :

```
CREATE INDEX annee_commandes2_paris_idx3
ON commandes2 ( (date_trunc ( 'year', date_commande, 'Europe/Paris')) );
ANALYZE commandes2 ;

EXPLAIN (COSTS OFF)
SELECT * FROM commandes2
WHERE date_trunc('year', date_commande, 'Europe/Paris') = '2021-01-01'::timestamptz;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx3 on commandes2
   Index Cond: (date_trunc('year'::text, date_commande, 'Europe/Paris'::text) =
↪  '2021-01-01 00:00:00+01'::timestamp with time zone)
```

Index Only Scan :

Obtenir un *Index Only Scan* est une optimisation importante pour les requêtes critiques avec peu de champs sur la table. Hélas, en raison d'une limitation du planificateur, les index fonctionnels ne donnent pas lieu à un *Index Only Scan* :

```
EXPLAIN (COSTS OFF)
SELECT annee_paris (date_commande) FROM commandes2
WHERE annee_paris (date_commande) > 2021 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx on commandes2
   Index Cond: ((EXTRACT(year FROM (date_commande AT TIME ZONE
↪  'Europe/Paris'::text)))::integer > 2021)
```

Plus insidieusement, le planificateur peut choisir un *Index Only Scan*… sur la colonne sur laquelle porte la fonction !

```
EXPLAIN SELECT count( annee_paris(date_commande) ) FROM commandes2 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
  Aggregate  (cost=28520.40..28520.41 rows=1 width=8)
   ->  Index Only Scan using commandes2_date_commande_idx on commandes2
↪  (cost=0.42..18520.41 rows=999999 width=8)
```

Ce qui entraîne au moins un gaspillage de CPU pour réexécuter les fonctions sur chaque ligne.

Sacrifier un peu d'espace disque pour une colonne générée et son index (non fonctionnel) peut s'avérer une solution :

```
-- Attention, cette commande réécrit la table
ALTER TABLE commandes2 ADD COLUMN annee_paris smallint
  GENERATED ALWAYS  AS ( annee_paris (date_commande) ) STORED ;
CREATE INDEX commandes2_annee_paris_idx ON commandes2 (annee_paris) ;
-- Prise en compte des statistiques et des lignes mortes sur la table réécrite
VACUUM ANALYZE commandes2;

EXPLAIN SELECT count( annee_paris ) FROM commandes2 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=14609.10..14609.11 rows=1 width=8)
   ->  Gather  (cost=14608.88..14609.09 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=13608.88..13608.89 rows=1 width=8)
               ->  Parallel Index Only Scan using commandes2_annee_paris_idx on
↪  commandes2  (cost=0.42..12567.20 rows=416672 width=2)
```

3.5.6 Index fonctionnels : maintenance

- Ne pas oublier `ANALYZE` après création d'un index fonctionnel
  - les statistiques peuvent même être l'intérêt majeur (<v14)
- La fonction ne doit jamais tomber en erreur
- Si modification de la fonction
  - réindexation

Statistiques :

Après la création de l'index fonctionnel, un `ANALYZE nom_table` est conseillé : en effet, l'optimiseur ne peut utiliser les statistiques déjà connues pour le résultat d'une fonction. Par contre, PostgreSQL peut créer des statistiques sur le résultat de la fonction pour chaque ligne. Ces statistiques seront visibles dans la vue système `pg_stats` (`tablename` contient le nom de l'index, et non celui de la table !).

Ces statistiques à jour sont d'ailleurs un des intêrêts de l'index fonctionnel, même si l'index lui-même est superflu. Dans ce cas, à partir de PostgreSQL 14, on pourra utiliser `CREATE STATISTICS` sur l'expression pour ne pas avoir à créer et maintenir un index entier.

Avertissements :

La fonction ne doit jamais tomber en erreur ! Il ne faut pas tester l'index qu'avec les données en place, mais aussi avec toutes celles susceptibles de se trouver dans le champ concerné. Sinon, il y aura des refus d'insertion ou de mise à jour. Des `ANALYZE` ou `VACUUM` pourraient aussi échouer, avec de gros problèmes sur le long terme.

Si le contenu de la fonction est modifié avec `CREATE OR REPLACE FUNCTION`, il faudra impérativement réindexer, car PostgreSQL ne le fera pas automatiquement. Sans cela, les résultats des requêtes différeront selon qu'elles utiliseront ou non l'index !

3.5.7 Index couvrants : principe

- But : obtenir un *Index Only Scan*

```
CREATE UNIQUE INDEX clients_idx1
ON clients (id_client) INCLUDE (nom_client) ;
```

- Répondent à la clause `WHERE`
- **ET** contiennent toutes les colonnes demandées par la requête :

```
SELECT id_client,nom_client FROM clients WHERE id_client > 100 ;
```

- …si l'index n'est pas trop gros
  - à comparer à un index multicolonne

3.5.7.1 Principe des index couvrants

Un index couvrant (*covering index*) cherche à favoriser le nœud d'accès le plus rapide, l'*Index Only Scan* : il contient non seulement les champs servant de critères de recherche, mais aussi tous les champs résultats. Ainsi, il n'y a plus besoin d'interroger la table.

Les index couvrants peuvent être explicitement déclarés avec la clause `INCLUDE` :

```
CREATE TABLE t (id int NOT NULL, valeur int) ;

INSERT INTO t SELECT i, i*50 FROM generate_series(1,1000000) i;

CREATE UNIQUE INDEX t_pk ON t (id) INCLUDE (valeur) ;

VACUUM t ;

EXPLAIN ANALYZE SELECT valeur FROM t WHERE id = 555555 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan using t_pk on t  (cost=0.42..1.44 rows=1 width=4)
                                  (actual time=0.034..0.035 rows=1 loops=1)
   Index Cond: (id = 555555)
   Heap Fetches: 0
 Planning Time: 0.084 ms
 Execution Time: 0.065 ms
```

Dans cet exemple, il n'y a pas eu d'accès à la table. L'index est unique mais contient aussi la colonne `valeur`.

Noter le `VACUUM`, nécessaire pour garantir que la *visibility map* de la table est à jour et permet ainsi un *Index Only Scan* sans aucun accès à la table (clause *Heap Fetches* à 0).

Par abus de langage, on peut dire d'un index multicolonne sans clause `INCLUDE` qu'il est « couvrant » s'il répond complètement à la requête.

Dans les versions antérieures à la 11, on émulait cette fonctionnalité en incluant les colonnes dans des index multicolonne :

```
CREATE INDEX t_idx ON t (id, valeur) ;
```

Cette technique reste tout à fait valable dans les versions suivantes, car l'index multicolonne (complètement trié) peut servir de manière optimale à d'autres requêtes. Il peut même être plus petit que celui utilisant `INCLUDE`.

Un intérêt de la clause `INCLUDE` est de se greffer sur des index uniques ou de clés et d'économiser un nouvel index et un peu de place. Accessoirement, il évite le tri des champs dans la clause `INCLUDE`.

3.5.7.2 Inconvénients & limitation des index couvrants

Il faut garder à l'esprit que l'ajout de colonnes à un index (couvrant ou multicolonne) augmente sa taille. Cela peut avoir un impact sur les performances des requêtes qui n'utilisent pas les colonnes supplémentaires. Il faut également être vigilant à ce que la taille des enregistrements avec les colonnes incluses ne dépassent pas 2,6 ko. Au-delà de cette valeur, les insertions ou mises à jour échouent.

Enfin, la déduplication (apparue en version 13) n'est pas active sur les index couvrants, ce qui a un impact supplémentaire sur la taille de l'index sur le disque et en cache. Ça n'a pas trop d'importance si l'index principal contient surtout des valeurs différentes, mais s'il y en a beaucoup moins que de lignes, il serait dommage de perdre l'intérêt de la déduplication. Là encore, le planificateur peut ignorer l'index s'il est trop gros. Il faut tester avec les données réelles, et comparer avec un index multicolonne (dédupliqué).

Les méthodes d'accès aux index doivent inclure le support de cette fonctionnalité. C'est le cas pour le B-tree ou le GiST, et pour le SP-GiST en version 14.

3.5.8 Classes d'opérateurs

- Un index utilise des opérateurs de comparaison
- Texte : différentes collations = différents tris… complexes
  - Index inutilisable sur :

```
WHERE col_varchar LIKE 'chaine%'
```

- Solution : opérateur `varchar_pattern_ops` :
  - force le tri caractère par caractère, sans la collation

```
CREATE INDEX idx1
ON ma_table (col_varchar varchar_pattern_ops)
```

- Plus généralement :
  - nombreux autres opérateurs pour d'autres types d'index

Un opérateur sert à indiquer à PostgreSQL comment il doit manipuler un certain type de données. Il y a beaucoup d'opérateurs par défaut, mais il est parfois possible d'en prendre un autre.

Pour l'indexation, il est notamment possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison.

Le cas d'utilisation le plus fréquent dans PostgreSQL est la comparaison de chaîne `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (en France, généralement `fr_FR.UTF-8` ou `en_US.UTF-8`) ou la collation de la colonne de la table si elle diffère. Cette collation contient des notions de tri. Les règles sont différentes pour chaque collation. Et ces règles sont complexes.

Par exemple, le **ß** allemand se place entre **ss** et **t** (et ce, même en français). En danois, le tri est très particulier car le **å** et le **aa** apparaissent après le **z**.

```
-- Cette collation doit exister sur le système
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls ORDER BY x COLLATE "da_DK";
```

```
 x
----
 s
 ss
 ß
 t
 zz
 å
 aa
```

Il faut être conscient que cela a une influence sur le résultat d'un filtrage :

```
WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls
WHERE x > 'z' COLLATE "da_DK" ;

 x
----
 aa
 å
 zz
```

Il serait donc très complexe de réécrire le `LIKE` en un `BETWEEN`, comme le font habituellement tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte >= 'toto' and  coltexte < 'totp'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur Unicode par exemple. Même si l'index est dans la bonne collation, il n'est pas facilement utilisable :

```
CREATE INDEX ON textes (livre) ;
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..525328.76 rows=75173 width=123)
   Workers Planned: 2
   ->  Parallel Seq Scan on textes  (cost=0.00..516811.46 rows=31322 width=123)
         Filter: (livre ~~ 'Les misérables%'::text)
```

La classe d'opérateurs `varchar_pattern_ops` sert à changer ce comportement :

```
CREATE INDEX ON ma_table (col_varchar varchar_pattern_ops) ;
```

Ce nouvel index est alors construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient. Il devient alors trivial pour l'optimiseur de faire la réécriture :

```
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                               QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using textes_livre_idx1 on textes  (cost=0.69..70406.87 rows=75173
↪  width=123)
   Index Cond: ((livre ~>=~ 'Les misérables'::text) AND (livre ~<~ 'Les
↪  misérablet'::text))
   Filter: (livre ~~ 'Les misérables%'::text)
```

Cela convient pour un `LIKE 'critère%'`, car le début est fixe, et l'ordre de tri n'influe pas sur le résultat. (Par contre cela ne permet toujours pas d'indexer `LIKE %critère%`.) Noter la clause `Filter` qui filtre en deuxième intention ce qui a pu être trouvé dans l'index.

Il existe quelques autres cas d'utilisation d'`opclass` alternatives, notamment pour utiliser d'autres types d'index que B-tree. Deux exemples :

- indexation d'un JSON (type `jsonb`) par un index GIN :

```
CREATE INDEX ON stock_jsonb USING gin (document_jsonb jsonb_path_ops);
```

- indexation de trigrammes de textes avec le module `pg_trgm` et des index GiST :

```
CREATE INDEX ON livres USING gist (text_data gist_trgm_ops);
```

Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs[8].

Ne mettez pas systématiquement `varchar_pattern_ops` dans tous les index de chaînes de caractère. Cet opérateur est adapté au `LIKE 'critère%` mais ne servira pas pour un tri sur la chaîne (`ORDER BY`). Selon les requêtes et volumétries, les deux index peuvent être nécessaires.

3.5.9 Conclusion

- Responsabilité de l'indexation
- Compréhension des mécanismes
- Différents types d'index, différentes stratégies

L'indexation d'une base de données est souvent un sujet qui est traité trop tard dans le cycle de l'application. Lorsque celle-ci est gérée à l'étape du développement, il est possible de bénéficier de l'expérience et de la connaissance des développeurs. La maîtrise de cette compétence est donc idéalement transverse entre le développement et l'exploitation.

Le fonctionnement d'un index B-tree est somme toute assez simple, mais il est important de bien l'appréhender pour comprendre les enjeux d'une bonne stratégie d'indexation.

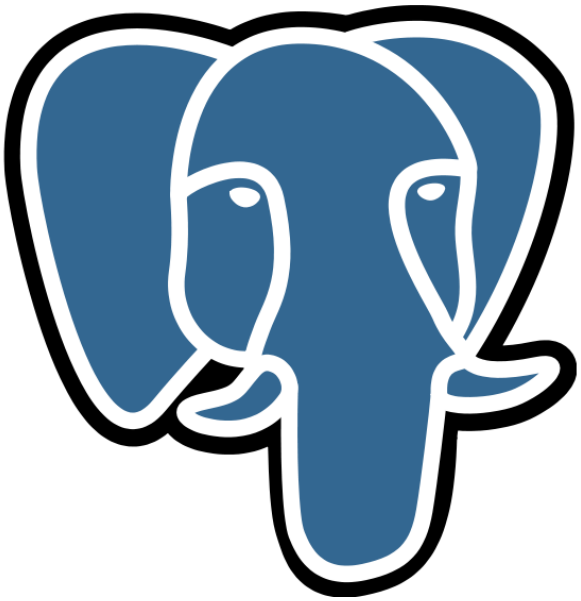
PostgreSQL fournit aussi d'autres types d'index moins utilisés, mais très précieux dans certaines situations : BRIN, GIN, GiST, etc.

[8] https://www.postgresql.org/docs/current/static/indexes-opclass.html

3.6 QUIZ

https://dali.bo/j4_quiz

4/ Comprendre EXPLAIN



4.1 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. 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.

Ce qui suit se concentrera sur les plans d'exécution.

4.1.1 Au menu

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

Nous ferons quelques rappels et approfondissements sur la façon dont une requête s'exécute globalement, et sur le planificateur : en quoi est-il utile, comment fonctionne-t-il, et comment le configurer.

Nous ferons un tour sur le fonctionnement de la commande `EXPLAIN` et les informations qu'elle fournit. Nous verrons aussi plus en détail l'ensemble des opérations utilisables par le planificateur, et comment celui-ci choisit un plan.

4.2 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.

Une lenteur dans une requête peut se trouver dans l'un ou l'autre de ces niveaux.

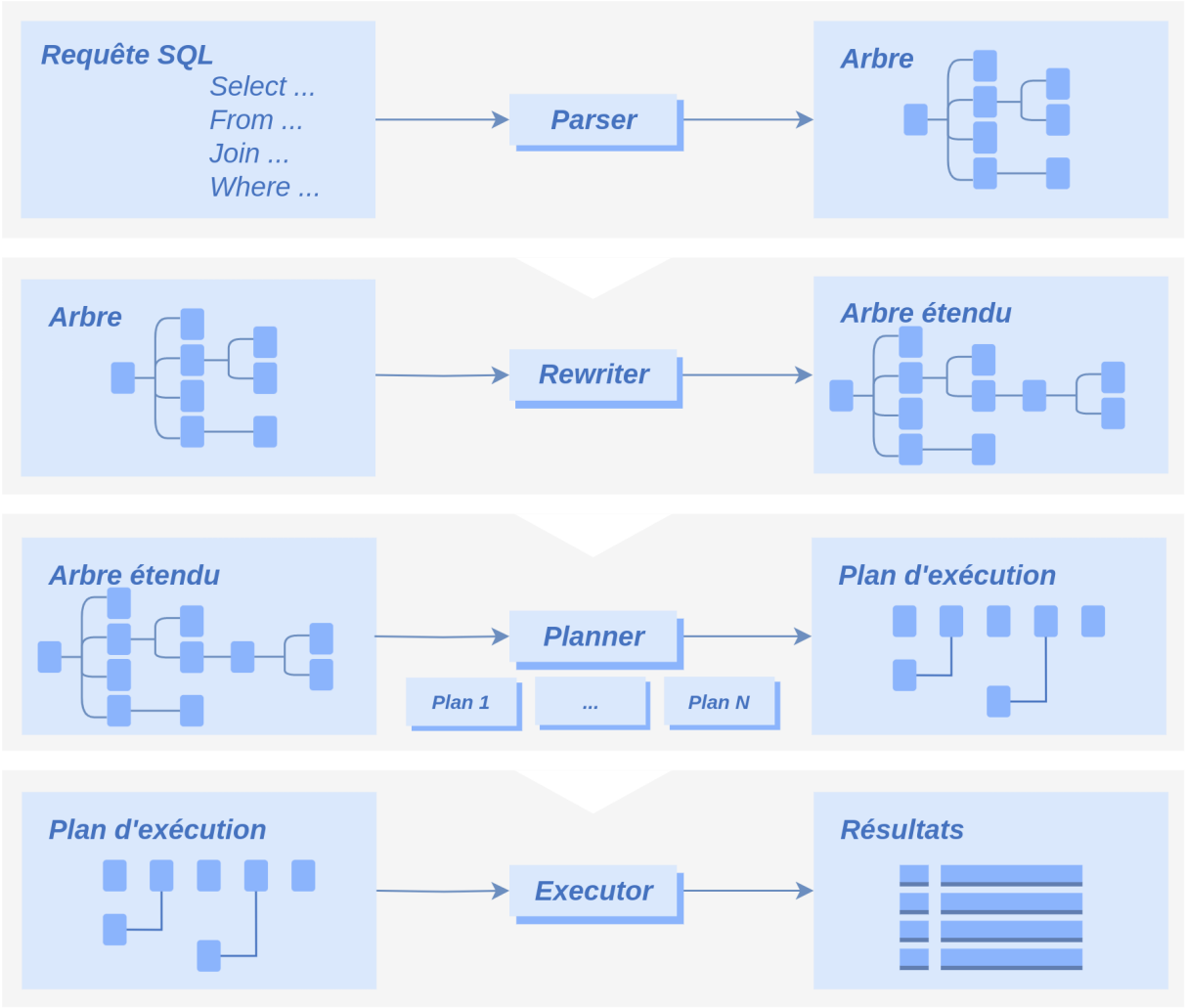
4.2.1 Niveau système

- 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. L'affichage peut aussi être un problème (afficher une ligne sera plus rapide qu'afficher un million de lignes, afficher un entier est plus rapide qu'afficher un document texte de 1 Mo, etc.).

4.2.2 Traitement d'une requête



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

Tout d'abord, le *parser* va réaliser une analyse syntaxique de la requête.

Puis le *rewriter* va réécrire, si nécessaire, la requête. Pour cela, il prend en compte les règles, les vues non matérialisées et les fonctions SQL.

Si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle.

Si une vue non matérialisée est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée. Il en est de même pour une fonction SQL intégrable.

Ensuite, 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.

Enfin, 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 :

```
SET log_parser_stats TO on;
SET log_planner_stats TO on;
SET log_executor_stats TO on;
SET client_min_messages TO log;
```

- Exécution de la requête :

```
SELECT fonction, COUNT(*) FROM employes_big GROUP BY fonction ORDER BY fonction;
```

- Trace du *parser* :

```
LOG:  PARSER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000026 s user, 0.000017 s system, 0.000042 s elapsed
!       [0.013275 s user, 0.008850 s system total]
!       17152 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/3 [0/575] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DÉTAIL : ! system usage stats:
!       0.000396 s user, 0.000263 s system, 0.000660 s elapsed
!       [0.013714 s user, 0.009142 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/32 [0/607] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *rewriter* :

```
LOG:  REWRITER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000010 s user, 0.000007 s system, 0.000016 s elapsed
!       [0.013747 s user, 0.009165 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/1 [0/608] page faults/reclaims, 0 [0] swaps
```

```
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *planner* :

```
DÉTAIL : ! system usage stats:
!       0.000255 s user, 0.000170 s system, 0.000426 s elapsed
!       [0.014021 s user, 0.009347 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/25 [0/633] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace de l'*executer* :

```
LOG:  EXECUTOR STATISTICS
DÉTAIL : ! system usage stats:
!       0.044788 s user, 0.004177 s system, 0.131354 s elapsed
!       [0.058917 s user, 0.013596 s system total]
!       46268 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/468 [0/1124] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       4/16 [9/16] voluntary/involuntary context switches
```

- Résultat de la requête :

```
     fonction      | count
-------------------+--------
 Commercial        |      2
 Comptable         |      1
 Consultant        | 499005
 Développeur       |      2
 Directeur Général |      1
 Responsable       |      4
```

4.2.3 Exceptions

- Procédures stockées (appelées avec `CALL`)
- 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.

4.3 QUELQUES DÉFINITIONS

- Prédicat
  - filtre de la clause `WHERE`
  - conditions de jointure
- Sélectivité
  - % de lignes retournées après application d'un prédicat
- Cardinalité
  - nombre de lignes d'une table
  - nombre de lignes retournées après filtrages

Un prédicat est une condition de filtrage présente dans la clause `WHERE` d'une requête. Par exemple `colonne = valeur`. On parle aussi de prédicats de jointure pour les conditions de jointures présentes dans la clause `WHERE` ou suivant la clause `ON` d'une jointure.

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 10 % des lignes, soit 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 de la sortie d'un nœud. Elle représente aussi le nombre de lignes retournées par la lecture d'une table après application d'un ou plusieurs prédicats.

4.3.1 Jeu de tests

- Tables
  - `services` : 4 lignes
  - `services_big` : 40 000 lignes
  - `employes` : 14 lignes
  - `employes_big` : ~500 000 lignes
- Index
  - `service*`.`num_service` (clés primaires)
  - `employes*`.`matricule` (clés primaires)
  - `employes*`.`date_embauche`
  - `employes_big`.`num_service` (clé étrangère)

Les deux volumétries différentes vont permettre de mettre en évidence certains effets.

4.3.2 Jeu de tests (schéma)

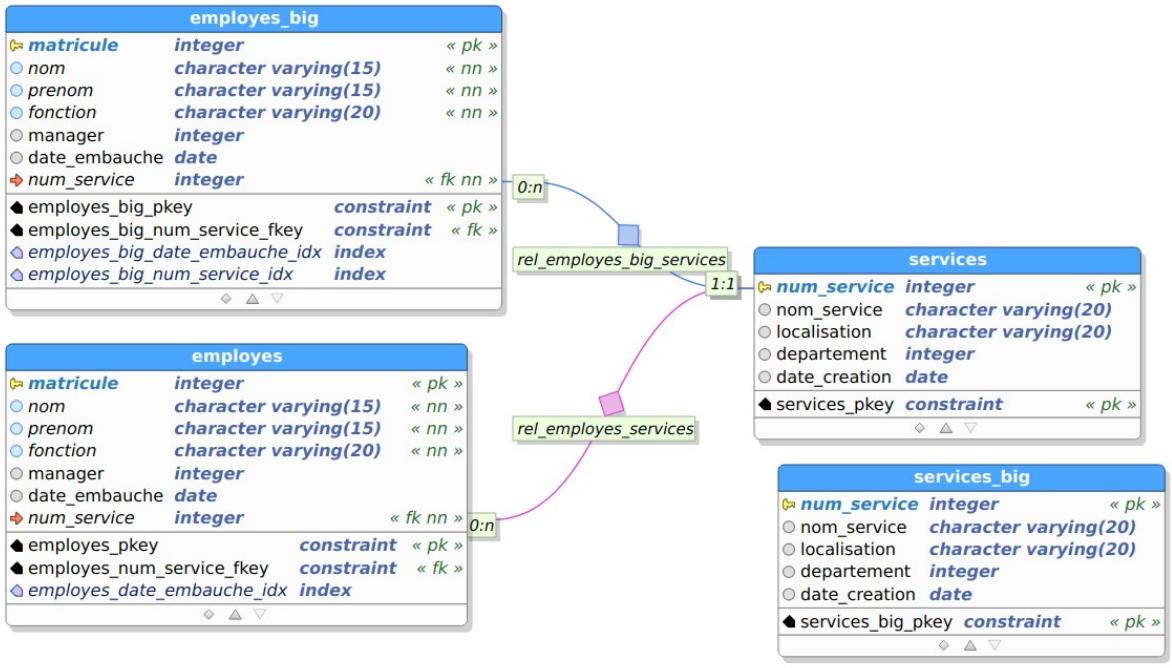


Figure 4/ .1: Tables employés & services

Les tables suivantes nous serviront d'exemple par la suite. Le script de création se télécharge et s'installe ainsi dans une nouvelle base **employes** :

```
curl -kL https://dali.bo/tp_employes_services -o employes_services.sql
createdb employes
psql employes < employes_services.sql
```

Les quelques tables occupent environ 80 Mo sur le disque.

```
-- suppression des tables si elles existent

DROP TABLE IF EXISTS services CASCADE;
DROP TABLE IF EXISTS services_big CASCADE;
DROP TABLE IF EXISTS employes CASCADE;
DROP TABLE IF EXISTS employes_big CASCADE;

-- définition des tables

CREATE TABLE services (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);
```

```
CREATE TABLE services_big (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);

CREATE TABLE employes (
    matricule     serial 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)
);

CREATE TABLE employes_big (
    matricule     serial 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)
);

-- ajout des données

INSERT INTO services
VALUES
    (1, 'Comptabilité', 'Paris', 75, '2006-09-03'),
    (2, 'R&D', 'Rennes', 40, '2009-08-03'),
    (3, 'Commerciaux', 'Limoges', 52, '2006-09-03'),
    (4, 'Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement, date_creation)
VALUES
    ('Comptabilité', 'Paris', 75, '2006-09-03'),
    ('R&D', 'Rennes', 40, '2009-08-03'),
    ('Commerciaux', 'Limoges', 52, '2006-09-03'),
    ('Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement, date_creation)
  SELECT s.nom_service, s.localisation, s.departement, s.date_creation
  FROM services s, generate_series(1, 10000);

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

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

-- on copie la table employes
INSERT INTO employes_big SELECT * FROM employes;

-- duplication volontaire des lignes d'un des employés
INSERT INTO employes_big
  SELECT i, nom,prenom,fonction,manager,date_embauche,num_service
  FROM employes_big,
  LATERAL generate_series(1000, 500000) i
  WHERE matricule=137;

-- création des index
CREATE INDEX ON employes(date_embauche);
CREATE INDEX ON employes_big(date_embauche);
CREATE INDEX ON employes_big(num_service);

-- calcul des statistiques sur les nouvelles données
VACUUM ANALYZE;
```

4.3.3 Requête étudiée

```
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 nous servira d'exemple. Elle permet de déterminer les employés basés à Nantes et pour résultat :

```
 matricule |   nom    |  prenom   | nom_service |  fonction   | localisation
-----------+----------+-----------+-------------+-------------+--------------
        33 | Roy      | Arthur    | Consultants | Consultant  | Nantes
       105 | Vacuum   | Anne-Lise | Consultants | Responsable | Nantes
       119 | Thierrie | Armand    | Consultants | Consultant  | Nantes
       136 | Barnier  | Germaine  | Consultants | Consultant  | Nantes
       137 | Pivert   | Victor    | Consultants | Consultant  | Nantes
```

En fonction du cache, elle dure de 1 à quelques millisecondes.

4.3.4 Plan de la requête étudiée

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

```
Hash Join  (cost=1.06..2.28 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.

Ce plan montre une jointure par hachage. La table `services` est parcourue intégralement (*Seq Scan*), mais elle est filtrée sur le critère sur « Nantes ».

Un *hash* de la colonne `num_service` des lignes résultantes de ce filtrage est effectué, et comparé aux valeurs rencontrées lors d'un parcours complet de `employes`.

S'affichent également les coûts estimés des opérations et le nombre de lignes que PostgreSQL s'attend à trouver à chaque étape.

4.4 PLANIFICATEUR

Rappels :

- SQL est un langage déclaratif
- Planificateur : trouver le meilleur plan
- Énumère tous les plans d'exécution possible
  - tous ou presque…
- Statistiques + configuration + règles → coût
- Coût le plus bas = meilleur plan

Le but du planificateur est assez simple. Pour une requête, il existe de nombreux plans d'exécution possibles. Il va donc tenter d'énumérer tous les plans d'exécution possibles ; même si leur nombre devient vite colossal dans une requête complexe : chaque table peut être accédée selon différents plans, selon l'un ou l'autre critère ou une combinaison, les algorithmes de jointure possibles sont multiples, etc.

Lors de cette énumération des différents plans, il calcule leur coût. Cela lui permet d'en ignorer certains alors qu'ils sont incomplets si leur plan d'exécution est déjà plus coûteux que les autres. Pour calculer le coût, 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.

À la fin de l'énumération et du calcul de coût, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût, à priori celui qui sera le plus rapide pour la requête demandée. (En toute rigueur, pour réduire le nombre de plans très voisins à étudier, des plans de coûts différents de 1% près peuvent être considérés équivalents[1], et le coût de démarrage peut alors les départager.)

Le coût d'un plan est une valeur calculée sans unité ni signification physique.

[1] https://doxygen.postgresql.org/pathnode_8c_source.html#l00151

4.4.1 Règles

- Règle 1 : récupérer le bon résultat
- Règle 2 : 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 : un résultat rapide mais faux n'a aucun intérêt ;
- 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 (charge des CPU utilisés et nombre de CPU utilisés) et l'utilisation de la mémoire.

Dans le cas des opérations disques, s'il doit en faire, il doit souvent privilégier les opérations séquentielles aux dépens des opérations aléatoires (qui demandent un déplacement de la tête de disque, opération la plus coûteuse sur les disques magnétiques).

4.4.2 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 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.

4.4.3 Optimisations

- À partir du modèle de données
  - suppression de jointures externes inutiles
- Transformation des sous-requêtes
  - certaines sous-requêtes transformées en jointures
  - ex : `critere IN (SELECT ...)`
- Appliquer les prédicats le plus tôt possible
  - réduit le jeu de données manipulé
  - CTE : barrière avant la v12 !
- Intègre le code des fonctions SQL simples (*inline*)
  - évite un appel de fonction coûteux

Suppression des jointures externes inutiles

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.

Sous certaines conditions, PostgreSQL peut supprimer des jointures externes, à condition que le résultat ne soit pas modifié. Dans l'exemple suivant, il ne sert à rien d'aller consulter la table `services` (ni données à récupérer, ni filtrage à faire, et même si la table est vide, le `LEFT JOIN` ne provoquera la disparition d'aucune ligne) :

```
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=19)
   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=0.00..2.27 rows=5 width=19)
   ->  Seq Scan on services s  (cost=0.00..1.05 rows=1 width=4)
         Filter: (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=0.00..2.25 rows=2 width=64)
   ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=21)
         Filter: (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter: (num_service = 1)
```

La sous-requête `ser` a été remontée dans l'arbre de requête pour être intégrée 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))
```

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 considérer dès le départ. C'est généralement une bonne chose.

Mais en cas de problème, il est possible d'utiliser une CTE matérialisée (*Common Table Expression*, clause `WITH … AS MATERIALIZED (…)`) pour bloquer cette optimisation et forcer PostgreSQL à exécuter le contenu de la requête en premier[2]. En versions 12 et ultérieures, une CTE est par défaut non matérialisée et donc intégrée avec le reste de la requête (du moins dans les cas simples comme ci-dessus), comme une sous-requête. On retombe exactement sur le plan précédent :

```
-- v12 : CTE sans MATERIALIZED (comportement par défaut)
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=1.21..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))
```

Pour recréer la « barrière d'optimisation », il est nécessaire d'ajouter le mot-clé `MATERIALIZED` :

```
-- v12 : CTE avec MATERIALIZED
EXPLAIN
  WITH e AS MATERIALIZED ( 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)
```

La CTE est alors intégralement exécutée avec son filtre propre, avant que le deuxième filtre soit appliqué dans un autre nœud. Jusqu'en version 11 incluse, ce dernier comportement était celui par défaut, et les CTE étaient une source fréquente de problèmes de performances.

Function inlining

Voici deux fonctions, la première écrite en SQL, la seconde en PL/pgSQL :

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

[2] https://docs.postgresql.fr/current/queries-with.html#QUERIES-WITH-CTE-MATERIALIZATION

```
$BODY$
  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 :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on employes (actual time=0.354..0.354 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date, '-1'::integer))
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.199 ms
 Execution Time: 0.509 ms
```

Effectivement, 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 est directement intégrée dans la clause de filtrage de la requête :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_sql(now()::date, -1);

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on employes (actual time=0.014..0.014 rows=0 loops=1)
   Filter: (date_embauche = (((now())::date + '-1 mons'::interval))::date)
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.111 ms
 Execution Time: 0.027 ms
```

Le temps d'exécution a été divisé presque par 20 sur ce jeu de données très réduit, montrant l'impact de l'appel d'une fonction dans une clause de filtrage.

Dans les deux cas ci-dessus, PostgreSQL a négligé l'index sur `date_embauche` : la table ne faisait de toute façon qu'un bloc ! Mais pour de plus grosses tables, l'index sera nécessaire, et la différence entre fonctions PL/pgSQL et SQL devient alors encore plus flagrante. Avec la même requête sur la table `employes_big`, beaucoup plus grosse, on obtient ceci :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on employes_big (actual time=464.531..464.531 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date, '-1'::integer))
   Rows Removed by Filter: 499015
   Buffers: shared hit=4664
 Planning:
   Buffers: shared hit=61
 Planning Time: 0.176 ms
 Execution Time: 465.848 ms
```

La fonction portant sur une « boîte noire », l'optimiseur n'a comme possibilité que le parcours complet de la table.

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_sql(now()::date, -1);
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
                                    (actual time=0.016..0.016 rows=0 loops=1)
   Index Cond: (date_embauche = (((now())::date + '-1 mons'::interval))::date)
   Buffers: shared hit=3
 Planning Time: 0.143 ms
 Execution Time: 0.032 ms
```

La fonction SQL est intégrée, l'optimiseur voit le critère dans `date_embauche` et peut donc se poser la question de l'utiliser (et ici, la réponse est oui : 3 blocs contre 4664, tous présents dans le cache dans cet exemple).

D'où une exécution beaucoup plus rapide.

4.4.4 Décisions

L'optimiseur doit choisir :

- Stratégie d'accès aux lignes
  - parcours de table, d'index, fonction, etc.
- Stratégie d'utilisation des jointures
  - ordre
  - ordre des tables jointes
  - type (*Nested Loop*, *Merge/Sort Join*, *Hash Join*…)
- Stratégie d'agrégation
  - brut, trié, haché
- En version parallélisée ?
- Tenir compte de la consommation mémoire

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut :

- utiliser un parcours de table (*Seq Scan*) qui ne lit que celle-ci ;
- parcourir un index et revenir chercher les lignes dans la table (*Index Scan* ou *Bitmap Scan*) ;
- ou se contenter de l'index s'il suffit (*Index Only Scan*).

Il existe encore d'autres types de parcours. Les accès aux tables et index sont généralement les premières opérations utilisées.

Pour joindre les tables, l'ordre est très important pour essayer de réduire la masse des données manipulées. Les jointures se font toujours entre deux des tables impliquées, pas plus ; ou entre une table et le résultat d'un nœud, ou entre les résultats de deux nœuds.

Pour la jointure elle-même, il existe plusieurs méthodes différentes : boucles imbriquées (*Nested Loops*), hachage (*Hash Join*), tri-fusion (*Merge Join*)…

Il existe également plusieurs algorithmes d'agrégation des lignes. Un tri peut être nécessaire pour une jointure, une agrégation, ou pour un `ORDER BY`, et là encore il y a plusieurs algorithmes possibles. L'optimiseur peut aussi décider d'utiliser un index (déjà trié) pour éviter ce tri.

Certaines des opérations ci-dessus sont parallélisables. Certaines sont aussi susceptibles de consommer beaucoup de mémoire, l'optimiseur doit en tenir compte.

4.4.5 Parallélisation

- Processus supplémentaires pour certains nœuds
  - parer à la limitation par le CPU
- En lecture (sauf exceptions)
- Parcours séquentiel
- Jointures : *Nested Loop* / *Hash Join* / *Merge Join*
- Agrégats
- Parcours d'index (B-Tree uniquement)
- Création d'index B-Tree
- Certaines créations de table et vues matérialisées
- `DISTINCT` (v15)

Principe :

À partir d'une certaine quantité de données à traiter par un nœud, un ou plusieurs processus auxiliaires (*parallel workers*) apparaissent pour répartir la charge sur d'autres processeurs. Sans cela, une requête n'est traitée que par un seul processus sur un seul processeur.

Il ne s'agit pas de lire une table avec plusieurs processus mais de répartir le traitement des lignes. La parallélisation n'est donc utile que si le CPU est le facteur limitant. Par exemple, un simple `SELECT` sur une grosse table sans `WHERE` ne mènera pas à un parcours parallélisé.

La parallélisation concerne en premier lieu les parcours de tables (*Seq Scan*), les jointures (*Nested Loop*, *Hash Join*, *Merge Join*), ainsi que certaines fonctions d'agrégat (comme `min`, `max`, `avg`, `sum`, etc.) ; mais encore les parcours d'index B-Tree (*Index Scan*, *Index Only Scan* et *Bitmap Scan*) La parallélisation est en principe disponible pour les autres types d'index, mais ils n'en font pas usage pour l'instant.

La parallélisation ne concerne encore que les opérations en lecture. Il y a des exceptions, comme la création des index B-Tree de façon parallélisée. Certaines créations de table avec `CREATE TABLE … AS`, `SELECT … INTO` sont aussi parallélisables, ainsi que `CREATE MATERIALIZED VIEW`.

En version 15, il devient possible de paralléliser des clauses `DISTINCT`.

Paramétrage :

Le paramétrage s'est affiné au fil des versions. Les paramètres suivants sont valables à partir de la version 13.

Le paramètre `max_parallel_workers_per_gather` (2 par défaut) désigne le nombre de processus auxiliaires maximum d'un nœud d'une requête. `max_parallel_maintenance_workers` (2 par défaut)

est l'équivalent dans les opérations de maintenance (réindexation notamment). Trop de processus parallèles peuvent mener à une saturation de CPU ; l'exécuteur de PostgreSQL ne lancera donc pas plus de `max_parallel_workers` processus auxiliaires simultanés (8 par défaut), lui-même limité par `max_worker_processes` (8 par défaut). En pratique, on ajustera le nombre de *parallel workers* en fonction des CPU de la machine et de la charge attendue.

La mise en place de l'infrastructure de parallélisation a un coût, défini par `parallel_setup_cost` (1000 par défaut), et des tailles de table ou index minimales, en-dessous desquels la parallélisation n'est pas envisagée.

La plupart de ces paramètres peuvent être modifiés dans une sessions par `SET`.

4.4.6 Limites actuelles de la parallélisation

- Lourd à déclencher
- Pas sur les écritures de données
- Très peu d'opérations DDL gérées
- Pas en cas de verrous
- Pas sur les curseurs
- En évolution à chaque version

Même si cette fonctionnalité évolue au fil des versions majeures, des limitations assez fortes restent présentes[3], notamment :

- elle est assez lourde à mettre en place, elle a donc un coût d'entrée qui la rend inutile quand il y a peu de lignes ;
- pas de parallélisation pour les écritures de données (`INSERT`, `UPDATE`, `DELETE`, etc.),
- peu de parallélisation sur les opérations DDL (par exemple un `ALTER TABLE` ne peut pas être parallélisé)

Il y a des cas particuliers, notamment `CREATE TABLE AS` ou `CREATE MATERIALIZED VIEW`, parallélisable à partir de la v11 ; ou le niveau d'isolation *serializable*: avant la v12, il ne permet aucune parallélisation.

[3] https://docs.postgresql.fr/current/when-can-parallel-query-be-used.html

4.5 MÉCANISME DE COÛTS & STATISTIQUES

- 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.

4.5.1 Coûts unitaires

- Coûts à connaître :
  - accès au disque séquentiel / non séquentiel
  - traitement d'un enregistrement issu d'une table
  - traitement d'un enregistrement issu d'un index
  - application d'un opérateur
  - traitement d'un enregistrement dans un parcours parallélisé
  - mise en place d'un parcours parallélisé
  - mise en place du JIT, du parallélisme…
- Chaque coût = un paramètre
  - modifiable dynamiquement avec `SET`

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.

Divers paramètres permettent d'ajuster les coûts relatifs. Ces coûts sont arbitraires, à ne comparer qu'entre eux, et ne sont pas liés directement à des caractéristiques physiques du serveur.

- `seq_page_cost` (1 par défaut) représente le coût relatif d'un accès séquentiel à un bloc sur le disque, c'est-à-dire à un bloc lu en même temps que ses voisins dans la table ;

- `random_page_cost` (4 par défaut) représente le coût relatif d'un accès aléatoire (isolé) à un bloc : 4 signifie que le temps d'accès de déplacement de la tête de lecture de façon aléatoire est estimé 4 fois plus important que le temps d'accès en séquentiel — ce sera moins avec un bon disque, voire 1 pour un SSD ;
- `cpu_tuple_cost` (0,01 par défaut) représente le coût relatif de la manipulation d'une ligne en mémoire ;
- `cpu_index_tuple_cost` (0,005 par défaut) répercute le coût de traitement d'une donnée issue d'un index ;
- `cpu_operator_cost` (défaut 0,0025) indique le coût d'application d'un opérateur sur une donnée ;
- `parallel_tuple_cost` (0,1 par défaut) indique le coût estimé du transfert d'une ligne d'un processus à un autre ;
- `parallel_setup_cost` (1000 par défaut) indique le coût de mise en place d'un parcours parallélisé, une procédure assez lourde qui ne se rentabilise pas pour les petites requêtes ;
- `jit_above_cost` (100 000 par défaut), `jit_inline_above_cost` (500 000 par défaut), `jit_optimize_above_cost` (500 000 par défaut) représentent les seuils d'activation de divers niveaux du JIT (*Just In Time* ou compilation à la volée des requêtes), outil qui ne se rentabilise que sur les gros volumes.

En général, on ne modifie pas ces paramètres sans justification sérieuse. Le plus fréquemment, on peut être amené à diminuer `random_page_cost` si le serveur dispose de disques rapides, d'une carte RAID équipée d'un cache important ou de SSD. Mais en faisant cela, il faut veiller à ne pas déstabiliser des plans optimaux qui obtiennent des temps de réponse constants. 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.

Pour des besoins particuliers, ces paramètres sont modifiables dans une session. Ils peuvent être modifiés dynamiquement par l'application avec l'ordre `SET` pour des requêtes bien particulières, pour éviter de toucher au paramétrage général.

4.6 STATISTIQUES

- Combien de lignes va-t-on traiter ?
- Toutes les décisions du planificateur se basent sur les statistiques
  - le choix du parcours
  - comme le choix des jointures
- Mettre à jour les statistiques sur les données :
  - `ANALYZE`
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est une bonne chose, mais si on ne sait pas le nombre de lignes à traiter, on ne peut pas calculer le coût total. Le planificateur se base alors principalement sur les statistiques pour ses décisions. Avec ces informations et le paramétrage, l'optimiseur saura par exemple calculer le ratio d'un filtre et décider s'il faut passer par un index, ou calculer le ratio d'une jointure pour choisir la stratégie de jointure. 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 planificateur 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 exécutée manuellement ou automatiquement (le démon autovacuum s'en occupe généralement, mais compléter avec une tâche planifiée avec cron ou les tâches planifiées sous Windows est possible). Nous allons voir comment les consulter.

4.6.1 Utilisation des statistiques

- Les statistiques indiquent :
  - la cardinalité d'un filtre → stratégie d'accès
  - la cardinalité d'une jointure → algorithme de jointure
  - la cardinalité d'un regroupement → 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 répar-

tition 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, le niveau de corrélation entre valeurs et place sur le disque, etc.

L'optimiseur peut donc déterminer la sélectivité d'un filtre (prédicat d'une clause `WHERE` ou une condition de jointure) et donc quelle sera la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué.

Ainsi, avec un filtre peu sélectif, `date_embauche = '2006-09-01'`, la requête va ramener pratiquement l'intégralité de la table. PostgreSQL choisira donc une lecture séquentielle de la table, ou *Seq Scan* :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-09-01';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..10901.69 rows=498998 width=40)
                                          (actual rows=499004 loops=1)
   Filter: (date_embauche = '2006-09-01'::date)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 42.624 ms
```

La partie `cost` montre que l'optimiseur estime que la lecture va ramener 498 998 lignes. Comme on peut le voir, ce n'est pas exact : elle en récupère 499 004. Ce n'est qu'une estimation basée sur des statistiques selon la répartition des données et ces estimations seront la plupart du temps un peu erronées. L'important est de savoir si l'erreur est négligeable ou si elle est importante. Dans notre cas, elle est négligeable. On lit aussi que 11 lignes ont été filtrées pendant le parcours (et 499 004 + 11 correspond bien aux 499 015 lignes de la table).

Avec un filtre sur une valeur beaucoup plus sélective, la requête ne ramènera que 2 lignes. L'optimiseur préférera donc passer par l'index que l'on a créé :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-01-01';
```

```
                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
        (cost=0.42..4.44 rows=1 width=41) (actual rows=2 loops=1)
   Index Cond: (date_embauche = '2006-01-01'::date)
 Planning Time: 0.213 ms
 Execution Time: 0.090 ms
```

Dans ce deuxième essai, l'optimiseur estime ramener 1 ligne. En réalité, il en ramène 2. L'estimation reste relativement précise étant donné le volume de données.

Dans le premier cas, l'optimiseur prévoit de sélectionner l'essentiel de la table et 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.

4.6.2 Statistiques des tables et index

- Dans `pg_class`
  - `relpages` : taille
  - `reltuples` : lignes

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;
```

4.6.3 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.

4.6.4 Stockage des statistiques mono-colonne

- Stockage dans `pg_statistic`
  - préférer la vue `pg_stats`
- Une table nouvellement créée n'a pas de statistiques
- Utilisation :

```
SELECT * FROM pg_stats
  WHERE  schemaname = 'public'
  AND tablename     = 'employes'
  AND attname       = 'date_embauche' \gx
```

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

4.6.5 Vue pg_stats

```
-[ RECORD 1 ]----------+--------------------------------------------------
 ↪  ---------
schemaname             | public
tablename              | employes
attname                | date_embauche
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -0.5
most_common_vals       |
 ↪  {2006-03-01,2006-09-01,2000-06-01,2005-03-06,2006-01-01}
most_common_freqs      | {0.214286,0.214286,0.142857,0.142857,0.142857}
histogram_bounds       | {2003-01-01,2006-06-01}
correlation            | 1
most_common_elems      | ¤
most_common_elem_freqs | ¤
elem_count_histogram   | ¤
```

Ce qui prècède est le contenu de `pg_stats` pour la colonne `date_embauche` de la table `employes`.

Trois champs identifient cette 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. Ce n'est pas le cas ici.

null_frac

Cette statistique correspond au pourcentage de valeurs `NULL` dans l'échantillon considéré. Elle est toujours calculée. Il n'y a pas de valeurs nulles dans l'exemple ci-dessus.

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`, `boolean`, `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 (c'est en moyenne le cas ici).

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 COLU`
où `parametre` vaut soit :

- `n_distinct` pour une table standard,
- ou `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 égales (aussi appelées classes), 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 `<`.

4.6.6 Statistiques : multicolonnes

- Pas par défaut
- `CREATE STATISTICS`
- Trois types de statistique
  - nombre de valeurs distinctes
  - dépendances fonctionnelles
  - liste MCV

Par défaut, la commande `ANALYZE` de PostgreSQL calcule des statistiques mono-colonnes uniquement. Elle peut aussi calculer certaines statistiques multicolonnes. En effet, les valeurs des colonnes ne sont pas indépendantes et peuvent varier ensemble.

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.

PostgreSQL supporte trois types de statistiques pour ces objets :

- `ndistinct` pour le nombre de valeurs distinctes sur ces colonnes ;
- `dependencies` pour les dépendances fonctionnelles ;
- `mcv` pour une liste des valeurs les plus fréquentes (depuis la version 12).

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

Prenons un exemple. On peut voir sur ces deux requêtes que les statistiques sont à jour :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris';

                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.019..4.773 rows=10001 loops=1)
   Filter: ((localisation)::text = 'Paris'::text)
   Rows Removed by Filter: 30003
 Planning time: 0.863 ms
 Execution time: 5.289 ms

EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE departement=75;

                              QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.020..7.013 rows=10001 loops=1)
   Filter: (departement = 75)
   Rows Removed by Filter: 30003
 Planning time: 0.219 ms
 Execution time: 7.785 ms
```

Cela fonctionne bien, *i.e.* l'estimation du nombre de lignes (10013) est très proche de la réalité (10001) dans le cas spécifique où le filtre se fait sur une seule colonne. Par contre, si le filtre se fait sur le lieu Paris *et* le département 75, l'estimation diffère d'un facteur 4, à 2506 lignes :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=2506 width=28)
                    (actual time=0.032..7.081 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
   Rows Removed by Filter: 30003
 Planning time: 0.257 ms
 Execution time: 7.767 ms
```

En fait, il y a une dépendance fonctionnelle entre ces deux colonnes (être dans le département 75 implique d'être à Paris), 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 dédié :

```
CREATE STATISTICS stat_services_big (dependencies)
  ON localisation, departement
  FROM services_big;
```

Après création de l'objet, il ne faut pas oublier de calculer les statistiques :

```
ANALYZE services_big;
```

Ceci fait, on peut de nouveau regarder les estimations :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10038 width=28)
                      (actual time=0.008..6.249 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
   Rows Removed by Filter: 30003
 Planning time: 0.121 ms
 Execution time: 6.849 ms
```

Cette fois, l'estimation (10038 lignes) est beaucoup plus proche de la réalité (10001). Cela ne change rien au plan choisi dans ce cas précis, mais dans certains cas la différence peut être énorme.

Maintenant, prenons le cas d'un regroupement :

```
EXPLAIN (ANALYZE)
  SELECT localisation, COUNT(*)
  FROM   services_big
  GROUP BY localisation ;
```

```
                               QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=886.06..886.10 rows=4 width=14)
          (actual time=12.925..12.926 rows=4 loops=1)
   Group Key: localisation
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=6)
                         (actual time=0.010..2.779 rows=40004 loops=1)
 Planning time: 0.162 ms
 Execution time: 13.033 ms
```

L'estimation du nombre de lignes pour un regroupement sur une colonne est très bonne.

À présent, testons avec un regroupement sur deux colonnes :

```
EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.23 rows=16 width=18)
           (actual time=15.830..15.831 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                            (actual time=0.005..3.094 rows=40004 loops=1)
 Planning time: 0.102 ms
 Execution time: 15.860 ms
```

Là aussi, on constate un facteur d'échelle de 4 entre l'estimation (16 lignes) et la réalité (4). Et là aussi, un objet statistique peut fortement aider :

```
DROP STATISTICS IF EXISTS stat_services_big;

CREATE STATISTICS stat_services_big (dependencies,ndistinct)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;

EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.11 rows=4 width=18)
          (actual time=14.351..14.352 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                            (actual time=0.013..2.786 rows=40004 loops=1)
 Planning time: 0.305 ms
 Execution time: 14.413 ms
```

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

PostgreSQL 12 ajoute la méthode MCV (*most common values*) qui permet d'aller plus loin sur l'estimation du nombre de lignes. Notamment, elle permet de mieux estimer le nombre de lignes à partir d'un prédicat utilisant les opérations `<` et `>`. En voici un exemple :

```
DROP STATISTICS stat_services_big;

EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
  Seq Scan on services_big  (cost=0.00..886.06 rows=2546 width=28)
                            (actual time=0.031..19.569 rows=10001 loops=1)
```

```
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.186 ms
 Execution Time: 21.403 ms
```

Il y a donc une erreur d'un facteur 4 (2 546 lignes estimées contre 10 001 réelles) que l'on peut corriger :

```
CREATE STATISTICS stat_services_big (mcv)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;

EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10030 width=28)
                           (actual time=0.017..18.092 rows=10001 loops=1)
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.337 ms
 Execution Time: 18.907 ms
```

Une limitation existait avant PostgreSQL 13 : un seul objet statistique pouvait être utilisé par table et une requête ne pouvait utiliser qu'un seul objet statistique pour chaque table.

4.6.7 Statistiques sur les expressions

```
CREATE STATISTICS employe_big_extract
ON extract('year' FROM date_embauche) FROM employes_big;
```

- Résout le problème des statistiques difficiles à estimer
- Pas créées par défaut
- Ne pas oublier `ANALYZE`
- (Avant v14 : index fonctionnel nécessaire)

À partir de la version 14, il est possible de créer un objet statistique sur des expressions.

Les statistiques sur des expressions permettent de résoudre le problème des estimations sur les résultats de fonctions ou d'expressions. C'est un problème récurrent et impossible à résoudre sans statistiques dédiées.

On voit dans cet exemple que les statistiques pour l'expression `extract('year' from data_embauche)` sont erronées :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter: (date_part('year'::text,
                 (date_embauche)::timestamp without time zone)
                 = '2006'::double precision)
```

L'ordre suivant crée des statistiques supplémentaires sur les résultats de l'expression. Les résultats sont calculés pour un échantilon des lignes et collectés en même temps ques les statistiques mono-colonnes habituelles. Il ne faut pas oublier de lancer manuellement la collecte la première fois :

```
CREATE STATISTICS employe_big_extract
    ON extract('year' FROM date_embauche) FROM employes_big;
ANALYZE employes_big;
```

Les estimations du plan sont désormais correctes :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..12149.22 rows=498998 width=40)
   Filter: (EXTRACT(year FROM date_embauche) = '2006'::numeric)
```

Cet objet statistique apparaît dans `psql` dans un `\d employes_big` :

```
…
Objets statistiques :
    "public.employe_big_extract" ON EXTRACT(year FROM date_embauche) FROM
↪   employes_big
```

Avant la version 14, calculer ces statistiques est possible indirectement, en créant un index fonctionnel sur l'expression, ce qui provoque le calcul de statistiques dédiées. Or l'index fonctionnel n'a pas toujours d'intérêt : dans l'exemple précédent, presque toutes les dates d'embauche sont en 2006, et l'index ne serait donc pas utilisé.

4.6.8 Catalogues pour les statistiques étendues

Vues disponibles :

- `pg_stats_ext`
- `pg_stats_ext_exprs` (pour les expressions, v14)

Les statistiques étendues sont visibles dans des tables comme `pg_statistic_ext` et `pg_statistic_ext_data` (à partir de la version 12), mais il est plus aisé de passer par la vue `pg_stats_ext` :

```
SELECT * FROM pg_stats_ext \gx
```

```
-[ RECORD 1 ]----------+----------------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
attnames               |
exprs                  | {"EXTRACT(year FROM date_embauche)"}
kinds                  | {e}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       |
most_common_val_nulls  |
most_common_freqs      |
most_common_base_freqs |
-[ RECORD 2 ]----------+----------------------------------------
schemaname             | public
tablename              | services_big
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  |
kinds                  | {m}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       | { {Paris,75},{Limoges,52},{Rennes,40},{Nantes,44} }
most_common_val_nulls  | { {f,f},{f,f},{f,f},{f,f} }
most_common_freqs      |
 ↪  {0.2512,0.2511666666666665,0.24886666666666668,0.24876666666666666}
most_common_base_freqs |
 ↪  {0.06310144,0.06308469444444444,0.061934617777777784,0.06188485444444444}
```

On voit qu'il n'y a pas d'informations détaillées sur les statistiques sur expression (première ligne). Elles sont visibles dans `pg_stats_ext_exprs` (à partir de la version 14) :

```
SELECT * FROM pg_stats_ext \gx
```

```
SELECT * FROM pg_stats_ext_exprs  \gx
-[ RECORD 1 ]----------+--------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
expr                   | EXTRACT(year FROM date_embauche)
inherited              | f
null_frac              | 0
avg_width              | 8
```

```
n_distinct             | 1
most_common_vals       | {2006}
most_common_freqs      | {1}
histogram_bounds       |
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

4.6.9 ANALYZE

- `ANALYZE [ VERBOSE ] [ table [ ( colonne [, ...] ) ] [, ...] ]`
  - sans argument : base entière
  - avec argument : table complète ou certaines colonnes
- Un échantillon de table → statistiques
- Table vide : conserve les anciennes statistiques
- Nouvelle table : valeur par défaut

`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 ou plusieurs arguments sont donnés, ils doivent correspondre au nom des tables à analyser (en les séparant par des virgules). 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 le catalogue système nommé `pg_statistics`, accessible par la vue `pg_stats`, comme vu précédemment.

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.
À partir de la version 14, lors de la planification, une table vide est bien considérée comme telle au niveau de son nombre de lignes, mais avec 10 blocs au minimum.
Pour les versions antérieures, une nouvelle table (nouvelle dans le sens `CREATE TABLE` mais aussi `VACUUM FULL` et `TRUNCATE`) n'est jamais considérée vide par l'optimiseur, qui utilise alors des valeurs par défaut dépendant de la largeur moyenne d'une ligne et d'un nombre arbitraire de blocs.

4.6.10 Fréquence d'analyse

- Dépend principalement de la fréquence des requêtes DML
- Autovacuum fait l'`ANALYZE` mais…
  - pas sur les tables temporaires
  - pas assez rapidement parfois
- Cron
  - `psql`
  - ou `vacuumdb --analyze-only`

Les statistiques doivent être mises à jour fréquemment. La fréquence exacte dépend surtout de la fréquence des requêtes d'insertion, de modification ou de 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'outil PostgreSQL pour lancer un seul `ANALYZE`, mais l'outil `vacuumdb` a une option `--analyze-only`. Ces deux ordres sont équivalents :

```
vacuumdb --analyze-only -t matable -d mabase
```

```
psql -c "ANALYZE matable" -d mabase
```

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, il 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 concernées juste après l'insertion ou la mise à jour massive. Pour des mises à jour plus régulières dans une grande table, il est assez fréquent qu'on doive réduire la valeur d'`autovacuum_analyze_scale_factor` (par défaut 10 % de la table doit être modifié pour déclencher automatiquement un `ANALYZE`).

4.6.11 Échantillon statistique

- `default_statistics_target` = 100
  - × 300 → 30 000 lignes au hasard
- Configurable par colonne

```
ALTER TABLE matable ALTER COLUMN nomchamp SET STATISTICS 300 ;
```

- Configurable par statistique étendue (v13+)

```
ALTER STATISTICS nom SET STATISTICS valeur ;
```

- `ANALYZE` ensuite
- Coût : temps de planification

Par défaut, un `ANALYZE` récupère 30 000 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` (100 par défaut). La taille de l'échantillon est de 300 fois `default_statistics_target`.

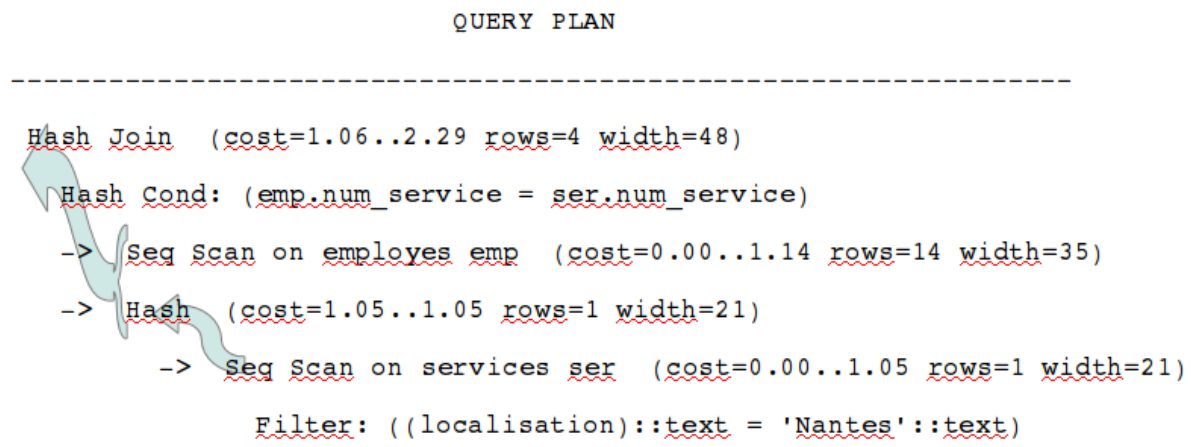
Si on l'augmente, les statistiques seront plus précises grâce à un échantillon plus important. Mais de ce fait, elles seront plus longues à calculer, prendront plus de place sur le disque et en RAM, et demanderont plus de travail au planificateur pour générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages : on évitera de dépasser 1000.

Il est possible de configurer ce paramètrage table par table et colonne par colonne :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS valeur ;
```

Ne pas oublier de relancer un `ANALYZE nom_table ;` juste après.

4.7 LECTURE D'UN PLAN



Un plan d'exécution se lit 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 justes 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 (elle dépend essentiellement de la taille de la table lue, mais aussi d'opération de filtrage). `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 :

```
SET enable_nestloop TO off;
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..2.34 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)

RESET enable_nestloop;
```

Ce plan débute en bas 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 (paramètre `seq_page_cost`) et à la manipulation des 4 lignes de la table `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 2,34. 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.

4.7.1 Rappel des options d'EXPLAIN

- `ANALYZE` : exécution (danger !)
- `BUFFERS` : blocs *read*/*hit*/*written*/*dirtied*, *shared*/*local*/*temp*
- `GENERIC_PLAN` : plan générique (requête préparée, v16)
- `SETTINGS` : paramètres configurés pour l'optimisation
- `WAL` : nombre d'enregistrements et nombre d'octets écrits dans les journaux
- `COSTS` : par défaut
- `TIMING` : par défaut
- `VERBOSE` : colonnes considérées
- `SUMMARY` : temps de planification
- `FORMAT` : sortie en text, XML, JSON, YAML

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, ou au contraire de masquer des informations affichées par défaut.

Option ANALYZE

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

Avec `ANALYZE`, la requête est réellement exécutée ! Attention donc aux `INSERT` / `UPDATE` / `DELETE`. N'oubliez pas non plus qu'un `SELECT` peut appeler des fonctions qui écrivent dans la base. Dans le doute, pensez à englober l'appel dans une transaction que vous annulerez après coup.

Voici un exemple utilisant cette option :

```
BEGIN;
EXPLAIN (ANALYZE) SELECT * FROM employes WHERE matricule < 100 ;
ROLLBACK;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                (actual time=0.004..0.005 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 0.013 ms
```

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 : comparer au nombre de la première parenthèse permet d'avoir une idée de la justesse des statistiques et de l'estimation ;
- `loops` est le nombre d'exécutions de ce nœud, car certains peuvent être répétés de nombreuses fois.

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 n'est en pratique 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 :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM employes WHERE matricule < 100;
```

```
                      QUERY PLAN
-------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                (actual time=0.002..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
   Buffers: shared hit=1
 Planning time: 0.024 ms
 Execution time: 0.011 ms
```

La nouvelle ligne est la ligne `Buffers`. `shared hit` indique un accès à une table ou index dans les *shared buffers* de PostgreSQL. Ces autres indications peuvent se rencontrer :

Informations	Type d'objet concerné	Explications
Shared hit	Table ou index permanent	Lecture d'un bloc dans le cache
Shared read	Table ou index permanent	Lecture d'un bloc hors du cache
Shared written	Table ou index permanent	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

Informations	Type d'objet concerné	Explications
Temp read	Tris et hachages	Lecture d'un bloc
Temp written	Tris et hachages	Écriture d'un bloc

`EXPLAIN (BUFFERS)`, sans `ANALYZE`, fonctionne certes à partir de PostgreSQL 13, mais n'affiche alors que les quelques blocs consommés par la planification.

Option GENERIC_PLAN

Cette option (à partir de PostgreSQL 16) sert quand on cherche le plan générique planifié pour une requête préparée (c'est-à-dire dont les paramètres seront fournis séparément).

```
EXPLAIN (GENERIC_PLAN, SUMMARY ON)
SELECT * FROM t1 WHERE c1 < $1 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.15..14.98 rows=333 width=8)
   Index Cond: (c1 < $1)
 Planning Time: 0.195 ms
```

Option SETTINGS

Cette option permet d'obtenir les valeurs des paramètres spécifiques à l'optimisation de requêtes qui ne sont pas à leur valeur par défaut. Elle est désactivée par défaut.

```
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;
```

```
                               QUERY PLAN
------------------------------------------------------------------------
Index Scan using employes_big_pkey on employes_big (cost=0.42..8.44 rows=1 width=41)
  Index Cond: (matricule = 33)

SET enable_indexscan TO off;
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big  (cost=4.43..8.44 rows=1 width=41)
   Recheck Cond: (matricule = 33)
   ->  Bitmap Index Scan on employes_big_pkey  (cost=0.00..4.43 rows=1 width=0)
         Index Cond: (matricule = 33)
 Settings: enable_indexscan = 'off'
```

Option WAL

Cette option permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions. Elle apparaît avec PostgreSQL 13 et est désactivée par défaut.

```
CREATE TABLE t1 (id integer);
EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT generate_series(1, 1000) ;
```

```
                          QUERY PLAN
--------------------------------------------------------------
 Insert on t1  (cost=0.00..15.02 rows=1000 width=12)
           (actual time=1.457..1.458 rows=0 loops=1)
   WAL: records=2009 bytes=123824
   ->  Subquery Scan on "*SELECT*"
       (cost=0.00..15.02 rows=1000 width=12)
       (actual time=0.003..0.146 rows=1000 loops=1)
         ->  ProjectSet  (cost=0.00..5.02 rows=1000 width=4)
                (actual time=0.002..0.068 rows=1000 loops=1)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
                  (actual time=0.001..0.001 rows=1 loops=1)
 Planning Time: 0.033 ms
 Execution Time: 1.479 ms
```

Option COSTS

Activée par défaut, l'option `COSTS` indique les estimations du planificateur. La désactiver permet de gagner un peu en lisibilité.

```
EXPLAIN (COSTS OFF) SELECT * FROM employes WHERE matricule < 100;
```

```
         QUERY PLAN
----------------------------
 Seq Scan on employes
   Filter: (matricule < 100)
```

```
EXPLAIN (COSTS ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
   Filter: (matricule < 100)
```

Option TIMING

Cette option n'est utilisable qu'avec l'option `ANALYZE` et est activée par défaut. Elle ajoute les informations sur les durées en milliseconde. 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 :

```
EXPLAIN (ANALYZE, TIMING ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
              (actual time=0.003..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.022 ms
 Execution time: 0.010 ms
```

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
```

```
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                                (actual rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.025 ms
 Execution time: 0.010 ms
```

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 nom du schéma, le nom de la fonction qualifié du nom du schéma, le nom du déclencheur (trigger), etc. Elle est désactivée par défaut.

```
EXPLAIN (VERBOSE) SELECT * FROM employes WHERE matricule < 100;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on public.employes  (cost=0.00..1.18 rows=3 width=43)
   Output: matricule, nom, prenom, fonction, manager, date_embauche,
           num_service
   Filter: (employes.matricule < 100)
```

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, et permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Par défaut, un `EXPLAIN` simple n'affiche pas le résumé, mais un `EXPLAIN ANALYZE` le fait.

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)

EXPLAIN (SUMMARY ON) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
 Planning time: 0.014 ms

EXPLAIN (ANALYZE) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
              (actual time=0.002..0.003 rows=14 loops=1)
 Planning time: 0.013 ms
 Execution time: 0.009 ms

EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM employes;
```

```
                      QUERY PLAN
------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
               (actual time=0.002..0.003 rows=14 loops=1)
```

Option FORMAT

Par défaut, la sortie est sous forme d'un texte destiné à être lu par un humain, 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 :

```
EXPLAIN (FORMAT XML) SELECT * FROM employes WHERE matricule < 100;
```

```
                         QUERY PLAN
------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                               +
     <Plan>                                              +
       <Node-Type>Seq Scan</Node-Type>                   +
       <Parallel-Aware>false</Parallel-Aware>            +
       <Relation-Name>employes</Relation-Name>           +
       <Alias>employes</Alias>                           +
       <Startup-Cost>0.00</Startup-Cost>                 +
       <Total-Cost>1.18</Total-Cost>                     +
       <Plan-Rows>3</Plan-Rows>                          +
       <Plan-Width>43</Plan-Width>                       +
       <Filter>(matricule &lt; 100)</Filter>             +
     </Plan>                                             +
   </Query>                                              +
 </explain>
(1 row)
```

Les signes `+` en fin de ligne indiquent un retour à la ligne lors de l'utilisation de l'outil `psql`. Il est possible de ne pas les afficher en configurant l'option `format` de `psql` à `unaligned`. Cela se fait ainsi :

```
\pset format unaligned
```

Ces formats semi-structurés sont utilisés principalement par des outils, car le contenu est plus facile à analyser, voire un peu plus complet.

4.7.2 Statistiques, cardinalités & 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. Cherchons comment calculer la cardinalité d'un filtre simple sur une table `employes` de 14 lignes, par exemple `WHERE num_service = 1`.

Ici, la valeur recherchée se trouve directement dans le tableau des valeurs les plus fréquentes (dans les champs `most_common_vals` et `most_common_freqs`) la cardinalité peut être calculée directement.

```
SELECT * FROM pg_stats
WHERE tablename = 'employes'
  AND attname   = 'num_service' \gx

-[ RECORD 1 ]----------+-----------------------------------------------
schemaname             | public
tablename              | employes
attname                | num_service
inherited              | f
null_frac              | 0
```

```
avg_width                 | 4
n_distinct                | -0.2857143
most_common_vals          | {4,3,2,1}
most_common_freqs         | {0.35714287,0.2857143,0.21428572,0.14285715}
histogram_bounds          | ¤
correlation               | 0.10769231
…
```

La requête suivante permet de récupérer la fréquence d'apparition de la valeur recherchée :

```
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 = 'employes'
           AND attname = 'num_service') get_mcv
 WHERE value = 1;
```

```
 tablename |   attname   | value |   freq
-----------+-------------+-------+----------
 employes  | num_service |     1 | 0.142857
```

Si l'on n'avait pas eu affaire à une des valeurs les plus fréquentes, il aurait fallu passer par l'histogramme des valeurs (`histogram_bounds`, ici vide car il y a trop peu de valeurs), pour calculer d'abord la sélectivité du filtre pour en déduire ensuite la cardinalité.

Une fois cette fréquence obtenue, l'optimiseur calcule la cardinalité du prédicat `WHERE num_service = 1` en la multipliant avec le nombre total de lignes de la table :

```
SELECT 0.142857 * reltuples AS cardinalite_predicat
  FROM pg_class
 WHERE relname = 'employes';
```

```
 cardinalite_predicat
----------------------
             1.999998
```

Le calcul est cohérent avec le plan d'exécution de la requête impliquant la lecture de `employes` sur laquelle on applique le prédicat évoqué plus haut :

```
EXPLAIN SELECT * FROM employes WHERE num_service = 1;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (num_service = 1)
```

Calcul de coût

Notre table `employes` peuplée de 14 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 `employes` est calculé à partir de deux composantes. Tout 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` (coût d'accès à un bloc dans un parcours complet) sera appliqué ensuite pour obtenir le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS cout_acces
  FROM pg_class
 WHERE relname = 'employes';

 relname  | cout_acces
----------+------------
 employes |          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 multiplie donc par `cpu_tuple_cost` (0,01 par défaut) 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 = 'employes';

 relname  | cout
----------+------
 employes | 1.14
```

Le calcul est bon :

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

Avec un filtre dans la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE date_embauche='2006-01-01'`, il faut non seulement extraire les lignes les unes après les autres, mais 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 = 'employes';

 relname  | cost
----------+-------
 employes | 1.175
```

Ce nombre se retrouve dans le plan, à l'arrondi près :

```
EXPLAIN SELECT * FROM employes WHERE date_embauche='2006-01-01';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (date_embauche = '2006-01-01'::date)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL contient un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus[4].

[4] https://docs.postgresql.fr/current/planner-stats-details.html

4.8 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- Un plan est composé de nœuds
- qui produisent des données
- ou en consomment et en retournent
- Chaque nœud consomme les données produites par le(s) nœud(s) parent(s)
- Le nœud final retourne les données à l'utilisateur

Les plans sont extrêmement sensibles aux données elles-mêmes bien sûr, aux paramètres, aux tailles réelles des objets, à la version de PostgreSQL, à l'ordre des données dans la table, voire au moment du passage d'un `VACUUM`. Il n'est donc pas étonnant de trouver parfois des plans différents de ceux reproduits ici pour une même requête.

4.8.1 Nœuds de type parcours

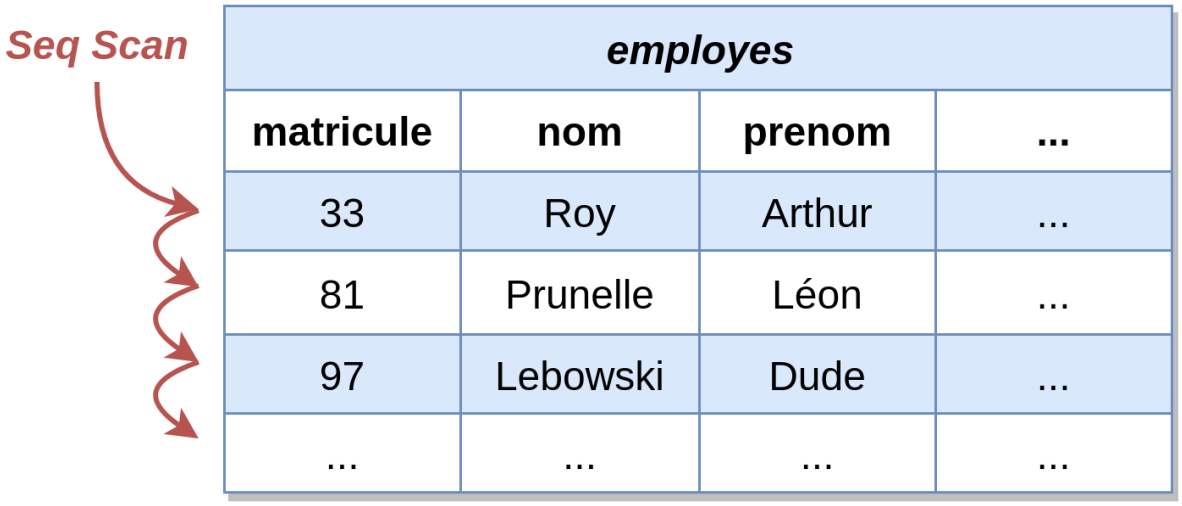
- Parcours de table
- Parcours d'index
- Autres parcours

Par parcours, on entend le renvoi d'un ensemble de lignes provenant soit d'un fichier, soit d'un traitement. Le fichier peut correspondre à une table ou à une vue matérialisée, et on parle dans ces deux cas d'un parcours de table. Le fichier peut aussi correspondre à un index, auquel cas on parle de parcours d'index. Un parcours peut être un traitement dans différents cas, principalement celui d'une procédure stockée.

4.8.2 Parcours de table

- *Seq Scan*
- *Parallel Seq Scan*

4.8.3 Parcours de table : Seq Scan



Seq Scan :

Les parcours de tables sont les principales opérations qui lisent les données des tables (normales, temporaires ou non journalisées) et des vues matérialisées. Elles ne prennent donc pas d'autre nœud en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

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.

Ce nœud apparaît lorsque la requête nécessite de lire l'intégralité ou la majorité de la table :

```
EXPLAIN SELECT * FROM employes;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

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

```
EXPLAIN SELECT * FROM employes WHERE matricule=135;
```

```
                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=1 width=43)
   Filter: (matricule = 135)
```

4.8.4 Parcours de table : paramètres

- *Seq Scan*
  - `seq_page_cost` (défaut : 1)
  - `cpu_tuple_cost` & `cpu_operator_cost`
  - `enable_seqscan`
- *Parallel Seq Scan*
  - `parallel_tuple_cost`, `min_parallel_table_scan_size`
  - et les autres paramètres de la parallélisation

Seq Scan :

Le coût d'un *Seq Scan* sera fonction du nombre de blocs à parcourir, et donc du paramètre `seq_page_cost`, ainsi que du nombre de lignes à décoder et, optionnellement, à filtrer. La valeur de `seq_page_cost`, par défaut à 1, est rarement modifiée ; elle est surtout importante comparée à `random_page_cost`.

Parallel Seq Scan :

Il est possible d'avoir un parcours parallélisé d'une table (*Parallel Seq Scan*). La parallélisation doit être activée comme décrit plus haut, notamment :

- les paramètres `max_parallel_workers_per_gather` et `max_parallel_workers` doivent être tous deux supérieurs à 0, ce qui est le cas par défaut ;
- la table à traiter doit avoir une taille minimale (`min_parallel_table_scan_size` est à 8 Mo) ; et plus elle sera grosse plus il y aura de *workers* ;

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. Il y a cependant un coût pour chaque ligne (paramètre `parallel_tuple_cost`). Autrement dit, dans la majorité des cas, il faut un filtre sur une table importante pour que la parallélisation se déclenche.

Dans les exemples suivants, la parallélisation est activée :

```
SET max_parallel_workers_per_gather TO 5 ;   /*  défaut : 2 */

-- Plan d'exécution parallélisé
EXPLAIN SELECT * FROM employes_big WHERE num_service <> 4;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: (num_service <> 4)
```

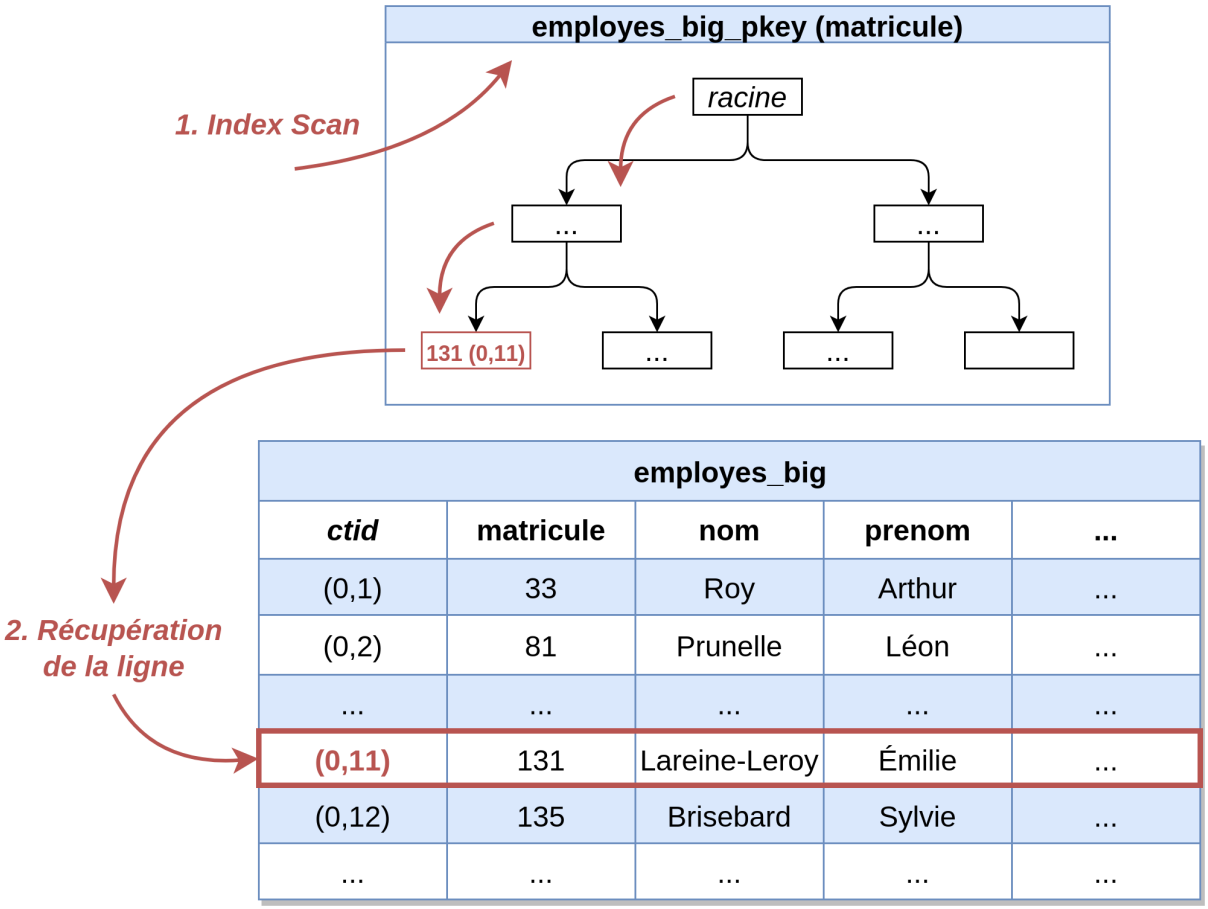
Ici, deux processus supplémentaires seront exécutés pour réaliser la requête. Dans le cas de ce type de parcours, chaque processus prend un bloc et traite toutes les lignes de ce bloc. Quand un processus a terminé de traiter son bloc, il regarde quel est le prochain bloc à traiter et le traite. (À partir de la version 14, il prend même un groupe de blocs pour profiter de la fonctionnalité de *read ahead* du noyau.)

4.8.5 Parcours d'index

- *Index Scan*
- *Index Only Scan*
- *Bitmap Index Scan*
- et leurs versions parallélisées (B-Tree)

PostgreSQL dispose de trois moyens d'accéder aux données à travers les index. Le plus connu est l'*Index Scan*, qui possède plusieurs variantes.

4.8.6 Index Scan



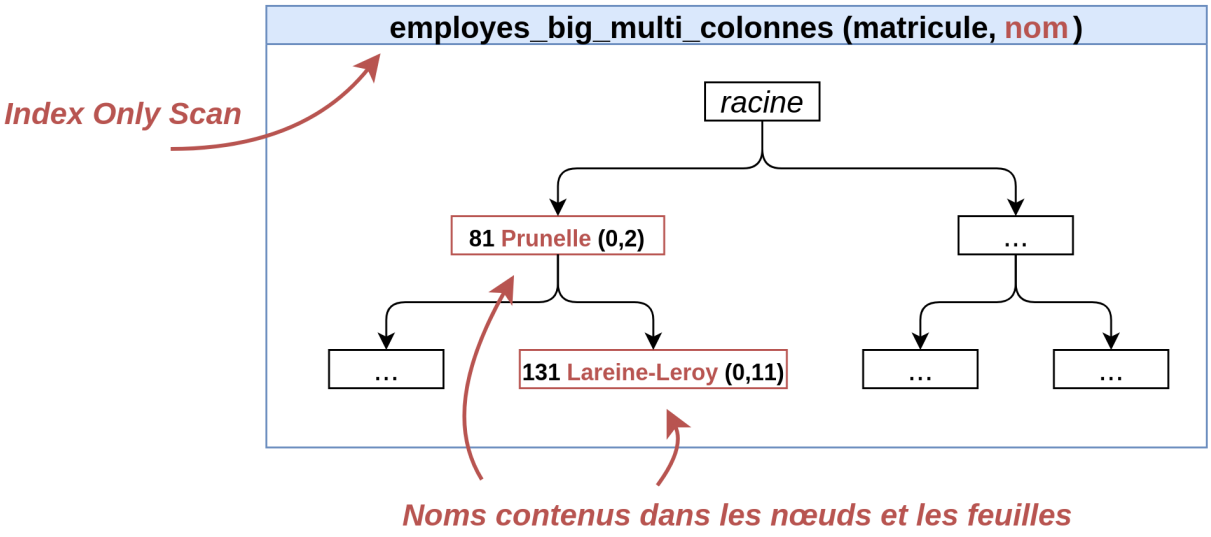
Le nœud *Index Scan* consiste à parcourir les blocs d'index jusqu'à trouver les pointeurs vers les blocs contenant les données. À chaque pointeur trouvé, PostgreSQL lit le bloc de la table pointée pour retrouver l'enregistrement et s'assurer notamment de sa visibilité pour la transaction en cours. De ce fait, il y a beaucoup d'accès non séquentiels pour lire l'index et la table.

```
EXPLAIN SELECT * FROM employes_big WHERE matricule = 132;
```

```
                    QUERY PLAN
---------------------------------------------------
 Index Scan using employes_big_pkey on employes_big
   (cost=0.42..8.44 rows=1 width=41)
   Index Cond: (matricule = 132)
```

Ce type de nœud ne permet pas d'extraire directement les données à retourner depuis l'index, sans passer par la lecture des blocs correspondants de la table.

4.8.7 Index Only Scan : principe



4.8.8 Index Only Scan : utilité & limites

- Très performant
- Besoin d'un `VACUUM` récent
  - sinon trop de *Heap Fetches*
- Ne fonctionne pas pour les index fonctionnels
  - ajouter une colonne générée ?

Utilité :

Le nœud *Index Only Scan* est une version plus performante de l'*Index Scan*. Il est choisi si **toutes** les colonnes de la table concernées par la requête font partie de l'index :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule
FROM   employes_big
WHERE  matricule < 100000 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using employes_big_pkey on employes_big (actual time=0.019..8.195
 ↪  rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Heap Fetches: 0
   Buffers: shared hit=274
```

```
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.083 ms
 Execution Time: 11.123 ms
```

Il n'y a donc plus besoin d'accéder à la table et l'on économise de nombreux accès disque. C'est d'autant plus appréciable que les lignes sont nombreuses.

Par contre, si l'on ajoute le champ `nom` dans la requête, l'optimiseur se rabat sur un *Index Scan*. En effet, `nom` ne peut être lu que dans la table puisqu'il ne fait pas partie de l'index. Dans notre exemple avec de petites tables, le temps reste correct mais il est moins bon :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule, nom
FROM employes_big
WHERE matricule < 100000 ;

                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using employes_big_pkey on employes_big (actual time=0.009..14.562
 ↪  rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Buffers: shared hit=1199
 Planning:
   Buffers: shared hit=60
 Planning Time: 0.181 ms
 Execution Time: 18.170 ms
```

Noter le nombre de blocs lus beaucoup plus important.

Index couvrants :

Il existe des index dits « couvrants », destinés à favoriser des *Index Only Scans*. Ils peuvent contenir des données en plus des champs indexés, même avec un index unique comme ici.

```
CREATE UNIQUE INDEX ON employes_big (matricule) INCLUDE (nom) ;
```

Cet index contient aussi `nom` et permet de revenir à de très bonnes performances :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule, nom
FROM   employes_big
WHERE  matricule < 100000 ;

                          QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using employes_big_matricule_nom_idx on employes_big (actual
 ↪  time=0.010..5.769 rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Heap Fetches: 0
   Buffers: shared hit=383
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.064 ms
 Execution Time: 7.747 ms
```

L'expérience montre qu'un index classique multicolonne (sur `(matricule,nom)`) sera aussi efficace, voire plus, car il peut être plus petit (la clause `INCLUDE` inhibe la déduplication). La clause `INCLUDE` reste utile pour faire d'une clé primaire, unique ou étrangère, un index couvrant et économiser un peu de place disque.

Conditions pour obtenir un Index Only Scan :

- Toutes les colonnes de la table utilisées par la requête doivent figurer dans l'index, aussi bien celles retournées que celles servant aux calculs ou au filtrage. En pratique, cela réserve l'*Index Only Scan* au cas où peu de colonnes de la table sont concernées par la requête.
- Il est courant d'ajouter des index dédiés aux requêtes critiques n'utilisant que quelques colonnes. Mais plus le nombre de colonnes et la taille de l'index augmentent, moins PostgreSQL sera tenté par l'*Index Only Scan*.
- Pour que l'*Index Only Scan* soit réellement efficace, il faut que la table soit fréquemment traitée par des opérations `VACUUM`. En effet, les informations de visibilité des lignes ne sont pas stockées dans l'index. Pour savoir si la ligne trouvée dans l'index est visible ou pas par la session, il faut soit aller vérifier dans la table (et on en revient à un *Index Scan*), soit aller voir dans la *visibility map* de la table que le bloc ne contient pas de lignes potentiellement mortes, ce qui est beaucoup plus rapide. Le choix se fait à l'éxécution pour chaque ligne. Dans l'idéal, le plan indique qu'il n'y a jamais eu à aller dans la table (*heap*) ainsi :

```
Heap Fetches: 0
```

S'il y a un trop grand nombre de *Heap Fetches*, l'*Index Only Scan* perd tout son intérêt. Il peut suffire de rendre l'`autovacuum` beaucoup plus agressif sur la table.

Limite pour les index fonctionnels :

Le planificateur a du mal à définir un *Index Only Scan* pour un index fonctionnel. Il échoue par exemple ici :

```
CREATE INDEX employes_big_upper_idx ON employes_big (upper (nom)) ;
-- Créer les statistiques pour l'index fonctionnel
ANALYZE employes_big ;

EXPLAIN (COSTS OFF)
SELECT upper(nom) FROM employes_big WHERE upper(nom) = 'CRUCHOT' ;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Index Scan using employes_big_upper_idx on employes_big
   Index Cond: (upper((nom)::text) = 'CRUCHOT'::text)
```

L'index fonctionnel est bien utilisé mais ce n'est pas un *Index Only Scan*.

Dans certains cas critiques, il peut être intéressant de créer une colonne générée reprenant la fonction et de créer un index dessus, qui permettra un *Index Only Scan* :

```
-- Attention : réécriture de la table
ALTER TABLE employes_big
ADD COLUMN nom_maj text GENERATED ALWAYS AS ( upper(nom) ) STORED ;
```

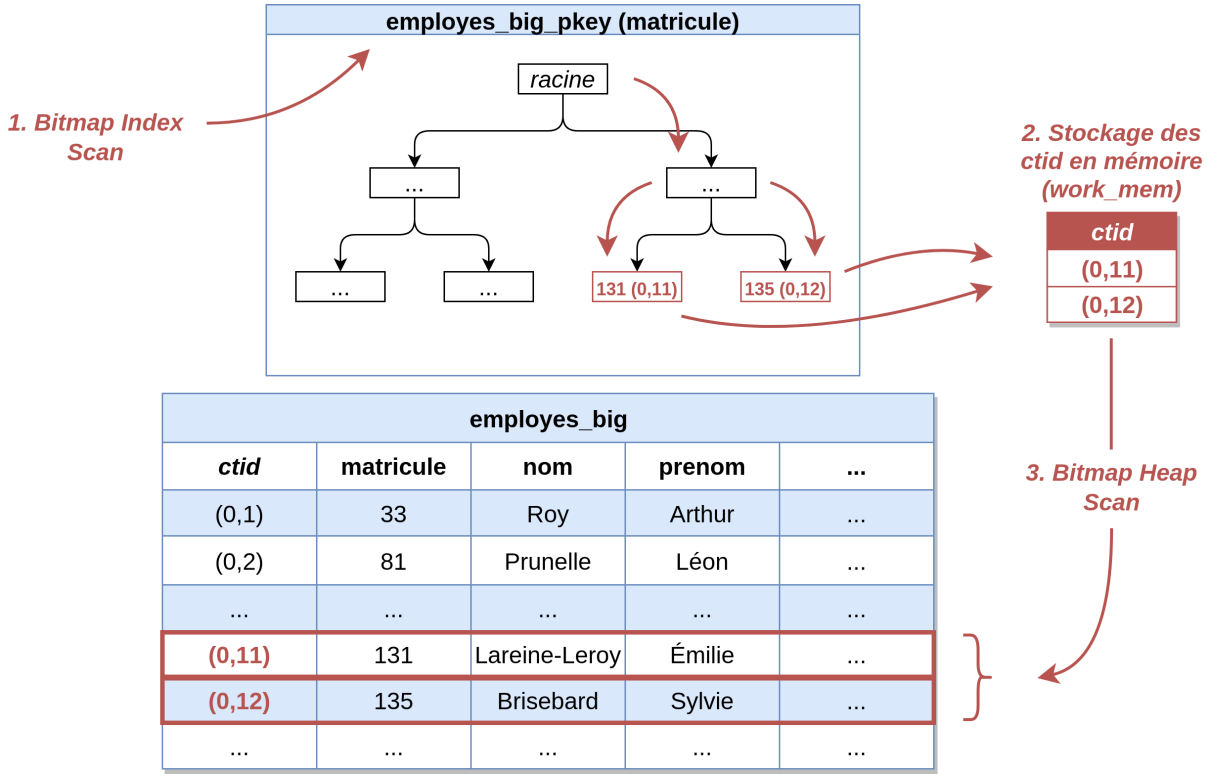
```
CREATE INDEX ON employes_big (nom_maj) ;
-- Créer les statistiques pour la nouvelle colonne
-- et éviter les Heap Fetches
VACUUM ANALYZE employes_big ;

EXPLAIN (COSTS OFF) SELECT nom_maj FROM employes_big WHERE nom_maj  = 'CRUCHOT' ;

                        QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using employes_big_nom_maj_idx on employes_big
   Index Cond: (nom_maj = 'CRUCHOT'::text)
```

La nouvelle colonne générée est bien sûr maintenue automatiquement si `nom` change et n'est pas modifiable directement. Mais sa création entraîne la réécriture de la table, qui d'ailleurs grossit un peu, et il faut adapter le code.

4.8.9 Bitmap Scan



Ce dernier parcours est particulièrement efficace pour des opérations de type *Range Scan*, c'est-à-dire où PostgreSQL doit retourner une plage de valeurs, ou pour combiner le résultat de la lecture de plusieurs index.

Contrairement à d'autres SGBD, un index *bitmap* de PostgreSQL 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).

```
SET enable_indexscan TO off ;

EXPLAIN
  SELECT * FROM employes_big WHERE matricule BETWEEN 200000 AND 300000;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=2108.46..8259.35 rows=99126 width=41)
   Recheck Cond: ((matricule >= 200000) AND (matricule <= 300000))
   ->  Bitmap Index Scan on employes_big_pkey*
       (cost=0.00..2083.68 rows=99126 width=0)
         Index Cond: ((matricule >= 200000) AND (matricule <= 300000))
```

```
RESET enable_indexscan;
```

Exemple de combinaison du résultat de la lecture de plusieurs index :

```
EXPLAIN
  SELECT * FROM employes_big
  WHERE matricule BETWEEN   1000 AND 100000
     OR matricule BETWEEN 200000 AND 300000;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=4265.09..12902.67 rows=178904 width=41)
   Recheck Cond: (((matricule >= 1000) AND (matricule <= 100000))
               OR ((matricule >= 200000) AND (matricule <= 300000)))
   ->  BitmapOr  (cost=4265.09..4265.09 rows=198679 width=0)
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2091.95 rows=99553 width=0)
               Index Cond: ((matricule >= 1000) AND (matricule <= 100000))
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2083.68 rows=99126 width=0)
               Index Cond: ((matricule >= 200000) AND (matricule <= 300000))
```

4.8.10 Parcours d'index : paramètres importants

- `random_page_cost` (4 ou moins ?)
- `cpu_index_tuple_cost`
- `effective_cache_size` (⅔ de la RAM ?)
- Selon le disque :
  - `effective_io_concurrency`
  - `maintenance_io_concurrency`
- `min_parallel_index_scan_size`
- `enable_indexscan`, `enable_indexonlyscan`, `enable_bitmapscan`

Index Scan :

L'*Index Scan* n'a d'intérêt que s'il y a très peu de lignes à récupérer, surtout si les disques sont mécaniques. Il faut donc que le filtre soit très sélectif.

Le rapport entre les paramètres `seq_page_cost` et `random_page_cost` est d'importance majeure dans le choix face à un *Seq Scan*. Plus il est proche de 1, plus les parcours d'index seront favorisés par rapport aux parcours de table.

Index Only Scan :

Il n'y a pas de paramètre dédié à ce parcours. S'il est possible, l'optimiseur le préfère à un *Index Scan*.

Bitmap Index Scan :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[5], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[6].

Toujours à partir de la version 13, un nouveau paramètre apparaît : `maintenance_io_concurrency`. Il a le même but que `effective_io_concurrency`, mais pour les opérations de maintenance, non les requêtes. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Sa valeur par défaut est de 10, et il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

[5] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[6] https://docs.postgresql.fr/13/release.html

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système (total du *shared buffers* et du cache du système). Une bonne pratique est de positionner ce paramètre à ⅔ 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 *Bitmap Index Scan*.

Parallélisation

Il est possible de paralléliser les parcours d'index. Cela donne donc les nœuds *Parallel Index Scan*, *Parallel Index Only Scan* et *Parallel Bitmap Heap Scan*. Cette infrastructure est actuellement uniquement utilisée pour les index B-Tree. Par contre, pour le *Bitmap Scan*, seul le parcours de la table est parallélisé. Un parcours parallélisé d'un index n'est considéré qu'à partir du moment où l'index a une taille supérieure à la valeur du paramètre `min_parallel_index_scan_size` (512 ko par défaut).

4.8.11 Autres parcours

- *Function Scan*
- *Values Scan*
- …et d'autres

On retrouve le nœud *Function Scan* lorsqu'une requête utilise directement le résultat d'une fonction, comme par exemple, dans des fonctions d'informations système de PostgreSQL :

```
EXPLAIN SELECT * from pg_get_keywords();
```

```
                            QUERY PLAN
------------------------------------------------------------------------
 Function Scan on pg_get_keywords  (cost=0.03..4.03 rows=400 width=65)
```

Il existe d'autres types de parcours, rarement rencontrés. Ils sont néanmoins détaillés en annexe[7].

[7] https://dali.bo/j6_html

4.8.12 Nœuds 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
- Pour `EXISTS`, `IN` et certaines jointures externes
  - *Hash Semi Join* & *Hash Anti Join*
- Paramètres :
  - `work_mem` (et `hash_mem_multiplier`)
  - `seq_page_cost` & `random_page_cost`.
  - `enable_nestloop`, `enable_hashjoin`, `enable_mergejoin`

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

Nested Loop :

La *Nested Loop* se retrouve principalement dans les jointures de petits ensembles de données. Dans l'exemple suivant, le critère sur `services` ramène très peu de lignes, il ne coûte pas grand-chose d'aller piocher à chaque fois dans l'index de `employes_big`.

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

```
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.42..10053.94 rows=124754 width=46)
   ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
         Filter: ((localisation)::text = 'Nantes'::text)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
         (cost=0.42..7557.81 rows=249508 width=33)
         Index Cond: (num_service = ser.num_service)
```

Hash Join :

Le *Hash Join* se retrouve lorsque l'ensemble de la table interne est petit. L'optimiseur construit alors une table de hachage avec les valeurs de la ou les colonne(s) de jointure de la table interne. Il réalise ensuite un parcours de la table externe, et, pour chaque ligne de celle-ci, recherche des lignes cor-

respondantes dans la table de hachage, toujours en utilisant la ou les colonne(s) de jointure comme clé

```
EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes_big emp
JOIN   services ser ON (emp.num_service = ser.num_service);

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=0.19..8154.54 rows=499015 width=45)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes_big emp  (cost=0.00..5456.55 rows=499015 width=32)
   ->  Hash  (cost=0.14..0.14 rows=4 width=21)
         ->  Seq Scan on services ser  (cost=0.00..0.14 rows=4 width=21)
```

Cette opération réclame de la mémoire de tri, visible avec `EXPLAIN (ANALYZE)` (dans le pire des cas, ce sera un fichier temporaire).

Merge Join :

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 :

```
EXPLAIN
  SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM   employes_big emp
  JOIN   services_big ser ON (emp.num_service = ser.num_service)
  ORDER BY ser.num_service;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=0.82..20094.77 rows=499015 width=49)
   Merge Cond: (emp.num_service = ser.num_service)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
       (cost=0.42..13856.65 rows=499015 width=33)
   ->  Index Scan using services_big_pkey on services_big ser
       (cost=0.29..1337.35 rows=40004 width=20)
```

Il s'agit d'un algorithme de jointure particulièrement efficace pour traiter les volumes de données importants, surtout si les données sont pré-triées grâce à l'existence d'un index.

Hash Anti/Semi Join :

Les clauses `EXISTS` et `NOT EXISTS` mettent également en œuvre des algorithmes dérivés de semi et anti-jointures. En voici un exemple avec la clause `EXISTS` :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE EXISTS (SELECT 1
                FROM employes_big e
                WHERE e.date_embauche>s.date_creation
                  AND s.num_service = e.num_service) ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Hash Semi Join  (cost=17841.84..19794.91 rows=1 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Un plan sensiblement identique s'obtient avec `NOT EXISTS`. Le nœud *Hash Semi Join* est remplacé par *Hash Anti Join* :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE NOT EXISTS (SELECT 1
                    FROM employes_big e
                    WHERE e.date_embauche>s.date_creation
                      AND s.num_service = e.num_service);

                          QUERY PLAN
---------------------------------------------------------------
 Hash Anti Join  (cost=17841.84..19794.93 rows=3 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Hash Join parallélisé :

Ces nœuds sont parallélisables. Pour les *Parallel Hash Join*, la table hachée est même commune pour les différents *workers* [8], et le calcul de celle-ci est réparti sur ceux-ci. Par contraste, pour les nœuds *Merge Join*, *Nested Loop* et *Hash Join* (non complètement parallélisé), seul le parcours de la table externe peut être parallélisé, tandis que la table interne est parcourue (voire hachée) entièrement par chaque worker.

Exemple (testé en version 16.1) :

```
CREATE TABLE foo(id serial, a int);
CREATE TABLE bar(id serial, foo_a int, b int);
INSERT INTO foo(a) SELECT i*2 FROM generate_series(1,1000000) i;
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,100) i;
VACUUM ANALYZE foo, bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;

                                QUERY PLAN
---------------------------------------------------------------------------------
 Gather (actual time=0.192..21.305 rows=33 loops=1)
   Output: foo.a, bar.b
```

[8] https://write-skew.blogspot.com/2018/01/parallel-hash-for-postgresql.html

```
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join (actual time=10.757..16.903 rows=11 loops=3)
         Output: foo.a, bar.b
         Hash Cond: (foo.a = bar.foo_a)
         Worker 0:  actual time=16.118..16.120 rows=0 loops=1
         Worker 1:  actual time=16.132..16.134 rows=0 loops=1
         ->  Parallel Seq Scan on public.foo (actual time=0.009..12.961 rows=111111
↪  loops=3)
               Output: foo.id, foo.a
               Filter: ((foo.a % 3) = 0)
               Rows Removed by Filter: 222222
               Worker 0:  actual time=0.011..12.373 rows=102953 loops=1
               Worker 1:  actual time=0.011..12.440 rows=102152 loops=1
         ->  Hash (actual time=0.022..0.023 rows=100 loops=3)
               Output: bar.b, bar.foo_a
               Buckets: 1024  Batches: 1  Memory Usage: 12kB
               Worker 0:  actual time=0.027..0.027 rows=100 loops=1
               Worker 1:  actual time=0.026..0.026 rows=100 loops=1
               ->  Seq Scan on public.bar (actual time=0.008..0.013 rows=100 loops=3)
                     Output: bar.b, bar.foo_a
                     Worker 0:  actual time=0.012..0.017 rows=100 loops=1
                     Worker 1:  actual time=0.011..0.016 rows=100 loops=1
 Planning Time: 0.116 ms
 Execution Time: 21.321 ms
```

Dans ce plan, la table externe `foo` est lue de manière parallélisée, les trois processus se partageant son contenu. Chacun a sa version de la toute petite table interne `bar`, qui est hachée trois fois (les deux *workers* et le processus principal lisent les 100 lignes).

Si `bar` est beaucoup plus grosse que `foo`, le plan bascule sur un *Parallel Hash Join* dont `bar` est cette fois la table externe :

```
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,300000) i;
VACUUM ANALYZE bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather (actual time=69.490..95.741 rows=100033 loops=1)
   Output: foo.a, bar.b
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Hash Join (actual time=66.408..84.866 rows=50016 loops=2)
         Output: foo.a, bar.b
         Hash Cond: (bar.foo_a = foo.a)
         Worker 0:  actual time=63.450..83.008 rows=52081 loops=1
         ->  Parallel Seq Scan on public.bar (actual time=0.002..5.332 rows=150050
↪  loops=2)
               Output: bar.id, bar.foo_a, bar.b
               Worker 0:  actual time=0.002..5.448 rows=148400 loops=1
         ->  Parallel Hash (actual time=49.467..49.468 rows=166666 loops=2)
               Output: foo.a
```

```
                Buckets: 262144 (originally 8192)  Batches: 4 (originally 1)  Memory
↪  Usage: 5344kB
                Worker 0:  actual time=48.381..48.382 rows=158431 loops=1
                ->  Parallel Seq Scan on public.foo (actual time=0.007..21.265
↪  rows=166666 loops=2)
                      Output: foo.a
                      Filter: ((foo.a % 3) = 0)
                      Rows Removed by Filter: 333334
                      Worker 0:  actual time=0.009..20.909 rows=158431 loops=1
Planning Time: 0.083 ms
Execution Time: 97.567 ms
```

Le *Hash* de `foo` se fait par deux processus qui ne traitent cette fois que la moitié du million de lignes, en en filtrant les ⅔ (la dernière ligne indique quasiment le tiers de 500 000). Le coût de démarrage de ce *hash* parallélisé est cependant assez lourd (la jointure ne commence qu'après 66 ms). Pour la même requête, PostgreSQL 10, qui ne connaît pas le *Parallel Hash Join*, procédait à un *Hash Join* classique et prenait 50 % plus longtemps. Précisons encore qu'augmenter `work_mem` ne change pas le plan, mais permet cas d'accélérer un peu les hachages (réduction du nombre de *batches*).

4.8.13 Nœuds de tris et de regroupements

- Deux nœuds de tri :
  - *Sort*
  - *Incremental Sort*
- Regroupement/agrégation :
  - *Aggregate*
  - *Hash Aggregate*
  - *Group Aggregate*
  - *Mixed Aggregate*
  - *Partial/Finalize Aggregate*
- Paramètres :
  - `enable_hashagg`
  - `work_mem` & `hash_mem_multiplier` (v13)

Pour réaliser un tri, PostgreSQL dispose de deux nœuds : *Sort* et *Incremental Sort*. Leur efficacité va dépendre du paramètre `work_mem` qui va définir la quantité de mémoire que PostgreSQL pourra utiliser pour un tri.

Sort :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes ORDER BY fonction;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Sort  (cost=1.41..1.44 rows=14 width=43)
       (actual time=0.013..0.014 rows=14 loops=1)
   Sort Key: fonction
   Sort Method: quicksort  Memory: 26kB
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
                     (actual time=0.003..0.004 rows=14 loops=1)
 Planning time: 0.021 ms
 Execution time: 0.021 ms
```

Si le tri ne tient pas en mémoire, l'algorithme de tri gère automatiquement le débordement sur disque (26 Mo ici) :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes_big ORDER BY fonction;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Sort  (cost=70529.24..71776.77 rows=499015 width=40)
   (actual time=252.827..298.948 rows=499015 loops=1)
   Sort Key: fonction
   Sort Method: external sort  Disk: 26368kB
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
                            (actual time=0.003..29.012 rows=499015 loops=1)
 Planning time: 0.021 ms
 Execution time: 319.283 ms
```

Cependant, un tri est coûteux. Donc si un index existe sur le champ de tri, PostgreSQL aura tendance à l'utiliser. Les données sont déjà triées dans l'index, il suffit de le parcourir pour lire les lignes dans l'ordre :

```
EXPLAIN SELECT * FROM employes_big  ORDER BY matricule;

                   QUERY PLAN
-------------------------------------------------
 Index Scan using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

Et ce, dans n'importe quel ordre de tri :

```
EXPLAIN SELECT * FROM employes_big ORDER BY matricule DESC;

                     QUERY PLAN
-----------------------------------------------------
 Index Scan Backward using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

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

Comme vu précédemment, PostgreSQL sait utiliser un index pour trier les données. Cependant, dans certains cas, il ne sait pas utiliser l'index alors qu'il pourrait le faire. Prenons un exemple.

Voici un jeu de données contenant une table à trois colonnes, et un index sur une colonne :

```
DROP TABLE IF exists t1;
CREATE TABLE t1 (c1 integer, c2 integer, c3 integer);
INSERT INTO t1 SELECT i, i+1, i+2 FROM generate_series(1, 10000000) AS i;
CREATE INDEX t1_c2_idx ON t1(c2);
VACUUM ANALYZE t1;
```

PostgreSQL sait utiliser l'index pour trier les données. Par exemple, voici le plan d'exécution pour un tri sur la colonne `c2` (colonne indexée au niveau de l'index `t1_c2_idx`) :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using t1_c2_idx on t1  (cost=0.43..313749.06 rows=10000175 width=12)
                                (actual time=0.016..1271.115 rows=10000000 loops=1)
   Buffers: shared hit=81380
 Planning Time: 0.173 ms
 Execution Time: 1611.868 ms
```

Par contre, si on essaie de trier par rapport aux colonnes `c2` et `c3`, les versions 12 et antérieures ne savent pas utiliser l'index, comme le montre ce plan d'exécution :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Gather Merge  (cost=697287.64..1669594.86 rows=8333480 width=12)
           (actual time=1331.307..3262.511 rows=10000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=54149, temp read=55068 written=55246
   ->  Sort  (cost=696287.62..706704.47 rows=4166740 width=12)
         (actual time=1326.112..1766.809 rows=3333333 loops=3)
         Sort Key: c2, c3
         Sort Method: external merge  Disk: 61888kB
         Worker 0:  Sort Method: external merge  Disk: 61392kB
         Worker 1:  Sort Method: external merge  Disk: 92168kB
         Buffers: shared hit=54149, temp read=55068 written=55246
         ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                              (actual time=0.015..337.901 rows=3333333 loops=3)
               Buffers: shared hit=54055
 Planning Time: 0.068 ms
 Execution Time: 3716.541 ms
```

Comme PostgreSQL ne sait pas utiliser un index pour réaliser ce tri, il passe par un parcours de table (parallélisé dans le cas présent), puis effectue le tri, ce qui prend beaucoup de temps, encore plus s'il faut déborder sur disque. La durée d'exécution a plus que doublé.

Incremental Sort :

La version 13 est beaucoup plus maline à cet égard. Elle est capable d'utiliser l'index pour faire un premier tri des données (sur la colonne `c2` d'après notre exemple), puis elle trie les données du résultat par rapport à la colonne `c3` :

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
```

```
 Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
             (actual time=0.082..2427.099 rows=10000000 loops=1)
   Sort Key: c2, c3
   Presorted Key: c2
   Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
   Buffers: shared hit=81387
   ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                 (actual time=0.007..1263.517 rows=10000000 loops=1)
         Buffers: shared hit=81380
 Planning Time: 0.059 ms
 Execution Time: 2766.530 ms
```

La requête en version 12 prenait 3,7 secondes. La version 13 n'en prend que 2,7 secondes. On remarque un nouveau type de nœud, le *Incremental Sort*, qui s'occupe de re-trier les données après un renvoi de données triées, grâce au parcours d'index. Le plan distingue bien la clé déjà triée (`c2`) et celles à trier (`c2, c3`).

L'apport en performance est déjà très intéressant, mais il devient remarquable si on utilise une clause `LIMIT`. Voici le résultat en version 12 :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3 LIMIT 10;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=186764.17..186765.34 rows=10 width=12)
      (actual time=718.576..724.791 rows=10 loops=1)
   Buffers: shared hit=54149
   ->  Gather Merge  (cost=186764.17..1159071.39 rows=8333480 width=12)
                         (actual time=718.575..724.788 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=54149
         ->  Sort  (cost=185764.15..196181.00 rows=4166740 width=12)
                   (actual time=716.606..716.608 rows=10 loops=3)
               Sort Key: c2, c3
               Sort Method: top-N heapsort  Memory: 25kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
               Buffers: shared hit=54149
               ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                                    (actual time=0.010..347.085 rows=3333333 loops=3)
                     Buffers: shared hit=54055
 Planning Time: 0.044 ms
 Execution Time: 724.818 ms
```

Et celui en version 13 :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.48..1.24 rows=10 width=12) (actual time=0.027..0.029 rows=10 loops=1)
   Buffers: shared hit=4
   ->  Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
                            (actual time=0.027..0.027 rows=10 loops=1)
```

```
        Sort Key: c2, c3
        Presorted Key: c2
        Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak
↪  Memory: 25kB
        Buffers: shared hit=4
        ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                          (actual time=0.012..0.014 rows=11 loops=1)
              Buffers: shared hit=4
Planning Time: 0.052 ms
Execution Time: 0.038 ms
```

La requête passe donc de 724 ms à 0,029 ms.

En version 16, l'*Incremental Sort* peut aussi servir à accélérer les `DISTINCT` :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT DISTINCT c2,c1,c3 FROM t1;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique (actual time=39.208..2479.229 rows=10000000 loops=1)
   Buffers: shared read=81380
   ->  Incremental Sort (actual time=39.206..1841.165 rows=10000000 loops=1)
         Sort Key: c2, c1, c3
         Presorted Key: c2
         Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
         Buffers: shared read=81380
         ->  Index Scan using t1_c2_idx on t1 (actual time=39.182..949.921
↪  rows=10000000 loops=1)
               Buffers: shared read=81380
 Planning:
   Buffers: shared read=3
 Planning Time: 0.274 ms
 Execution Time: 2679.447 ms
```

Cela devrait accélérer de nombreuses requêtes qui possèdent abusivement une clause `DISTINCT` ajoutée par un ETL, si le premier champ du `DISTINCT` est par chance indexé, comme ici :

```
EXPLAIN (COSTS OFF) SELECT DISTINCT date_commande, *  FROM commandes ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique
   ->  Incremental Sort
         Sort Key: date_commande, numero_commande, client_id, mode_expedition,
↪  commentaire
         Presorted Key: date_commande
         ->  Index Scan using commandes_date_commande_idx on commandes
```

Aggregate :

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

```
EXPLAIN SELECT count(*) FROM employes;

                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=1.18..1.19 rows=1 width=8)
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=0)
```

Hash Aggregate :

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

```
EXPLAIN SELECT fonction, count(*) FROM employes GROUP BY fonction;

                           QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=1.21..1.27 rows=6 width=20)
   Group Key: fonction
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=12)
```

Avant la version 13, l'inconvénient de ce nœud 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 nœud *HashAggregate* consomme plusieurs gigaoctets de mémoire et sature ainsi la mémoire du serveur.

La version 13 améliore cela en utilisant le disque à partir du moment où la mémoire nécessaire dépasse la multiplication de la valeur du paramètre `work_mem` et celle du paramètre `hash_mem_multiplier` (2 par défaut à partir de la version 15, 1 auparavant). La requête sera plus lente, mais la mémoire ne sera pas saturée.

Group Aggregate :

Lorsque l'optimiseur estime que le volume de données à traiter ne tient pas dans `work_mem` ou quand il peut accéder aux données pré-triées, il utilise plutôt l'algorithme *GroupAggregate* :

```
EXPLAIN SELECT matricule, count(*) FROM employes_big GROUP BY matricule;

                          QUERY PLAN
---------------------------------------------------------------
 GroupAggregate  (cost=0.42..20454.87 rows=499015 width=12)
   Group Key: matricule
   Planned Partitions: 16
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

Mixed Aggregate :

Le *Mixed Aggregate* est très efficace pour les clauses `GROUP BY GROUPING SETS` ou `GROUP BY CUBE` grâce à l'utilisation de *hashs* :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT manager, fonction, num_service, COUNT(*)
  FROM employes_big
  GROUP BY CUBE(manager,fonction,num_service) ;

                          QUERY PLAN
-------------------------------------------------------------------
 MixedAggregate  (cost=0.00..34605.17 rows=27 width=27)
        (actual time=581.562..581.573 rows=51 loops=1)
   Hash Key: manager, fonction, num_service
   Hash Key: manager, fonction
   Hash Key: manager
   Hash Key: fonction, num_service
   Hash Key: fonction
   Hash Key: num_service, manager
   Hash Key: num_service
   Group Key: ()
   Batches: 1  Memory Usage: 96kB
   Buffers: shared hit=4664
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=19)
                        (actual time=0.015..35.840 rows=499015 loops=1)
         Buffers: shared hit=4664
 Planning time: 0.223 ms
 Execution time: 581.671 ms
```

(Comparer avec le plan et le temps obtenus auparavant, que l'on peut retrouver avec `SET enable_hashagg TO off;`)

Le calcul d'un agrégat peut être parallélisé. Dans ce cas, deux nœuds 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 :

```
EXPLAIN (ANALYZE,COSTS OFF)
  SELECT date_embauche, count(*), min(date_embauche), max(date_embauche)
  FROM employes_big
  GROUP BY date_embauche;

                               QUERY PLAN
-----------------------------------------------------------------------
 Finalize GroupAggregate (actual time=92.736..92.740 rows=7 loops=1)
   Group Key: date_embauche
   ->  Sort (actual time=92.732..92.732 rows=9 loops=1)
         Sort Key: date_embauche
         Sort Method: quicksort  Memory: 25kB
         ->  Gather (actual time=92.664..92.673 rows=9 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial HashAggregate
                   (actual time=89.531..89.532 rows=3 loops=3)
                     Group Key: date_embauche
                     ->  Parallel Seq Scan on employes_big
                         (actual time=0.011..35.801 rows=166338 loops=3)
 Planning time: 0.127 ms
 Execution time: 95.601 ms
```

4.8.14 Les autres nœuds

- *Limit*
- *Unique* (`DISTINCT`)
- *Append* (`UNION ALL`), *Except*, *Intersect*
- *Gather* (parallélisme)
- *InitPlan*, *Subplan*, etc.
- *Memoize* (14+)

Limit :

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

```
EXPLAIN SELECT * FROM employes_big LIMIT 1;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.00..0.02 rows=1 width=40)
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
```

Le nœud *Sort* utilisera dans ce cas une méthode de tri appelée *top-N heapsort* qui permet d'optimiser le tri pour retourner les n premières lignes :

```
EXPLAIN ANALYZE
  SELECT * FROM employes_big ORDER BY fonction LIMIT 5;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Limit  (cost=17942.61..17942.62 rows=5 width=40)
      (actual time=80.359..80.360 rows=5 loops=1)
   ->  Sort  (cost=17942.61..19190.15 rows=499015 width=40)
                (actual time=80.358..80.359 rows=5 loops=1)
         Sort Key: fonction
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on employes_big
             (cost=0.00..9654.15 rows=499015 width=40)
             (actual time=0.005..27.506 rows=499015 loops=1)
 Planning time: 0.035 ms
 Execution time: 80.375 ms
```

Unique :

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

```
EXPLAIN SELECT DISTINCT matricule FROM employes_big;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique  (cost=0.42..14217.19 rows=499015 width=4)
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

On le verra plus loin, il est souvent plus efficace d'utiliser `GROUP BY` pour dédoublonner les résultats d'une requête.

Append, Except, Intersect :

Les nœuds *Append*, *Except* et *Intersect* se rencontrent avec les opérateurs ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Par exemple, avec `UNION ALL` :

```
EXPLAIN
  SELECT * FROM employes
  WHERE num_service = 2
  UNION ALL
  SELECT * FROM employes
  WHERE num_service = 4;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Append  (cost=0.00..2.43 rows=8 width=43)
   ->  Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
         Filter: (num_service = 2)
   ->  Seq Scan on employes employes_1  (cost=0.00..1.18 rows=5 width=43)
         Filter: (num_service = 4)
```

InitPlan :

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

```
EXPLAIN
  SELECT *,
    (SELECT nom_service FROM services WHERE num_service=1)
  FROM employes WHERE num_service = 1;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=2 width=101)
   Filter: (num_service = 1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=10)
           Filter: (num_service = 1)
```

SubPlan :

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

```
EXPLAIN
  SELECT * FROM employes
    WHERE num_service NOT IN (SELECT num_service FROM services
                              WHERE nom_service = 'Consultants');
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=7 width=43)
   Filter: (NOT (hashed SubPlan 1))
```

```
SubPlan 1
  ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=4)
        Filter: ((nom_service)::text = 'Consultants'::text)
```

Gather :

Le nœud *Gather* n'apparaît que s'il y a du parallélisme. Il est utilisé comme nœud de rassemblement des données.

Memoize :

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop*.

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

4.9 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.

4.9.1 Statistiques pas à jour

Les statistiques sont-elles à jour ?

- Traitement lourd
  - faire tout de suite `ANALYZE`
- Table trop grosse
  - régler l'échantillonnage
  - régler l'autovacuum sur cette table
- Retard de mise à jour suite à crash ou restauration

Il est fréquent que les statistiques ne soient pas à jour. Il peut y avoir plusieurs causes.

Gros chargement :

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. 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. À fortiori si le traitement complet est imbriqué dans une seule transaction.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. Notamment, 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;
```

Volumétrie importante :

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

Il est possible d'augmenter la taille de l'échantillon de données analysées, ainsi que la précision des statistiques, pour les colonnes où cela est important, à l'aide de cet ordre vu plus haut :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS valeur;
```

Autre problème : l'autovacuum se base par défaut sur la proportion de lignes modifiées par rapport à celles existantes pour savoir s'il doit déclencher un `ANALYZE` (10 % par défaut). Au fil du temps, beaucoup de tables grossissent en accumulant des lignes statiques. À volume d'activité constante, les lignes actives représentent alors une proportion de plus en plus faible et l'autovacuum se déclenche de moins en moins souvent. Il est courant de descendre la valeur de `autovacuum_analyze_scale_factor` pour compenser. On peut aussi chercher à isoler les données statiques dans leur partition.

Perte des statistiques d'activité après un arrêt brutal :

Le fonctionnement du collecteur des statistiques d'activité implique qu'un arrêt brutal de PostgreSQL les réinitialise. (Il s'agit des statistiques sur les lignes insérées ou modifiées, que l'on trouve notamment dans `pg_stat_user_tables`, pas des statistiques sur les données, qui sont bien préservées.) Elles ne sont pas directement utilisées par le planificateur, mais cette réinitialisation peut mener à un retard dans l'activité de l'autovacuum et la mise à jour des statistiques des données. Après un plantage, un arrêt en mode immédiat ou une restauration physique, il est donc conseillé de relancer un `ANALYZE` général (et même un `VACUUM` ensuite si possible.)

4.9.2 Colonnes corrélées

```
SELECT * FROM corr1 WHERE c1=1 AND c2=1
```

- Si `c1 = 1` pour 20 % des lignes
- et `c2 = 1` pour 10 % des lignes
- Alors le planificateur calcule : 2 % des lignes (20 % × 10 %)
    - Mais en réalité ?
- Pour corriger :

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
```

PostgreSQL conserve des statistiques par colonne simple. Mais dans la vie, les valeurs ne sont pas indépendantes. Dans cet exemple, les calculs d'estimation du résultat seront mauvais :

```
CREATE TABLE corr1 AS
SELECT mod(i,5) AS c1 ,mod(i,10) AS c2, i FROM generate_series (1,100000) i;
CREATE INDEX ON corr1 (c1,c2) ;
SELECT c1,c2, count(*) FROM corr1 GROUP BY 1,2 ORDER BY 1,2;
```

```
 c1 | c2 | count
----+----+-------
  0 |  0 | 10000
  0 |  5 | 10000
  1 |  1 | 10000
  1 |  6 | 10000
  2 |  2 | 10000
…
(10 lignes)
```

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`. Faute de mieux, il multiplie les deux estimations et obtient 2 % (20 % × 10 %), soit environ 2000 lignes, ce qui est faux :

```
ANALYZE corr1 ;
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=29.40..636.28 rows=2059 width=12)
                     (actual time=0.653..3.034 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..28.88 rows=2059 width=0)
                                    (actual time=0.480..0.481 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Pour corriger cela, il faut générer des statistiques sur plusieurs colonnes spécifiques. Ce n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS`.

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
ANALYZE corr1 ;    /* ne pas oublier */

EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=139.85..867.39 rows=10103 width=12)
                      (actual time=0.748..3.505 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..137.32 rows=10103 width=0)
                                     (actual time=0.563..0.564 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Dans ce cas précis, de meilleures statistiques ne changent pas le plan. Par contre, avec le critère `c1 = 1 AND c2 = 2` (qui ne renvoie rien), les meilleures statistique permettent de basculer du même *Bitmap Scan* que ci-dessus à un *Index Scan* plus léger :

```
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 2 ;

                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using corr1_c1_c2_idx on corr1  (cost=0.29..8.31 rows=1 width=12)
                                    (actual time=0.010..0.011 rows=0 loops=1)
   Index Cond: ((c1 = 1) AND (c2 = 2))
```

4.9.3 La jointure de trop

- PostgreSQL choisit l'ordre des jointures
  - uniquement pour les X premières tables
  - où X = `join_collapse_limit` (défaut : 8)
- Les jointures supplémentaires sont ajoutées *après*
- … d'où plans non optimaux
- → augmenter `join_collapse_limit` si nécessaire (12-15)
  - ainsi que `from_collapse_limit`

Voici un exemple complet de ce problème. Disons que `join_collapse_limit` est configuré à 2 (le défaut est en réalité 8).

```
SET join_collapse_limit TO 2 ;
```

Nous allons déjà créer deux tables et les peupler avec 1 million de lignes chacune :

```
CREATE TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 1000000);
CREATE TABLE t2 (id integer);
INSERT INTO t2 SELECT generate_series(1, 1000000);
ANALYZE;
```

Maintenant, nous allons demander le plan d'exécution pour une jointure entre les deux tables :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=30832.00..70728.00 rows=1000000 width=8)
            (actual time=2355.012..6141.672 rows=1000000 loops=1)
   Hash Cond: (t1.id = t2.id)
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
                       (actual time=0.012..1137.629 rows=1000000 loops=1)
   ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
```

```
            (actual time=2354.750..2354.753 rows=1000000 loops=1)
         Buckets: 131072  Batches: 16  Memory Usage: 3227kB
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                             (actual time=0.008..1144.492 rows=1000000 loops=1)
 Planning Time: 0.095 ms
 Execution Time: 7246.491 ms
```

PostgreSQL choisit de lire la table `t2`, de remplir une table de hachage avec le résultat de cette lecture, puis de parcourir la table `t1`, et enfin de tester la condition de jointure grâce à la table de hachage.

Ajoutons maintenant une troisième table, sans données cette fois :

```
CREATE TABLE t3 (id integer);
```

Et ajoutons une jointure à la requête précédente. Cela nous donne cette requête :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id;
```

Son plan d'exécution, avec la configuration par défaut de PostgreSQL, sauf le `join_collapse_limit` à 2, est :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=77972.88..80334.59 rows=2550 width=12)
         (actual time=2902.385..2913.956 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Merge Join  (cost=76972.88..79079.59 rows=1062 width=12)
                   (actual time=2894.440..2894.615 rows=0 loops=3)
         Merge Cond: (t1.id = t3.id)
         ->  Sort  (cost=76793.10..77834.76 rows=416667 width=8)
                   (actual time=2894.405..2894.572 rows=1 loops=3)
               Sort Key: t1.id
               Sort Method: external merge  Disk: 5912kB
               Worker 0:  Sort Method: external merge  Disk: 5960kB
               Worker 1:  Sort Method: external merge  Disk: 5848kB
               ->  Parallel Hash Join  (cost=15428.00..32202.28 rows=416667 width=8)
                                       (actual time=1892.071..2400.515 rows=333333 loops=3)
                     Hash Cond: (t1.id = t2.id)
                     ->  Parallel Seq Scan on t1  (cost=0.00..8591.67 rows=416667
↪  width=4)
                                                  (actual time=0.007..465.746 rows=333333
↪  loops=3)
                     ->  Parallel Hash  (cost=8591.67..8591.67 rows=416667 width=4)
                                        (actual time=950.509..950.514 rows=333333 loops=3)
                           Buckets: 131072  Batches: 16  Memory Usage: 3520kB
                           ->  Parallel Seq Scan on t2  (cost=0.00..8591.67
↪  rows=416667 width=4)
                                                        (actual time=0.017..471.653 rows=333333
↪  loops=3)
         ->  Sort  (cost=179.78..186.16 rows=2550 width=4)
                   (actual time=0.028..0.032 rows=0 loops=3)
```

```
               Sort Key: t3.id
               Sort Method: quicksort  Memory: 25kB
               Worker 0:  Sort Method: quicksort  Memory: 25kB
               Worker 1:  Sort Method: quicksort  Memory: 25kB
               ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                   (actual time=0.019..0.020 rows=0 loops=3)
 Planning Time: 0.120 ms
 Execution Time: 2914.661 ms
```

En effet, dans ce cas, PostgreSQL va trier les jointures sur les 2 premières tables (soit `t1` et `t2`), et il ajoutera ensuite les autres jointures dans l'ordre indiqué par la requête. Donc, ici, il joint `t1` et `t2`, puis le résultat avec `t3`, ce qui nous donne une requête exécutée en un peu moins de 3 secondes. C'est beaucoup quand on considère que la table `t3` est vide et que le résultat sera forcément vide lui aussi (l'optimiseur a certes estimé trouver 2550 lignes dans `t3`, mais cela reste très faible par rapport aux autres tables).

Maintenant, voici le plan d'exécution pour la même requête avec un `join_collapse_limit` à 3 :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=35861.44..46281.24 rows=2550 width=12)
         (actual time=14.943..15.617 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join  (cost=34861.44..45026.24 rows=1062 width=12)
                  (actual time=0.119..0.134 rows=0 loops=3)
         Hash Cond: (t2.id = t1.id)
         ->  Parallel Seq Scan on t2  (cost=0.00..8591.67 rows=416667 width=4)
                                      (actual time=0.010..0.011 rows=1 loops=3)
         ->  Hash  (cost=34829.56..34829.56 rows=2550 width=8)
                   (actual time=0.011..0.018 rows=0 loops=3)
               Buckets: 4096  Batches: 1  Memory Usage: 32kB
               ->  Hash Join  (cost=30832.00..34829.56 rows=2550 width=8)
                              (actual time=0.008..0.013 rows=0 loops=3)
                     Hash Cond: (t3.id = t1.id)
                     ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                         (actual time=0.006..0.007 rows=0 loops=3)
                     ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
                               (never executed)
                           ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000
↪  width=4)
                                               (never executed)
 Planning Time: 0.331 ms
 Execution Time: 15.662 ms
```

Déjà, on voit que la planification a pris plus de temps. La durée reste très basse (0,3 milliseconde) ceci dit.

Cette fois, PostgreSQL commence par joindre `t3` à `t1`. Comme `t3` ne contient aucune ligne, `t1`

n'est même pas parcourue (texte `never executed`) et le résultat de cette première jointure renvoie 0 lignes. De ce fait, la création de la table de hachage est très rapide. La table de hachage étant vide, le parcours de `t2` est abandonné après la première ligne lue. Cela nous donne une requête exécutée en 15 millisecondes.

Une configuration adéquate de `join_collapse_limit` est donc essentielle pour de bonnes performances, notamment sur les requêtes réalisant un grand nombre de jointures.

Il est courant de monter `join_collapse_limit` à 12 si l'on a des requêtes avec autant de tables (y compris celles des vues).

Il existe un paramètre très voisin, `from_collapse_limit`, qui définit à quelle profondeur « aplatir » les sous-requêtes. On le monte à la même valeur que `join_collapse_limit`.

Comme le temps de planification augmente très vite avec le nombre de tables, il vaut mieux ne pas monter `join_collapse_limit` beaucoup plus haut sans tester que ce n'est pas contre-productif. Dans la session concernée, il reste possible de définir :

```
SET join_collapse_limit = … ;
SET from_collapse_limit = … ;
```

À l'inverse, la valeur 1 permet de forcer les jointures dans l'ordre de la clause `FROM`, ce qui est à réserver aux cas désespérés.

Au-delà de 12 tables intervient encore un autre mécanisme, l'optimiseur génétique (GEQO[9]). Pour limiter le nombre de plans étudiés, seul un échantillonnage aléatoire est testé puis recombiné.

4.9.4 Prédicats et statistiques

```
SELECT *
FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

- L'optimiseur n'a pas de statistiques sur le résultat de la fonction `extract`
- Il estime la sélectivité du prédicat à 0,5 % …
- `CREATE STATISTIC` (v14)

Lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen de 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 alors que la requête va ramener 2495 lignes :

[9] https://docs.postgresql.fr/current/geqo-pg-intro.html

```
EXPLAIN
  SELECT * FROM employes_big
  WHERE extract('year' from date_embauche) = 2006;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter: (date_part('year'::text,
                 (date_embauche)::timestamp without time zone)
                 = '2006'::double precision)
```

Ces 2495 lignes correspondent à 0,5 % de la table `employes_big`.

Nous avons vu qu'il est préférable de réécrire la requête de manière à pouvoir utiliser les statistiques existantes sur la colonne, mais ce n'est pas toujours aisé ou même possible.

Dans ce cas, on peut se rabattre sur l'ordre `CREATE STATISTICS`. Nous avons vu plus haut qu'il permet de calculer des statistiques sur des résultats d'expressions (ne pas oublier `ANALYZE`).

```
CREATE STATISTICS employe_big_extract
    ON extract('year' from date_embauche) FROM employes_big;
ANALYZE employes_big;
```

Les estimations du plan sont désormais correctes :

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..12149.22 rows=498998 width=40)
   Filter: (EXTRACT(year FROM date_embauche) = '2006'::numeric)
```

Avant PostgreSQL 14, il est nécessaire de créer un index fonctionnel sur l'expression pour que des statistiques soient calculées.

4.9.5 Problème avec LIKE

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

- PostgreSQL peut utiliser un index dans ce cas
- **MAIS** si l'encodage n'est pas C
  - déclarer l'index avec une classe d'opérateur
  - `varchar_pattern_ops` / `text_pattern_ops`, etc.

```
CREATE INDEX ON matable (champ_texte varchar_pattern_ops);
```

- Outils pour `LIKE '%mot%'` :
  - `pg_trgm`,
  - *Full Text Search*

Il existe cependant une spécificité à PostgreSQL : dans le cas d'une recherche avec préfixe, il peut utiliser directement un index sur la colonne si l'encodage est « C ». Or le collationnement par défaut d'une base est presque toujours `en_US.UTF-8` ou `fr_FR.UTF-8`, selon les choix à l'installation de l'OS ou de PostgreSQL :

```
\l
                      Liste des bases de données
    Nom    | Propriétaire | Encodage | Collationnement | Type caract. | …
-----------+--------------+----------+-----------------+--------------+---
 pgbench   | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 postgres  | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 template0 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 template1 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 textes_10 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
```

Il faut alors 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);
```

Ce n'est qu'à cette condition qu'un `LIKE 'mot%'` pourra utiliser l'index. Par contre, l'opérateur `varchar_pattern_ops` ne permet pas de trier (`ORDER BY` notamment), faute de collation, il faudra donc peut-être indexer deux fois la colonne.

Un encodage C (purement anglophone) ne nécessite pas l'ajout d'une classe d'opérateurs `varchar_pattern_ops`.

Pour les recherches à l'intérieur d'un texte (`LIKE '%mot%'`), il existe deux autres options :

- `pg_trgm` est une extension permettant de faire des recherches de type par trigramme et un index GIN ou GiST ;
- la *Full Text Search* est une fonctionnalité extrêmement puissante, mais avec une syntaxe différente.

4.9.6 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 (2 312 835 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 !

4.9.7 Dédoublonnage

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

- `DISTINCT` est souvent utilisé pour dédoublonner les lignes
  - souvent utilisé de manière abusive
  - tri !!
  - barrière à l'optimisation
- Penser à :
  - `DISTINCT ON`
  - `GROUP BY`
- Une clé primaire permet de dédoublonner efficacement

Un `DISTINCT` est une opération coûteuse à cause du tri nécessaire. Il est fréquent de le voir ajouté abusivement, « par prudence » ou pour compenser un bug de jointure. De plus il constitue une « barrière à l'optimisation » s'il s'agit d'une partie de requête.

Si le résultat contient telles quelles les clés primaires de toutes les tables jointes, le `DISTINCT` est mathématiquement inutile ! PostgreSQL ne sait malheureusement pas repérer tout seul ce genre de cas.

Quand le dédoublonnage est justifié, il faut savoir qu'il y a deux alternatives principales au `DISTINCT`. Leurs efficacités relatives sont très dépendantes du paramétrage mémoire (`work_mem`) ou des volumétries, ou encore de la présence d'index permettant d'éviter le tri.

- Un `GROUP BY` des colonnes retournées est fastidieux à coder, mais donne parfois un plan efficace. Cette astuce est plus fréquemment utile avant PostgreSQL 13.
- Une autre possibilité est d'utiliser la syntaxe `DISTINCT ON (champs)`, qui renvoie la première ligne rencontrée sur une clé fournie (documentation[10]).

Exemples (sous PostgreSQL 15.2, configuration par défaut sur une petite configuration, cache chaud) :

Il s'agit ici d'afficher la liste des membres des différents services.

- Plan avec `DISTINCT` : notez le tri sur disque.

[10] https://docs.postgresql.fr/current/sql-select.html#SQL-DISTINCT

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Unique (actual time=2930.441..4765.048 rows=499015 loops=1)
   ->  Sort (actual time=2930.435..3351.819 rows=499015 loops=1)
         Sort Key: employes_big.matricule, employes_big.nom, employes_big.prenom,
↪  employes_big.fonction, employes_big.manager, employes_big.date_embauche,
↪  employes_big.num_service, services.nom_service, services.localisation,
↪  services.departement
         Sort Method: external merge  Disk: 38112kB
         ->  Hash Join (actual time=0.085..1263.867 rows=499015 loops=1)
               Hash Cond: (employes_big.num_service = services.num_service)
               ->  Seq Scan on employes_big (actual time=0.030..273.710 rows=499015
↪  loops=1)
               ->  Hash (actual time=0.032..0.035 rows=4 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on services (actual time=0.014..0.020 rows=4 loops=1)
 Planning Time: 0.973 ms
 Execution Time: 4938.634 ms
```

- Réécriture avec `GROUP BY` : il n'y a pas de gain en temps dans ce cas précis, mais il n'y a plus de tri sur disque, car l'index sur la clé primaire est utilisé. Noter que PostgreSQL est assez malin pour repérer les clés primaire (ici `matricule` et `num_service`). Il évite alors d'inclure dans les données à regrouper ces clés, et tous les champs de la première table.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service)
GROUP BY
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Group (actual time=0.409..5067.075 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   ->  Incremental Sort (actual time=0.405..3925.924 rows=499015 loops=1)
         Sort Key: employes_big.matricule, services.nom_service,
↪  services.localisation, services.departement
         Presorted Key: employes_big.matricule
         Full-sort Groups: 15595  Sort Method: quicksort  Average Memory: 28kB  Peak
↪  Memory: 28kB
         ->  Nested Loop (actual time=0.092..2762.395 rows=499015 loops=1)
```

```
               ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.050..861.828 rows=499015 loops=1)
               ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
                     Cache Key: employes_big.num_service
                     Cache Mode: logical
                     Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory
↪  Usage: 1kB
                     ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                           Index Cond: (num_service = employes_big.num_service)
 Planning Time: 0.900 ms
 Execution Time: 5190.287 ms
```

- Si l'on monte `work_mem` de 4 à 100 Mo, les deux versions basculent sur ce plan, ici plus efficace, qui n'utilise plus l'index, mais ne trie qu'en mémoire, avec la même astuce que ci-dessus.

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 HashAggregate (actual time=3122.612..3849.449 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   Batches: 1  Memory Usage: 98321kB
   ->  Hash Join (actual time=0.136..1354.195 rows=499015 loops=1)
         Hash Cond: (employes_big.num_service = services.num_service)
        ->  Seq Scan on employes_big (actual time=0.050..322.423 rows=499015 loops=1)
         ->  Hash (actual time=0.042..0.046 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on services (actual time=0.020..0.026 rows=4 loops=1)
 Planning Time: 0.967 ms
 Execution Time: 3970.353 ms
```

- La technique la plus propre consiste à indiquer quel est le critère fonctionnel pour dédupliquer. Ici, la structure des tables impose qu'il n'y ait qu'un service par matricule, ce n'est pas évident en regardant la requête. Les index suffisent à ramener une ligne de `service` pour chacune d'`employes_big`.

```
RESET work_mem ;

EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT ON (matricule)
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Unique (actual time=0.093..3812.414 rows=499015 loops=1)
   ->  Nested Loop (actual time=0.090..2741.919 rows=499015 loops=1)
         ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.049..847.356 rows=499015 loops=1)
         ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
               Cache Key: employes_big.num_service
               Cache Mode: logical
```

```
              Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory Usage: 1kB
              ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                    Index Cond: (num_service = employes_big.num_service)
Planning Time: 0.711 ms
Execution Time: 3982.201 ms
```

- Le plus propre et performant reste tout de même de remarquer que les deux clés primaires sont dans le résultat, et que le `DISTINCT` est inutile. La jointure peut se faire de manière plus classique.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join (actual time=0.083..1014.796 rows=499015 loops=1)
   Hash Cond: (employes_big.num_service = services.num_service)
   ->  Seq Scan on employes_big (actual time=0.027..214.360 rows=499015 loops=1)
   ->  Hash (actual time=0.032..0.036 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on services (actual time=0.013..0.019 rows=4 loops=1)
 Planning Time: 0.719 ms
 Execution Time: 1117.126 ms
```

Si les `DISTINCT` sont courants et critiques dans votre application, notez que le nœud est parallélisable depuis PostgreSQL 15.

4.9.8 Index inutilisés

- Statistiques pas à jour/peu précises/oubliées
- Trop de lignes retournées
- Ordre des colonnes de l'index (B-tree)
- Index trop gros
- Prédicat avec transformation

```
WHERE col1 + 2 > 5  →  WHERE col1 > 5 - 2
```

- Opérateur non supporté par l'index

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

- Paramètres
  - `random_page_cost`
  - `effective_cache_size`

Il est relativement fréquent de créer soigneusement un index, et que PostgreSQL ne daigne pas l'utiliser. Il peut y avoir plusieurs raisons à cela.

Problème de statistiques :

Les statistiques de la table peuvent être périmées ou imprécises, pour les causes vues plus haut.

Un index fonctionnel possède ses propres statistiques : il faut donc penser à lancer `ANALYZE` après sa création. De même après un `CREATE STATISTICS`.

Nombre de lignes trouvées dans l'index :

Il faut se rappeler que PostgreSQL aura tendance à ne pas utiliser un index s'il doit chercher trop de lignes (ou plutôt de blocs), dans l'index comme dans la table ensuite. Il sera par contre tenté si cet index permet d'éviter des tris ou s'il est couvrant, et pas trop gros. La dispersion des lignes rencontrées dans la table est un facteur également pris en compte.

Colonnes d'un index B-tree :

Un index B-tree multicolonne est inutilisable, en tout cas beaucoup moins performant, si les premiers champs ne sont pas fournis. L'ordre des colonnes a son importance.

Taille d'un index :

PostgreSQL tient compte de la taille des index. Un petit index peut être préféré à un index multicolonne auquel on a ajouté trop de champs pour qu'il soit couvrant.

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 assez représentatif et démontre bien le problème :

```
SELECT * FROM employes WHERE date_embauche + interval '1 month' = '2006-01-01';
```

Avec une telle construction, l'optimiseur ne saura pas tirer partie d'un quelconque index, à moins d'avoir créé un index fonctionnel sur `date_embauche + interval '1 month'`, mais cet index est largement contre-productif par rapport à une réécriture de la requête.

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

```
SELECT * FROM employes WHERE date_trunc('month', date_embauche) = 12;
```

ou encore plus fréquemment rencontré :

```
SELECT * FROM employes WHERE extract('year' from date_embauche) = 2006;
SELECT * FROM employes WHERE upper(prenom) = 'GASTON';
```

Opérateurs non-supportés :

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires (entiers, chaînes, dates, mais pas les types composés comme les 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 :

```
EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..8264.74 rows=17 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=7 width=41)
         Filter: (num_service <> 4)
```

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

```
CREATE INDEX ON employes_big(num_service) WHERE num_service<>4;

EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using employes_big_num_service_idx1 on employes_big
   (cost=0.14..12.35 rows=17 width=40)
```

Paramétrage de PostgreSQL

Plusieurs paramètres de PostgreSQL influencent l'optimiseur sur l'utilisation 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 sont 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 voire 2 ou 1.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[11]

4.9.9 Écriture du SQL

- `NOT IN` avec une sous-requête
  - remplacer par `NOT EXISTS`
- `UNION` entraîne un tri systématique
  - préférer `UNION ALL`
- 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 performante lorsqu'elle est utilisée avec une sous-requête. L'optimiseur ne parvient pas à exécuter ce type de requête efficacement.

```
SELECT *
  FROM services
 WHERE num_service NOT IN (SELECT num_service FROM employes_big);
```

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

```
SELECT *
  FROM services s
 WHERE NOT EXISTS (SELECT 1
                   FROM employes_big e
                   WHERE s.num_service = e.num_service);
```

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

4.9.10 Absence de hints

- Certains regrettent l'absence de *hints*
- C'est la politique du projet :
  - vouloir ne signifie pas avoir besoin
  - PostgreSQL est un projet libre qui a le luxe de se défaire de la pression du marché
  - cela permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur
- Ne pensez pas être plus intelligent que le planificateur
- Mais il ne peut faire qu'avec ce qu'il a

L'absence de la notion de *hints*, qui permettent au DBA de forcer l'optimiseur à choisir des plans d'exécution jugés pourtant trop coûteux, est voulue. Elle a même été intégrée dans la liste des fonctionnalités dont la communauté ne voulait pas (« *Features We Do Not Want*[12] »).

L'absence des *hints* est très bien expliquée dans un billet de Josh Berkus, ancien membre de la Core Team de PostgreSQL[13] :

Le fait que certains DBA demandent cette fonctionnalité ne veut pas dire qu'ils ont réellement besoin de cette fonctionnalité. Parfois ce sont de mauvaises habitudes d'une époque révolue, où les optimiseurs étaient parfaitement stupides. Ajoutons à cela que les SGBD courants étant des projets commerciaux, ils sont forcément plus poussés à accéder aux demandes des clients, même si ces demandes ne se justifient pas, ou sont le résultat de pressions de pur court terme. Le fait que PostgreSQL soit un projet libre permet justement aux développeurs du projet de choisir les fonctionnalités implémentées suivant leurs idées, et non pas la pression du marché.

Selon le wiki sur le sujet[14], l'avis de la communauté PostgreSQL est que les *hints*, du moins tels qu'ils sont implémentés ailleurs, mènent à une plus grande complexité du code applicatif, donc à des problèmes de maintenabilité, 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. L'absence de *hints* permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur. Sur le long terme, cela est meilleur pour le projet comme pour les utilisateurs. Cela a notamment mené à améliorer l'optimiseur et le recueil des statistiques.

L'accumulation de *hints* dans un système a tendance à poser problème lors de l'évolution des besoins, de la volumétrie ou après des mises à jour. Si le plan d'exécution généré n'est pas optimal, il est préférable de chercher à comprendre d'où vient l'erreur. Il est rare que l'optimiseur se trompe : en général c'est lui qui a raison. Mais il ne peut faire qu'avec les statistiques à sa disposition, le modèle qu'il voit,

[12] https://wiki.postgresql.org/wiki/Todo#Features_We_Do_Not_Want
[13] https://it.toolbox.com/blogs/josh-berkus/why-postgresql-doesnt-have-query-hints-020411
[14] https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion

les index que vous avez créés. Nous avons vu dans ce module quelles pouvaient être les causes entraînant des erreurs de plan :

- mauvaise écriture de requête ;
- modèle de données pas optimal ;
- manque d'index adéquats (et PostgreSQL en possède une grande variété) ;
- statistiques pas à jour ;
- statistiques pas assez fines ;
- colonnes corrélées ;
- paramétrage de la mémoire ;
- paramétrage de la profondeur de recherche de l'optimiseur ;
- …

Ajoutons qu'il existe des outils comme PoWA[15] pour vous aider à optimiser des requêtes.

[15] https://powa.readthedocs.io/en/latest/

4.10 OUTILS D'OPTIMISATION

- auto_explain
- plantuner
- HypoPG

4.10.1 auto_explain

- Tracer les plans des requêtes lentes automatiquement
- Contrib officielle
- Mise en place globale (traces) :
  - globale :

```
shared_preload_libraries='auto_explain'   -- redémarrage !
ALTER DATABASE erp SET auto_explain.log_min_duration = '3s' ;
```

  - session :

```
LOAD 'auto_explain' ;
SET auto_explain.log_analyze TO true;
```

L'outil `auto_explain` est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à certains moments, et qu'on suspecte que son plan diffère entre deux exécutions. Elle permet de tracer dans les journaux applicatifs, voire dans la console, le plan de la requête dès qu'elle dépasse une durée configurée.

C'est une « contrib » officielle de PostgreSQL (et non une extension). Tracer systématiquement le plan d'exécution d'une requête souvent répétée prend de la place, et est assez coûteux. C'est donc un outil à utiliser parcimonieusement. En général on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre `auto_explain.log_min_duration`. Par défaut, ce paramètre vaut -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement, un `SET auto_explain.log_analyze TO true;` ou un `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment : `auto_explain.log_buffers`, `auto_explain.log_settings`.

Quant à `auto_explain.sample_rate`, il permet de ne tracer qu'un échantillon des requêtes (voir la documentation[16]).

Pour utiliser `auto_explain` globalement, il faut charger la bibliothèque au démarrage dans le fichier `postgresql.conf` via le paramètre `shared_preload_libraries`.

```
shared_preload_libraries='auto_explain'
```

Après un redémarrage de l'instance, il est possible de configurer les paramètres de capture des plans d'exécution par base de données. Dans l'exemple ci-dessous, l'ensemble des requêtes sont tracées sur la base de données `bench`, qui est utilisée par `pgbench`.

```
ALTER DATABASE bench SET auto_explain.log_min_duration = '0';
ALTER DATABASE bench SET auto_explain.log_analyze = true;
```

Attention, l'activation des traces complètes sur une base de données avec un fort volume de requêtes peut être très coûteux.

Un benchmark `pgbench` est lancé sur la base de données `bench` avec 1 client qui exécute 1 transaction par seconde pendant 20 secondes :

```
pgbench -c1 -R1 -T20 bench
```

Les plans d'exécution de l'ensemble les requêtes exécutées par `pgbench` sont alors tracés dans les traces de l'instance.

```
2021-07-01 13:12:55.790 CEST [1705] LOG:  duration: 0.041 ms  plan:
    Query Text: SELECT abalance FROM pgbench_accounts WHERE aid = 416925;
    Index Scan using pgbench_accounts_pkey on pgbench_accounts
        (cost=0.42..8.44 rows=1 width=4) (actual time=0.030..0.032 rows=1 loops=1)
        Index Cond: (aid = 416925)
2021-07-01 13:12:55.791 CEST [1705] LOG:  duration: 0.123 ms  plan:
    Query Text: UPDATE pgbench_tellers SET tbalance = tbalance + -3201 WHERE tid = 19;
    Update on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                               (actual time=0.120..0.121 rows=0 loops=1)
        ->  Seq Scan on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                                         (actual time=0.040..0.058 rows=1 loops=1)
            Filter: (tid = 19)
            Rows Removed by Filter: 99
2021-07-01 13:12:55.797 CEST [1705] LOG:  duration: 0.116 ms  plan:
    Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -3201 WHERE bid = 5;
    Update on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.112..0.114 rows=0 loops=1)
        ->  Seq Scan on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.036..0.038 rows=1 loops=1)
            Filter: (bid = 5)
            Rows Removed by Filter: 9
[...]
```

[16] https://docs.postgresql.fr/current/auto-explain.html

Pour utiliser `auto_explain` uniquement dans la session en cours, il faut penser à descendre au niveau de message `LOG` (défaut de `auto_explain`). On procède ainsi :

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = true;
SET client_min_messages to log;
SELECT count(*)
   FROM pg_class, pg_index
   WHERE oid = indrelid AND indisunique;

LOG:  duration: 1.273 ms  plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
Aggregate  (cost=38.50..38.51 rows=1 width=8)
    (actual time=1.247..1.248 rows=1 loops=1)
  ->  Hash Join  (cost=29.05..38.00 rows=201 width=0)
         (actual time=0.847..1.188 rows=198 loops=1)
        Hash Cond: (pg_index.indrelid = pg_class.oid)
        ->  Seq Scan on pg_index  (cost=0.00..8.42 rows=201 width=4)
                      (actual time=0.028..0.188 rows=198 loops=1)
              Filter: indisunique
              Rows Removed by Filter: 44
        ->  Hash  (cost=21.80..21.80 rows=580 width=4)
           (actual time=0.726..0.727 rows=579 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 29kB
              ->  Seq Scan on pg_class  (cost=0.00..21.80 rows=580 width=4)
                            (actual time=0.016..0.373 rows=579 loops=1)
 count
-------
   198
```

`auto_explain` est aussi un moyen de suivre les plans au sein de fonctions. Par défaut, un plan n'indique les compteurs de blocs *hit*, *read*, *temp*… que de l'appel global à la fonction.

Une fonction simple en PL/pgSQL est définie pour récupérer le solde le plus élevé dans la table `pgbench_accounts` :

```
CREATE OR REPLACE function f_max_balance() RETURNS int AS $$
    DECLARE
        acct_balance int;
    BEGIN
        SELECT max(abalance)
        INTO acct_balance
        FROM pgbench_accounts;
        RETURN acct_balance;
    END;
$$ LANGUAGE plpgsql ;
```

Un simple `EXPLAIN ANALYZE` de l'appel de la fonction ne permet pas d'obtenir le plan de la requête `SELECT max(abalance) FROM pgbench_accounts` contenue dans la fonction :

```
EXPLAIN (ANALYZE,VERBOSE) SELECT f_max_balance();
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=4) (actual time=49.214..49.216 rows=1 loops=1)
   Output: f_max_balance()
 Planning Time: 0.149 ms
 Execution Time: 49.326 ms
```

Par défaut, `auto_explain` ne va pas capturer plus d'information que la commande `EXPLAIN ANALYZE`. Le fichier log de l'instance capture le même plan lorsque la fonction est exécutée.

```
2021-07-01 15:39:05.967 CEST [2768] LOG:  duration: 42.937 ms  plan:
    Query Text: select f_max_balance();
    Result  (cost=0.00..0.26 rows=1 width=4)
            (actual time=42.927..42.928 rows=1 loops=1)
```

Il est cependant possible d'activer le paramètre `log_nested_statements` avant l'appel de la fonction, de préférence uniquement dans la ou les sessions concernées :

```
\c bench
SET auto_explain.log_nested_statements = true;
SELECT f_max_balance();
```

Le plan d'exécution de la requête SQL est alors visible dans les traces de l'instance :

```
2021-07-01 14:58:40.189 CEST [2202] LOG:  duration: 58.938 ms  plan:
Query Text: select max(abalance)
            from pgbench_accounts
Finalize Aggregate
(cost=22632.85..22632.86 rows=1 width=4)
(actual time=58.252..58.935 rows=1 loops=1)
    ->  Gather
        (cost=22632.64..22632.85 rows=2 width=4)
        (actual time=57.856..58.928 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate
                (cost=21632.64..21632.65 rows=1 width=4)
                (actual time=51.846..51.847 rows=1 loops=3)
            ->  Parallel Seq Scan on pgbench_accounts
                    (cost=0.00..20589.51 rows=417251 width=4)
                    (actual time=0.014..29.379 rows=333333 loops=3)
```

pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[17] pour une version plus lisible.

[17] https://explain.depesz.com/

4.10.2 Extension plantuner

- Pour :
  - interdire certains index
  - forcer à zéro les statistiques d'une table vide
- Intéressant en développement pour tester les plans
  - pas en production !

Cette extension est disponible à cette adresse[18] (le miroir GitHub ne semble pas maintenu). Oleg Bartunov, l'un de ses auteurs, a publié en 2018 un article intéressant[19] sur son utilisation.

Il faudra récupérer le source et le compiler. La configuration est basée sur trois paramètres :

- `plantuner.enable_index` pour préciser les index à activer ;
- `plantuner.disable_index` pour préciser les index à désactiver ;
- `plantuner.fix_empty_table` pour forcer à zéro les statistiques des tables de 0 bloc.

Ils sont configurables à chaud, comme le montre l'exemple suivant :

```
LOAD 'plantuner';
EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
   Index Cond: (date_embauche = '1000-01-01'::date)

SET plantuner.disable_index='employes_big_date_embauche_idx';

EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                      QUERY PLAN
------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
         Filter: (date_embauche = '1000-01-01'::date)
```

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.

[18] http://www.sai.msu.su/~megera/wiki/plantuner
[19] https://obartunov.livejournal.com/197604.html

Cependant, généralement, cette extension a sa place sur un serveur de développement pour bien comprendre les choix de planification, pas sur un serveur de production. En tout cas, pas dans le but de tromper le planificateur.

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur.

4.10.3 Extension pg_plan_hint

- Pour :
  - forcer l'utilisation d'un nœud entre deux tables
  - imposer une valeur de paramètre
  - appliquer automatiquement ces *hints* à des requêtes

Cette extension existe depuis longtemps. Elle doit être compilée et installée depuis le dépôt Github[20].

La documentation[21] en anglais peut être complétée par la version japonaise[22] plus à jour, ou cet article[23].

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur !

4.10.4 Extension HypoPG

- Extension PostgreSQL
- Création d'index hypothétiques pour tester leur intérêt
  - avant de les créer pour de vrai
- Limitations : surtout B-Tree, statistiques

[20] https://github.com/ossc-db/pg_hint_plan
[21] http://pghintplan.osdn.jp/pg_hint_plan.html
[22] http://pghintplan.osdn.jp/pg_hint_plan-ja.html
[23] https://docs.yugabyte.com/latest/explore/query-1-performance/pg-hint-plan/

Cette extension est disponible sur GitHub[24] et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

- `hypopg_create_index()` pour créer un index hypothétique ;
- `hypopg_drop_index()` pour supprimer un index hypothétique particulier ou `hypopg_reset()` pour tous les supprimer ;
- `hypopg_list_indexes` pour les lister.

Un index hypothétique n'existe que dans la session, ni en mémoire ni sur le disque, mais le planificateur le prendra en compte dans un `EXPLAIN` simple (évidemment pas un `EXPLAIN ANALYZE`). En quittant la session, tous les index hypothétiques restants et créés sur cette session sont supprimés.

L'exemple suivant est basé sur la base dont le script peut être téléchargé sur https://dali.bo/tp_employes_services.

```
CREATE EXTENSION hypopg;

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_create_index('CREATE INDEX ON employes_big(prenom)');

 indexrelid |            indexname
------------+---------------------------------
      24591 | <24591>btree_employes_big_prenom

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using <24591>btree_employes_big_prenom on employes_big
                               (cost=0.05..4.07 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_list_indexes;

 indexrelid |            indexname             | nspname |   relname    | amname
------------+----------------------------------+---------+--------------+--------
      24591 | <24591>btree_employes_big_prenom | public  | employes_big | btree

SELECT * FROM hypopg_reset();

 hypopg_reset
--------------

(1 row)
```

[24] https://github.com/HypoPG/hypopg

```
CREATE INDEX ON employes_big(prenom);

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using employes_big_prenom_idx on employes_big
                        (cost=0.42..4.44 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

Le cas idéal d'utilisation est l'index B-Tree sur une colonne. Un index fonctionnel est possible, mais, faute de statistiques disponibles avant la création réelle de l'index, les estimations peuvent être fausses. Les autres types d'index sont moins bien ou non supportées.

4.11 CONCLUSION

- Planificateur très avancé
- Ne pensez pas être plus intelligent que lui
- Il faut bien comprendre son fonctionnement

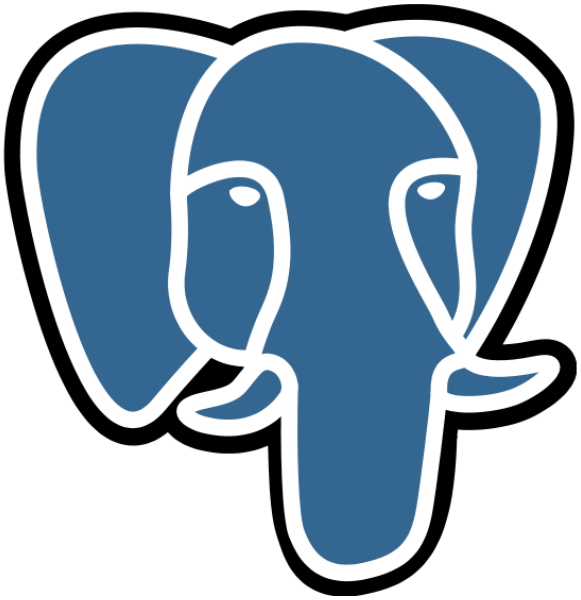
4.11.1 Questions

N'hésitez pas, c'est le moment !

4.12 QUIZ

https://dali.bo/j2_quiz

5/ Référence sur les nœuds d’exécution



5.1 INTRODUCTION

- 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.

Cet annexe a pour but d'entrer dans le détail de chaque type de nœuds, ses avantages et inconvénients.

5.2 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

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.

5.2.1 Parcours de table

- Parcours séquentiel de la table (*Sequential Scan* ou *Seq Scan*)
  - parallélisation possible (*Parallel Seq Scan*)
- Aussi appelé *Full table scan* par d'autres SGBD
- La table est lue entièrement
  - même si seulement quelques lignes satisfont la requête
  - sauf pour `LIMIT` sans `ORDER BY`
- 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 séquen-

tielle, 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.

Voici quelques exemples à partir de ce jeu de tests :

```
CREATE TABLE t1 (c1 integer);
INSERT INTO t1 (c1) SELECT generate_series(1, 100000);
ANALYZE t1;
```

Ici, nous faisons une lecture complète de la table. De ce fait, un parcours séquentiel sera plus rapide du fait de la rapidité de la lecture séquentielle des blocs :

```
EXPLAIN SELECT * FROM t1 ;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=4)
```

Le coût est relatif au nombre de blocs lus, au nombre de lignes décodées et à la valeur des paramètres `seq_page_cost` et `cpu_tuple_cost`. Si un filtre est ajouté, cela aura un coût supplémentaire dû à l'application du filtre sur toutes les lignes de la table (pour trouver celles qui correspondent à ce filtre) :

```
EXPLAIN SELECT * FROM t1 WHERE c1=1000 ;

                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1693.00 rows=1 width=4)
   Filter: (c1 = 1000)
```

Ce coût supplémentaire dépend du nombre de lignes dans la table et de la valeur du paramètre `cpu_operator_cost` (défaut 0,0025) ou de la valeur du paramètre `COST` de la fonction appelée. L'exemple ci-dessus montre le coût (1693) en utilisant l'opérateur standard d'égalité. Maintenant, si on crée une fonction qui utilise cet opérateur (mais écrite en PL/pgSQL, cela reste invisible pour PostgreSQL), avec un coût forcé à 10 000, cela donne :

```
CREATE FUNCTION egal(integer,integer) RETURNS boolean LANGUAGE plpgsql AS $$
begin
return $1 = $2;
end
$$
COST 10000;

EXPLAIN SELECT * FROM t1 WHERE egal(c1, 1000) ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..2501443.00 rows=33333 width=4)
   Filter: egal(c1, 1000)
```

La ligne *Filter* indique le filtre réalisé. Le nombre de lignes indiqué par `rows=` est le nombre de lignes après filtrage. Pour savoir combien de lignes ne satisfont pas le prédicat de la clause `WHERE`, il faut exécuter la requête et donc utiliser l'option `EXPLAIN` :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM t1 WHERE c1=1000 ;

                      QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..1693.00 rows=1 width=4)
          (actual time=0.236..19.615 rows=1 loops=1)
   Filter: (c1 = 1000)
   Rows Removed by Filter: 99999
   Buffers: shared hit=443
 Planning time: 0.110 ms
 Execution time: 19.649 ms
```

Il s'agit de la ligne `Rows Removed by Filter`.

L'option `BUFFERS` permet en plus de savoir le nombre de blocs lus dans le cache et hors du cache.

Le calcul réalisé pour le coût final est le suivant :

```
SELECT
  round((
    current_setting('seq_page_cost')::numeric*relpages      +
    current_setting('cpu_tuple_cost')::numeric*reltuples    +
    current_setting('cpu_operator_cost')::numeric*reltuples
        )::numeric, 2)
  AS cout_final
FROM pg_class
WHERE relname='employes';
```

Si le paramètre `synchronize_seqscans` est activé (et il l'est par défaut), 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 parcours 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 tenter de 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`.

Une nouvelle optimisation vient de la parallélisation. Depuis la version 9.6, il est possible d'obtenir un parcours de table parallélisé. Dans ce cas, le nœud s'appelle un *Parallel Seq Scan*. Le processus responsable de la requête demande l'exécution de plusieurs processus, appelés des *workers* qui auront tous pour charge de lire la table et d'appliquer le filtre. Chaque *worker* travaillera sur des blocs différents. Le prochain bloc à lire est enregistré en mémoire partagée. Quand un *worker* a terminé de travailler sur un bloc, il consulte la mémoire partagée pour connaître le prochain bloc à traiter, et incrémente ce numéro pour que le *worker* suivant puisse travailler sur un autre bloc. Il n'y a aucune assurance que chaque *worker* travaillera sur le même nombre de blocs. Voici un exemple de plan parallélisé pour un parcours de table :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT sum(c2) FROM t1 WHERE c1 BETWEEN 100000 AND 600000 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate  (cost=12196.94..12196.95 rows=1 width=8)
                    (actual time=91.886..91.886 rows=1 loops=1)
   Buffers: shared hit=1277
   ->  Gather  (cost=12196.73..12196.94 rows=2 width=8)
             (actual time=91.874..91.880 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=1277
         ->  Partial Aggregate  (cost=11196.73..11196.74 rows=1 width=8)
                           (actual time=83.760..83.760 rows=1 loops=3)
               Buffers: shared hit=4425
               ->  Parallel Seq Scan on t1
                             (cost=0.00..10675.00 rows=208692 width=4)
                      (actual time=12.727..62.608 rows=166667 loops=3)
                     Filter: ((c1 >= 100000) AND (c1 <= 600000))
                     Rows Removed by Filter: 166666
                     Buffers: shared hit=4425
 Planning time: 0.528 ms
 Execution time: 94.877 ms
```

Dans ce cas, le planificateur a prévu l'exécution de deux *workers*, et deux ont bien été lancés lors de l'exécution de la requête.

5.2.2 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)
- Gros gain en performance si filtre très sélectif
- Les lignes renvoyées sont triées
- Parallélisation possible
  - B-Tree uniquement
- Sur d'autres SGBD : `INDEX RANGE SCAN` + `TABLE ACCESS BY INDEX ROWID`

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 4 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 :

```
CREATE TABLE t1 (c1 integer, c2 integer);
INSERT INTO t1 VALUES (1,2), (2,4), (3,6);
CREATE INDEX i1 ON t1(c1);
ANALYZE t1;
```

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

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1=2;
```

```
                  QUERY PLAN
-------------------------------------------------
 Seq Scan on t1  (cost=0.00..1.04 rows=1 width=8)
        (actual time=0.020..0.023 rows=1 loops=1)
   Filter: (c1 = 2)
   Rows Removed by Filter: 2
   Buffers: shared hit=1
 Planning time: 0.163 ms
 Execution time: 0.065 ms
```

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 coûterait forcément plus cher. Grâce à l'option `BUFFERS`, nous savons que seul un bloc a été lu.

Pour faire un parcours d'index, nous allons désactiver les parcours séquentiels et réinitialiser les statistiques :

```
SET enable_seqscan TO off;
```

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

Maintenant relançons la requête :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1=2;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using i1 on t1  (cost=0.13..8.15 rows=1 width=8)
                   (actual time=0.117..0.121 rows=1 loops=1)
   Index Cond: (c1 = 2)
   Buffers: shared hit=1 read=1
 Planning time: 0.174 ms
 Execution time: 0.174 ms
```

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

```
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 |             1 |            0
```

Une page disque a été lue dans l'index (colonne `idx_blks_read` 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*.

5.2.3 Parcours d'index bitmap

- *Bitmap Index Scan* / *Bitmap Heap Scan*
- Réduire les allers-retours index <-> table
  - trouver les blocs de l'index
  - lecture des blocs intéressantde la table
- Combiner plusieurs index en mémoire
  - nœud *BitmapAnd*
  - nœud *BitmapOr*
- Coût de démarrage généralement important (pas intéressant avec `LIMIT`)
- Parallélisation possible
- B-Tree uniquement
- Sensible à :
  - `effective_io_concurrency`

Principe :

D'autres SGBD connaissent les index bitmap, mais sous PostgreSQL, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Il se manifeste par le couple de noeuds *Bitmap Index Scan* et *Bitmap Heap Scan*.

Le principe est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table. Même avec un SSD, il évite d'aller chercher trop souvent les mêmes blocs et améliore l'utilisation du cache. Son principe est le suivant :

- lecture en une passe de l'index (*Bitmap Index Scan*) ;
- récupération des TID (*tuple id*) en mémoire (1 bit par ligne dans le cas idéal) ;
- tri des blocs à parcourir dans la table dans l'ordre physique de la table (pas dans l'ordre logique de l'index) ;
- lecture en une passe de la partie intéressante de la table (*Bitmap Heap Scan*).

Un bitmap est souvent utilisé quand il y a un grand nombre de valeurs à filtrer, notamment pour les clauses `IN` et `ANY`.

Ce type d'index présente un autre gros intérêt : pouvoir combiner plusieurs index en mémoire. Les bitmaps de TID obtenus se combinent facilement avec des opérations booléennes `AND` et `OR`.

Exemple :

Cet exemple utilise PostgreSQL 15 dans sa configuration par défaut. La table suivante possède trois champs indexés susceptibles de servir de critère de recherche :

```
CREATE UNLOGGED TABLE tbt
(i int GENERATED ALWAYS AS IDENTITY PRIMARY KEY, j int, k int, t text)  ;

INSERT INTO tbt (j,k,t)
SELECT (i / 1000) , i / 777, chr (64+ (i % 58))
FROM generate_series(1,10000000) i ;

CREATE INDEX tbt_j_idx ON tbt (j) ;
CREATE INDEX tbt_k_idx ON tbt (k) ;
CREATE INDEX tbt_t_idx ON tbt (t) ;

VACUUM ANALYZE tbt ;
```

Lors de la recherche sur les plusieurs critères, les lignes renvoyées par les *Bitmap Index Scan* peuvent être combinées :

```
-- pour la lisibilité des plans
SET max_parallel_workers_per_gather  TO 0 ;
SET jit TO off ;

EXPLAIN (ANALYZE, BUFFERS, VERBOSE, SETTINGS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
AND k = 10
AND t = 'a';
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on public.tbt  (cost=23.02..27.04 rows=1 width=14) (actual
 ↪  time=0.598..0.702 rows=9 loops=1)
```

```
   Output: i, j, k, t
   Recheck Cond: ((tbt.k = 10) AND (tbt.j = 8))
   Filter: (tbt.t = 'a'::text)
   Rows Removed by Filter: 538
   Heap Blocks: exact=4
   Buffers: shared read=11
   ->  BitmapAnd  (cost=23.02..23.02 rows=1 width=0) (actual time=0.557..0.558
↪  rows=0 loops=1)
         Buffers: shared read=7
         ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.62 rows=824 width=0)
↪  (actual time=0.501..0.501 rows=777 loops=1)
               Index Cond: (tbt.k = 10)
               Buffers: shared read=4
         ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.15 rows=1029 width=0)
↪  (actual time=0.053..0.053 rows=1000 loops=1)
               Index Cond: (tbt.j = 8)
               Buffers: shared read=3
 Settings: jit = 'off', max_parallel_workers_per_gather = '0'
 Planning Time: 0.114 ms
 Execution Time: 0.740 ms
```

Dans le plan précédent[1] :

- deux *Bitmap Index Scan* parcourent séparément deux index, qui remontent l'un 777 lignes (toutes les lignes de la table où `k` vaut 10), l'autre 1000 lignes (où `j` vaut 8) ;
- le troisième index sur `t` est ignoré : il y a trop de lignes avec cette valeur (un décompte en trouverait 172 414), et surtout dispersées dans toute la table ;
- la combinaison des lignes remontées désigne seulement 4 blocs dans la table possédant des lignes correspondant aux deux critères (mention *Heap Blocks*) ;
- le *Bitmap Heap Scan* lit ces 4 blocs séquentiellement, et donc une seule fois chacun, et trouve 547 lignes ;
- la clause *Recheck* vérifie que ces lignes sont réellement visibles (ici rien n'est rejeté) ;
- il reste à appliquer le critère non géré par les index utilisés, soit `t = 'a'` : c'est le rôle de la clause *Filter*, qui écarte 538 lignes et n'en garde que 9.

Le coût de démarrage est généralement important à cause de la lecture préalable de l'index et du tri des TID. Ce type de parcours est donc moins intéressant quand on recherche un coût de démarrage faible (clause `LIMIT`, curseur…). Un parcours d'index simple sera généralement choisi dans ce cas.

Clause OR :

Les index sont également utiles avec une clause `OR` :

```
EXPLAIN (ANALYZE,BUFFERS, COSTS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
OR k = 10
OR t = 'a';
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
```

[1] https://explain.dalibo.com/plan/54bc5g7b3c9h76a7

```
 Bitmap Heap Scan on tbt  (cost=2039.91..59155.01 rows=174831 width=14) (actual
↪  time=27.860..385.638 rows=173623 loops=1)
   Recheck Cond: ((j = 8) OR (k = 10) OR (t = 'a'::text))
   Heap Blocks: exact=54054
   Buffers: shared hit=10 read=54199
   ->  BitmapOr  (cost=2039.91..2039.91 rows=174863 width=0) (actual
↪  time=12.514..12.515 rows=0 loops=1)
         Buffers: shared hit=4 read=151
         ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.18 rows=1033 width=0)
↪  (actual time=0.049..0.049 rows=1000 loops=1)
               Index Cond: (j = 8)
               Buffers: shared read=3
         ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.68 rows=833 width=0)
↪  (actual time=0.028..0.028 rows=777 loops=1)
               Index Cond: (k = 10)
               Buffers: shared hit=4
        ->  Bitmap Index Scan on tbt_t_idx  (cost=0.00..1885.92 rows=172998 width=0)
↪  (actual time=12.435..12.435 rows=172414 loops=1)
               Index Cond: (t = 'a'::text)
               Buffers: shared read=148
 Planning Time: 0.076 ms
 Execution Time: 394.014 ms
```

Ce plan[2] utilise cette fois les trois index. Au final, le *Bitmap Heap Scan* lit quand même toute la table ! En effet, il y a des `t='a'` dans tous les blocs (cas le plus défavorable). 98 % des comparaisons de critères sont tout de même évitées, et ce plan s'avère plus efficace qu'un parcours séquentiel, trois fois plus long sur la même machine.

Rôle du work_mem :

Si le `work_mem` est trop bas, PostgreSQL n'a plus la place de stocker un bit par ligne dans son tableau, mais utilise un bit par page. La mention *lossy* apparaît alors sur la ligne *Heap Blocks*, et toutes les lignes de la page doivent être vérifiées. Avec la requête précédente, la performance est cette fois pire qu'un parcours complet :

```
SET work_mem TO '256kB' ;

EXPLAIN (ANALYZE,BUFFERS, COSTS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
OR k = 10
OR t = 'a';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tbt  (cost=1955.42..224494.16 rows=167501 width=14) (actual
↪  time=8.987..1601.912 rows=173623 loops=1)
   Recheck Cond: ((j = 8) OR (k = 10) OR (t = 'a'::text))
   Rows Removed by Index Recheck: 9350021
   Heap Blocks: exact=2620 lossy=51434
   Buffers: shared read=54209
   ->  BitmapOr  (cost=1955.42..1955.42 rows=167532 width=0) (actual
↪  time=8.498..8.500 rows=0 loops=1)
```

[2] https://explain.dalibo.com/plan/a5ebd41930fcccde

```
        Buffers: shared read=155
        ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.19 rows=1034 width=0)
↪  (actual time=0.451..0.451 rows=1000 loops=1)
              Index Cond: (j = 8)
              Buffers: shared read=3
        ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.65 rows=828 width=0)
↪  (actual time=0.034..0.034 rows=777 loops=1)
              Index Cond: (k = 10)
              Buffers: shared read=4
      ->  Bitmap Index Scan on tbt_t_idx  (cost=0.00..1806.96 rows=165670 width=0)
↪  (actual time=8.011..8.011 rows=172414 loops=1)
              Index Cond: (t = 'a'::text)
              Buffers: shared read=148
Planning Time: 0.089 ms
Execution Time: 1610.028 ms
```

effective_io_concurrency :

Les parcours *Bitmap Heap Scan* sont sensibles au paramètre `effective_io_concurrency`, qu'il peut être très bénéfique d'augmenter. `effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[3], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[4].

Enfin, le paramètre `enable_bitmapscan` permet d'activer ou de désactiver l'utilisation des parcours d'index bitmap.

5.2.4 Parcours d'index seul

```
SELECT c1 FROM t1 WHERE c1<10
```

- Avant 9.2 : PostgreSQL devait lire l'index + la table
- À présent : le planificateur utilise la *Visibility Map*
  - nœud *Index Only Scan*
  - index B-Tree
  - index SP-GiST
  - index GiST => Types : point, box, inet, range

[3] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[4] https://docs.postgresql.fr/13/release.html

Voici un exemple sous PostgreSQL 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

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
```

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
```

```
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
```

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
```

```
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
```

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
```

- 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
```

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[5].

5.2.5 Parcours : autres

- *TID Scan*
- *Function Scan*
- *Values*
- *Result*

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

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

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. Il est possible de le désactiver via le paramètre `enable_tidscan`.

```
b1=# EXPLAIN SELECT * FROM pg_class WHERE ctid = '(1,1)';

                        QUERY PLAN
-----------------------------------------------------------
 Tid Scan on pg_class  (cost=0.00..4.01 rows=1 width=265)
   TID Cond: (ctid = '(1,1)'::tid)
```

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)
```

`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
(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)
```

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)
```

5.3 JOINTURES

- Prend 2 ensembles de données en entrée
  - *inner* (interne)
  - *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.

5.3.1 Nested Loops

Boucles imbriquées

- 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
  - 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)
```

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)
```

Le paramètre `enable_nestloop` permet d'activer ou de désactiver ce type de nœud.

5.3.2 Merge Join

Jointure d'ensembles triés

- Trier l'ensemble interne
- Trier l'ensemble externe
- Tant qu'il reste des lignes dans un des ensembles
  - lire les deux ensembles en parallèle
  - si la condition de jointure est avérée : émettre la ligne
- 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)
```

Le paramètre `enable_mergejoin` permet d' activer ou de désactiver ce type de nœud.

5.3.3 Hash Join

Jointure par hachage

- 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
- 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. Le paramétrage de `work_mem` et `hash_mem_multiplier` (à partir de la 13) peut donc avoir un gros impact. De même, diminuer le nombre de colonnes récupérées permet de diminuer la mémoire à utiliser pour le hachage, et donc 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)
```

Le paramètre `enable_hashjoin` permet d' activer ou de désactiver ce type de nœud.

5.3.4 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

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.

Par exemple, PostgreSQL 8.4 donnait ce plan :

```
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)
```

Et la même requête exécutée à partir de PostgreSQL 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)
```

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.

5.3.5 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 (`geqo_threshold`)
  - limite le temps de planification et l'utilisation de mémoire
  - `join_collapse_limit` , `from_collapse_limit` : limites de 8 tables

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)
```

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)
```

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 18 363 au lieu de 18 341 précédemment).

Pour des requêtes avec de très nombreuses tables (décisionnel…), pour ne pas avoir à réordonner les tables dans la clause `FROM`, on peut monter les valeurs de `join_collapse_limit` ou `from_collapse_limit` le temps de la session ou pour un couple utilisateur/base précis. Le faire au niveau global risque de faire exploser les temps de planification d'autres requêtes.

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. Il se déclenche lorsque le nombre de tables dans la clause `FROM` est supérieure ou égale à la valeur du paramètre `geqo_threshold`, qui vaut 12 par défaut.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe[6]). 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[7] pour approfondir ce point).

[6] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f
[7] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116279

5.4 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.

5.4.1 Append

- Prend plusieurs ensembles de données
- Sortie non triée
- Utilisation :
  - tables héritées (dont partitionnement)
  - `UNION ALL` et des `UNION`
  - NB : `UNION` sans `ALL` élimine les doublons (tri !)
- Opération parallélisable (v11)

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 d'un héritage ou une table partitionnée).

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)
```

Nouvel exemple avec un filtre sur la clé de partitionnement :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 off

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)
```

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

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)
```

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)
```

`UNION ALL` récupère toutes les lignes des deux ensembles de données, même en cas de doublon. Pour n'avoir que les lignes distinctes, il est possible d'utiliser `UNION` *sans* la clause `ALL` mais cela entraîne une déduplication des données, ce qui est souvent coûteux :

```
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)
```

L'utilisation involontaire de `UNION` au lieu de `UNION ALL` est un problème de performance très fréquent.

Le paramètre `enable_partition_pruning` permet d'activer l'élagage des partitions de la même manière que `constraint_exclusion` pour les tables implémentant l'héritage. Il peut prendre deux valeurs `on` ou `off`. Cette fonctionnalité est activée par défaut.

```
                                    Partitioned table "public.tpart"
 Column |  Type   | Collation | Nullable | Default | Storage  | Compression | Stats
 ↪  target | Description
--------+---------+-----------+----------+---------+----------+-------------+-------
 ↪  -------+-------------
 i      | integer |           |          |         | plain    |             |
 t      | text    |           |          |         | extended |             |
Partition key: RANGE (i)
Partitions: part1 FOR VALUES FROM (0) TO (100),
            part2 FOR VALUES FROM (100) TO (200),
            part3 FOR VALUES FROM (200) TO (300)
```

Avec l'élagage activé, on observe que seules les partitions dont la contrainte `CHECK` correspond au prédicat sont visitées. À la différence du partitionnement par héritage, la table mère n'est pas scannée car elle ne contient pas de données.

```
b2=# EXPLAIN SELECT * FROM tpart WHERE i > 100;
                               QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=0.00..5.50 rows=199 width=36)
   ->  Seq Scan on part2 tpart_1  (cost=0.00..2.25 rows=99 width=36)
```

```
         Filter: (i > 100)
   ->  Seq Scan on part3 tpart_2  (cost=0.00..2.25 rows=100 width=36)
         Filter: (i > 100)
(5 rows)
```

En désactivant l'élagage, toutes les partitions sont visitées.

```
b2=# SET enable_partition_pruning TO off;
b2=# EXPLAIN SELECT * FROM tpart WHERE i > 100;

                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=0.00..7.75 rows=200 width=36)
   ->  Seq Scan on part1 tpart_1  (cost=0.00..2.25 rows=1 width=36)
         Filter: (i > 100)
   ->  Seq Scan on part2 tpart_2  (cost=0.00..2.25 rows=99 width=36)
         Filter: (i > 100)
   ->  Seq Scan on part3 tpart_3  (cost=0.00..2.25 rows=100 width=36)
         Filter: (i > 100)
(7 rows)
```

À partir de la version 11, les fils d'un nœud *Append* sont parallélisables.

5.4.2 MergeAppend

- Append avec optimisation
- Sortie triée
- Utilisation :
  - `UNION ALL`, partitionnement/héritage
  - avec parcours triés
  - idéal avec `LIMIT`

Le nœud *MergeAppend* est une optimisation spécifiquement conçue pour le partitionnement. Elle permet de répondre plus efficacement aux requêtes effectuant un tri sur un `UNION ALL`, soit explicite, soit induit par héritage ou 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*, avec `LIMIT 5` :

```
                                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
```

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`.

5.5 AUTRES NŒUDS

- Nœud *HashSetOp Except*
  - `EXCEPT` et `EXCEPT ALL`
- Nœud *HashSetOp Intersect*
  - `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)
```

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)
```

5.5.1 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.

5.5.2 Tris

- Sort
- Incremental Sort

5.5.2.1 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 `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 B-Tree. 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
```

- 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
```

- 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
```

- 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
```

Les paramètres `enable_sort` et `enable_incremental_sort` permettent de défavoriser l'utilisation d'un tri, respectivement non incrémental ou incrémental. 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.

5.5.2.2 Incremental Sort

- Utilisé lorsqu'un index existe sur les premières colonnes du tri
  - `ORDER BY`, `DISTINCT`, `GROUP BY`, `UNION`
  - Les jointures de type *Merge Join*
- Délai de démarrage réduit

Lorsqu'un tri est réalisé sur plusieurs colonnes, si aucun index ne permet de réaliser un tri classique, PostgreSQL peut essayer d'utiliser un index existant sur une des premières colonnes du tri pour réaliser un pré-tri.

Il pourra alors réaliser un tri sur les colonnes suivantes en tirant parti des groupes établis avec l'index utilisé. Dans ce cas, le délai de démarrage est réduit ce qui peut améliorer les performances lorsque la requête contient une clause `LIMIT`.

```
=# EXPLAIN (ANALYZE, COSTS OFF) SELECT * FROM clients ORDER BY id, ddn;
                                QUERY PLAN
--------------------------------------------------------------------------------
 Incremental Sort (actual time=0.170..0.171 rows=0 loops=1)
   Sort Key: id, ddn
   Presorted Key: id
   Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak Memory:
↪  25kB
   ->  Index Scan using clients_pkey on clients (actual time=0.008..0.008 rows=0
↪  loops=1)
 Planning Time: 0.209 ms
 Execution Time: 0.214 ms
(7 rows)
```

Le `DISTINCT` est géré depuis la version 16.

5.5.3 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)
```

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)
```

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)
```

5.5.4 HashAggregate

- 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)
```

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. Cependant, la version 13 améliore cela en utilisant le disque à partir du moment où la mémoire nécessaire dépasse la multiplication de la valeur du paramètre `work_mem` et celle du paramètre `hash_mem_multiplier` (2 par défaut à partir de la version 15, 1 auparavant). La requête sera plus lente, mais la mémoire ne sera pas saturée.

Le paramètre `enable_hashagg` permet d'activer et de désactiver l'utilisation de ce type de nœud.

5.5.5 GroupAggregate

- 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)
```

Un parcours d'index est possible pour remplacer le parcours séquentiel et le tri.

5.5.6 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)
```

5.5.7 Limit

- Limiter le nombre de résultats renvoyés
- Utilisation :
  - `LIMIT` et `OFFSET` dans 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 (*Seq Scan*, *Nested Loop*)

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)
```

5.5.8 Memoize

- Apparu en version 14
- Cache de résultat
- Utilisable par la table interne des *Nested Loop*
- Utile si :
  - peu de valeurs distinctes dans l'ensemble interne
  - beaucoup de valeurs dans l'ensemble externe
  - peu de correspondance entre les deux ensembles
- Paramètres : `work_mem` `hash_mem_multiplier`

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop* généré par une jointure classique ou `LATERAL`.

Le cas idéal pour cette optimisation concerne des jointures où de large portions des lignes de l'ensemble interne de la jointure n'ont pas de correspondance dans l'ensemble externe. Dans ce genre de cas, un *Hash Join* serait moins efficace car il devrait calculer la clé de hachage de valeurs

qui ne seront jamais utilisées ; et le *Merge Join* devrait ignorer un grand nombre de lignes dans son parcours de la table interne.

L'intérêt du cache de résultat augmente lorsqu'il y a peu de valeurs distinctes dans l'ensemble interne et que le nombre de valeurs dans l'ensemble externe est grand, ce qui provoque beaucoup de boucles. Ce nœud est donc très sensible aux statistiques sur le nombre de valeurs distinctes (`ndistinct`).

Cette fonctionnalité utilise une table de hashage pour stocker les résultats. Cette table est dimensionnée grâce aux paramètres `work_mem` et `hash_mem_multiplier` . Si le cache se remplit, les valeurs les plus anciennes sont exclues du cache.

Exemple :

```
CREATE TABLE t1(i int, j int);
CREATE TABLE t2(k int, l int);
INSERT INTO t2 SELECT x % 20,x FROM generate_series(1, 3000000) AS F(x);
INSERT INTO t1 SELECT x,x FROM generate_series(1, 300000) AS F(x);
CREATE INDEX ON t1(j);
ANALYZE t1,t2;

EXPLAIN (TIMING off, COSTS off, SUMMARy off, ANALYZE)
  SELECT * FROM t1 INNER JOIN t2 ON t1.j = t2.k;

                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop (actual rows=950 loops=1)
   ->  Seq Scan on t2 (actual rows=1000 loops=1)
   ->  Memoize (actual rows=1 loops=1000)
         Cache Key: t2.k
         Cache Mode: logical
         Hits: 980  Misses: 20  Evictions: 0  Overflows: 0  Memory Usage: 3kB
         ->  Index Scan using t1_j_idx on t1 (actual rows=1 loops=20)
               Index Cond: (j = t2.k)
```

On voit ici que le cache fait 3 ko. Il a permis de stocker 20 valeurs et de faire 980 accès au cache sur 1000 accès au total. Aucune valeur n'a été exclue du cache.

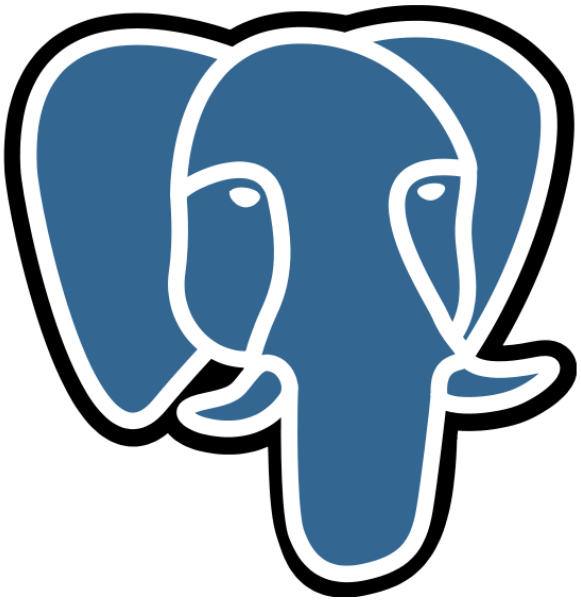
En désactivant ce nœud, on bascule sur un *Hash Join*:

```
SET enable_memoize TO off;
EXPLAIN (TIMING off, COSTS off, SUMMARY off, ANALYZE)
  SELECT * FROM t1 INNER JOIN t2 ON t1.j = t2.k;

                     QUERY PLAN
-----------------------------------------------------
 Hash Join (actual rows=950 loops=1)
   Hash Cond: (t2.k = t1.j)
   ->  Seq Scan on t2 (actual rows=1000 loops=1)
   ->  Hash (actual rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 48kB
         ->  Seq Scan on t1 (actual rows=1000 loops=1)
```

Dans ce petit exemple, le gain est nul, mais pour de grosses jointures, il peut être conséquent.

6/ Analyses et diagnostics



6.1 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).

6.1.1 Menu

- Supervision occasionnelle système
  - Linux
  - Windows
- Supervision occasionnelle PostgreSQL
- Outils

6.2 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.

6.2.1 Unix - ps

- `ps` est l'outil de base pour les processus
- Exemples
  - `ps aux`
  - `ps f -f -u 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 -u postgres f
10149 pts/5 S  0:00  \_ postmaster
10165 ?     Ss 0:00  |   \_ postgres: checkpointer
10166 ?     Ss 0:00  |   \_ postgres: background writer
10168 ?     Ss 0:00  |   \_ postgres: wal writer
10169 ?     Ss 0:00  |   \_ postgres: autovacuum launcher
10171 ?     Ss 0:00  |   \_ postgres: 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
500 10166  0.0  0.0 294624  5120 Ss   postgres: background writer
```

```
500 10168  0.0  0.0 294624  8680 Ss   postgres: wal writer
500 10169  0.0  0.0 295056  5976 Ss   postgres: autovacuum launcher
500 10171  0.0  0.0 294916  4004 Ss   postgres: logical replication launcher
[...]
```

Attention à la colonne `RSS`. Elle indique la quantité de mémoire utilisée par chaque processus, en prenant aussi en compte la mémoire partagée lue par le processus. Il peut donc arriver qu'en additionnant les valeurs de cette colonne, on arrive à une valeur bien plus importante que la mémoire physique, ce qui est donc normal. La valeur de la colonne `VSZ` comprend toujours l'intrégralité de la mémoire partagée allouée initialement par le processus postmaster.

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
500 9137  0.0  0.0 194156  1372 Ss   \_ postgres: background writer
500 9138  0.0  0.0 194156  1104 Ss   \_ postgres: wal writer
500 9139  0.0  0.0 194992  2360 Ss   \_ postgres: autovacuum launcher
500 9141  0.0  0.0 194156  1372 Ss   \_ postgres: logical replication launcher
```

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 -u postgres f
4563 pts/0    S      0:00  \_ postmaster
4569 ?        Ss     0:00  |   \_ postgres: checkpointer
4570 ?        Ss     0:00  |   \_ postgres: background writer
4571 ?        Ds     0:00  |   \_ postgres: wal writer
4572 ?        Ss     0:00  |   \_ postgres: autovacuum launcher
4574 ?        Ss     0:00  |   \_ postgres: 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 réelle 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.

6.2.2 Unix - top

- Principal intérêt : `%CPU` et `%MEM`
- Intérêts secondaires
  - charge `CPU`
  - consommation mémoire
- Autres outils
  - `atop`, `htop`

`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
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
7893 20   0  178m 3356 1220 S  0.0  0.1 postgres: autovacuum launcher
```

Attention à la valeur de la colonne `free`. La mémoire réellement disponible correspond plutôt à la soustraction `total - (used + buffers + cached)` (`cached` étant le cache disque mémoire du noyau). En réalité, c'est un peu moins, car tout ce qu'il y a dans `cache` ne peut être libéré sans recourir au *swapping*. Les versions plus modernes de `top` affichent une colonne `avail Mem`, équivalent de la colonne `available` de la commande `free`, et qui correspond beaucoup mieux à l'idée de « mémoire disponible ».

`top` n'existe pas directement sur Solaris. L'outil par défaut sur ce système est `prstat`.

6.2.3 Unix - iotop

- Principal intérêt : `%IO`
- Accès root nécessaire

`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
  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:
                                                       background writer
    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 rafraîchi fréquemment (toutes les secondes par défaut). Cet outil ne peut être utilisé qu'en tant qu'administrateur (**root**).

6.2.4 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.

Contrairement à `top` ou `iotop`, il envoie l'information directement sur la sortie standard, sans utiliser une interface particulière.

Cette commande accepte plusieurs options en ligne de commande, mais il faut fournir au minimum un argument indiquant la fréquence de rafraichissement. En l'absence du deuxième argument `count`, la commande s'exécute en permanence jusqu'à son arrêt avec un Ctrl-C. Ce comportement est identique pour les commandes `iostat` et `sar` notamment.

```
$ 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
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*.

6.2.5 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.
- Les deux derniers arguments en fin de commande ont la même sémantique que pour `vmstat`.

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 mode étendu compact :

```
$ iostat -d -x --dec=1 -s sdb 1

Device             tps      kB/s    rqm/s   await  areq-sz  aqu-sz  %util
sdb               76.0     324.0      4.0     0.8      4.3     0.1    1.2

Device             tps      kB/s    rqm/s   await  areq-sz  aqu-sz  %util
sdb              192.0  139228.0     49.0     8.1    725.1     1.5   28.0

Device             tps      kB/s    rqm/s   await  areq-sz  aqu-sz  %util
sdb              523.0  364236.0     86.0     9.0    696.4     4.7   70.4
```

Les colonnes ont les significations suivantes :

- `Device` : le périphérique
- `rrqm/s` et `wrqm/s` : `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 10 000 RPMS, 300 pour un 15 000 RPMS, et 10000 pour un SSD.
- `rkB/s` et `wkB/s` : les débits en lecture et écriture. Ils peuvent être faibles, avec un disque pourtant à 100 %.
- `areq-sz` (`avgrq-sz` dans les anciennes versions) : taille moyenne d'une requête. Plus elle est proche de 1 (1 ko), 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.
- `aqu-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. 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).
- `%util` : le pourcentage d'utilisation. Une valeur proche de 100 indique une saturation pour les disques rotatifs classiques, mais pas forcément pour les système RAID ou les disques SSD qui peuvent traiter plusieurs requêtes en parallèle.

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
```

6.2.6 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 les distributions Linux modernes intégrant systemd, une fois `sysstat` installé, il faut configurer son exécution automatique pour récupérer des statistiques périodiques avec :

```
sudo systemctl enable sysstat
sudo systemctl start  sysstat
```

Il est par ailleurs recommandé de positionner la variable `SADC_OPTIONS` à `"-S XALL"` dans le fichier de configuration (`/etc/sysstat/sysstat` pour Debian).

Le paquet sysstat dispose notamment de l'outil `pidstat`. Ce dernier récupère les informations système spécifiques à un processus (et en option à ses fils).

Pour plus d'information, consultez le readme[1].

6.2.7 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 -g
               total        used        free      shared  buff/cache   available
Mem:             251           9          15           8         226         232
Swap:             12           0          11
```

Ce serveur dispose de 251 Go de mémoire d'après la colonne `total`. Les applications utilisent 9 Go de mémoire. Seuls 15 Go ne sont pas utilisés. Le système utilise 226 Go de cette mémoire pour son cache disque (et un peu de bufferisation au niveau noyau), comme le montre la colonne `buff/cache`. La colonne `available` correspond à la quantité de mémoire libre, plus celle que le noyau est capable de libérer sans recourir au *swapping*. On voit ici que c'est un peu inférieur à la somme de `free` et `buff/cache`.

[1] https://github.com/sysstat/sysstat

6.3 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.

6.3.1 Windows - tasklist

- `ps` et `grep` en une commande

tasklist est le seul outil en ligne de commande discuté 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[2] pour plus de détails.

6.3.2 Windows - Process Monitor

- Surveillance des processus
- Filtres
- Récupération de la ligne de commande, identificateur de session et utilisateur
- Site officiel[3]

Process Monitor permet de lister les appels système des processus, comme le montre la copie d'écran ci-dessous :

[2] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tasklist

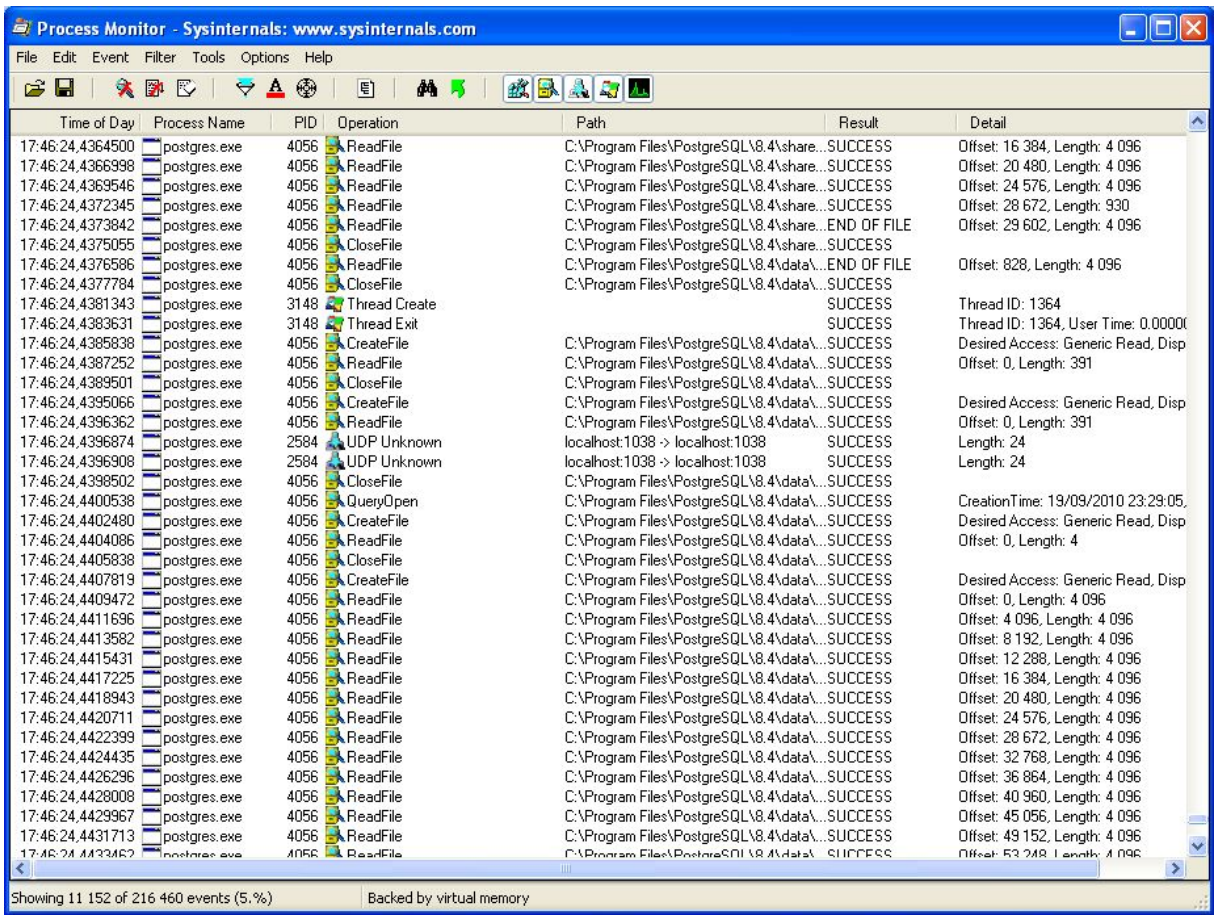


Figure 6/ .1: Process Monitor

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.

6.3.3 Windows - Process Explorer

- Semblable à `top`
- Site officiel[4]

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.

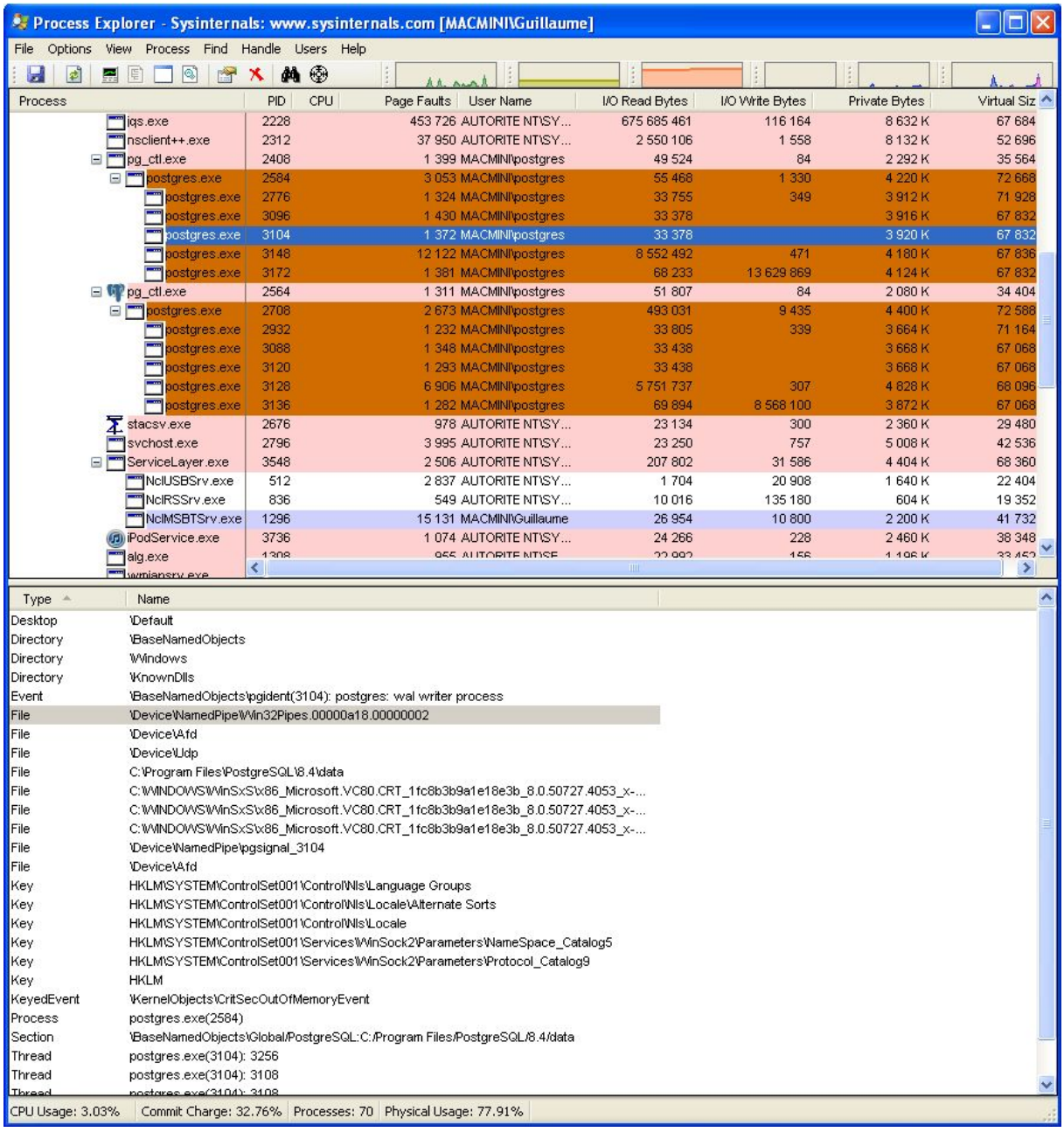


Figure 6/ .2: Process Explorer

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 :

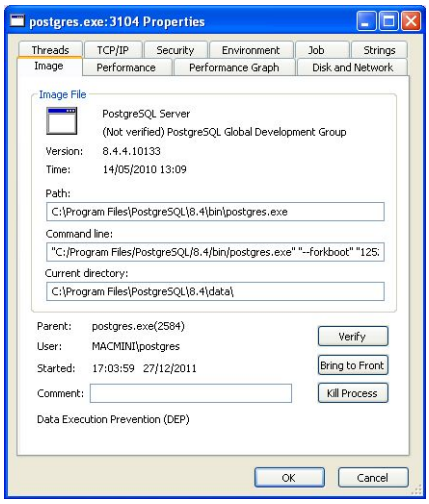


Figure 6/ .3: Process Explorer

- le deuxième, « Performances » fournit des informations textuelles sur les performances :

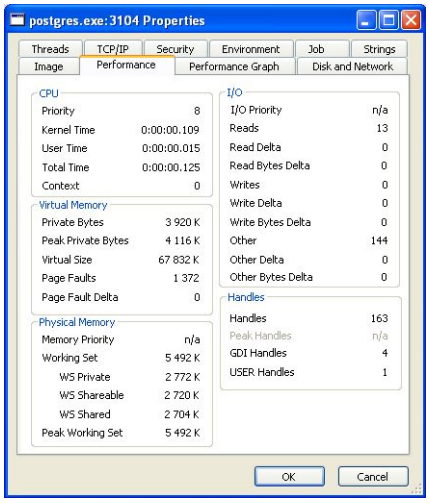


Figure 6/ .4: Process Explorer

- le troisième affiche quelques graphes :

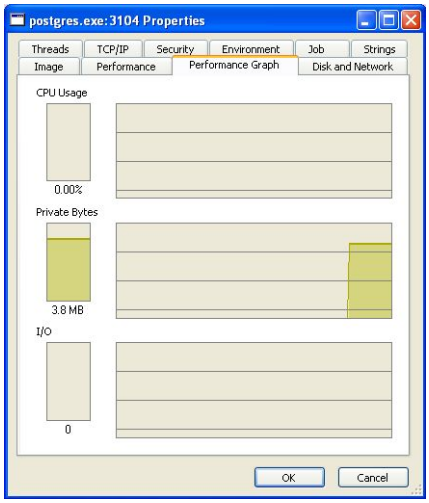


Figure 6/ .5: Process Explorer

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.

6.3.4 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.

6.4 SURVEILLER L'ACTIVITÉ DE POSTGRESQL

- Plusieurs aspects à surveiller :
  - activité de la base
  - activité sur les tables
  - requêtes SQL
  - écritures

Superviser une instance PostgreSQL consiste à surveiller à la fois ce qui s'y passe, depuis quelles sources, vers quelles tables, selon quelles requêtes et comment sont gérées les écritures.

PostgreSQL offre de nombreuses vues internes pour suivre cela.

6.4.1 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
- Erreurs de checksums
- Temps d'entrées/sorties

```
# \d pg_stat_database
```

```
                    Vue « pg_catalog.pg_stat_database »
        Colonne          |           Type           | …
-------------------------+--------------------------+---
 datid                   | oid                      |
 datname                 | name                     |
 numbackends             | integer                  |
 xact_commit             | bigint                   |
 xact_rollback           | bigint                   |
 blks_read               | bigint                   |
 blks_hit                | bigint                   |
 tup_returned            | bigint                   |
 tup_fetched             | bigint                   |
 tup_inserted            | bigint                   |
 tup_updated             | bigint                   |
 tup_deleted             | bigint                   |
```

```
conflicts                | bigint                   |
temp_files               | bigint                   |
temp_bytes               | bigint                   |
deadlocks                | bigint                   |
checksum_failures        | bigint                   |
checksum_last_failure    | timestamp with time zone |
blk_read_time            | double precision         |
blk_write_time           | double precision         |
session_time             | double precision         |
active_time              | double precision         |
idle_in_transaction_time | double precision         |
sessions                 | bigint                   |
sessions_abandoned       | bigint                   |
sessions_fatal           | bigint                   |
sessions_killed          | bigint                   |
stats_reset              | timestamp with time zone |
```

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 serveur secondaire) ;
- `temp_files` : le nombre de fichiers temporaires (utilisés pour le tri) créés par cette base depuis son démarrage ;
- `temp_bytes` : le nombre d'octets correspondant à ces fichiers temporaires : permet de trouver les bases effectuant beaucoup de tris sur disque ;
- `deadlocks` : le nombre de deadlocks (interblocages) ;
- `checksum_failures` : le nombre d'échecs lors de la vérification d'une somme de contrôle;
- `checksum_last_failure` : l'horodatage du dernier échec ;

- `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 ;
- `session_time` : temps passé par les sessions sur cette base, en millisecondes ;
- `active_time` : temps passé par les sessions à exécuter des requêtes SQL dans cette base ;
- `idle_in_transaction_time` : temps passé par les sessions dans une transaction mais sans exécuter de requête ;
- `sessions` : nombre total de sessions établies sur cette base ;
- `sessions_abandoned` : nombre total de sessions sur cette base abandonnées par le client ;
- `sessions_fatal` : nombre total de sessions terminées par des erreurs fatales sur cette base ;
- `sessions_killed` : nombre total de sessions terminées par l'administrateur ;
- `stats_reset` : la date de dernière remise à zéro des compteurs de cette vue.

6.5 GÉRER LES CONNEXIONS

- qui est connecté ?
- qui fait quoi ?
- qui est bloqué ?
- qui bloque les autres ?
- comment arrêter une requête ?

6.5.1 Vue pg_stat_activity

- Liste des processus
  - sessions (backends)
  - processus en tâche de fond (10+)
- Requête en cours/dernière exécutée
- *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). On y trouve notamment les noms des utilisateurs connectés et les requêtes, et leur statuts :

```
SELECT datname, pid, usename, application_name, backend_start, state, backend_type,
↪  query
FROM   pg_stat_activity
\gx
```

```
-[ RECORD 1 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26378
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236776+02
state            | ¤
backend_type     | autovacuum launcher
query            |
-[ RECORD 2 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26380
usename          | postgres
application_name |
backend_start    | 2019-10-24 18:25:28.238157+02
state            | ¤
```

```
backend_type     | logical replication launcher
query            |
-[ RECORD 3 ]----+-----------------------------------------------------------
datname          | pgbench
pid              | 22324
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.167611+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + -3810
                   WHERE aid = 91273;
-[ RECORD 4 ]----+-----------------------------------------------------------
datname          | postgres
pid              | 22429
usename          | postgres
application_name | psql
backend_start    | 2019-10-28 10:27:09.599426+01
state            | active
backend_type     | client backend
query            | select datname, pid, usename, application_name,
                   backend_start, state, backend_type, query from pg_stat_activity
-[ RECORD 5 ]----+-----------------------------------------------------------
datname          | pgbench
pid              | 22325
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.172585+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 4360
                   WHERE aid = 407881;
-[ RECORD 6 ]----+-----------------------------------------------------------
datname          | pgbench
pid              | 22326
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.178514+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 2865
                   WHERE aid = 8138;
-[ RECORD 7 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26376
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235574+02
state            | ¤
backend_type     | background writer
query            |
-[ RECORD 8 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26375
usename          | ¤
application_name |
```

```
backend_start    | 2019-10-24 18:25:28.235064+02
state            | ¤
backend_type     | checkpointer
query            |
-[ RECORD 9 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26377
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236239+02
state            | ¤
backend_type     | walwriter
query            |
```

Cette vue fournit aussi des informations sur ce que chaque session attend. Pour les détails sur `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente), voir le tableau des événements d'attente[5].

```
# SELECT datname, pid, wait_event_type, wait_event, query FROM pg_stat_activity
  WHERE backend_type='client backend' AND wait_event IS NOT NULL \gx

-[ RECORD 1 ]---+-----------------------------------------------------------
datname         | pgbench
pid             | 1590
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | UPDATE pgbench_accounts SET abalance = abalance + 1438
                  WHERE aid = 747101;
-[ RECORD 2 ]---+-----------------------------------------------------------
datname         | pgbench
pid             | 1591
state           | idle
wait_event_type | Client
wait_event      | ClientRead
query           | END;
-[ RECORD 3 ]---+-----------------------------------------------------------
datname         | pgbench
pid             | 1593
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
                  VALUES (3, 4, 870364, -703, CURRENT_TIMESTAMP);
-[ RECORD 4 ]---+-----------------------------------------------------------
datname         | postgres
pid             | 1018
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | delete from t1 ;
-[ RECORD 5 ]---+-----------------------------------------------------------
datname         | postgres
pid             | 1457
```

[5] https://docs.postgresql.fr/current/monitoring-stats.html#wait-event-table

```
state           | active
wait_event_type | Lock
wait_event      | transactionid
query           | delete  from t1 ;
```

La vue contient aussi des informations sur l'outil client utilisé, les dates de connexion ou changement d'état, les numéros de transaction impliqués :

```
\d pg_stat_activity

        Vue « pg_catalog.pg_stat_activity »

     Colonne      |           Type           | ...
------------------+--------------------------+----
 datid            | oid                      |
 datname          | name                     |
 pid              | integer                  |
 leader_pid       | integer                  |
 usesysid         | oid                      |
 usename          | name                     |
 application_name | text                     |
 client_addr      | inet                     |
 client_hostname  | text                     |
 client_port      | integer                  |
 backend_start    | timestamp with time zone |
 xact_start       | timestamp with time zone |
 query_start      | timestamp with time zone |
 state_change     | timestamp with time zone |
 wait_event_type  | text                     |
 wait_event       | text                     |
 state            | text                     |
 backend_xid      | xid                      |
 backend_xmin     | xid                      |
 query            | text                     |
 backend_type     | text                     |
```

Cette vue a beaucoup évolué au fil des versions, et des champs ont porté d'autres noms. En version 9.6, la colonne `waiting` est remplacée par les colonnes `wait_event_type` et `wait_event`. La version 10 ajoute une colonne supplémentaire, `backend_type`, indiquant le type de processus : par exemple `background worker`, `background writer`, `autovacuum launcher`, `client backend`, `walsender`, `checkpointer`, `walwriter`. La version 13 ajoute une nouvelle colonne, `leader_pid`, indiquant le PID du leader dans le cas de l'exécution parallélisée d'une requête, et NULL pour le leader. Depuis la version 14, il est possible d'avoir en plus l'identifiant de la requête grâce à la nouvelle colonne `query_id`. Pour cela, il faut activer le paramètre `compute_query_id`.

Les autres champs contiennent :

- `datname` : le nom de la base à laquelle la session est connectée (`datid` est son identifiant (OID)) ;
- `pid` : le numéro du processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client ;
- `usename` : le nom de l'utilisateur connecté (`usesysid` est son OID) ;

- `application_name` : un nom facultatif renseigné par l'application cliente (avec `SET application_name TO 'n`
- `client_addr` : l'adresse IP du client connecté (ou `NULL` si connexion sur socket Unix) ;
- `client_hostname` : le nom associé à cette IP, renseigné si `log_hostname` est à `on` (ce paramètre peut fortement ralentir la connexion à cause de la résolution DNS) ;
- `client_port` : le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP ;
- `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.

Dans les versions récentes, `query` contient la dernière requête exécutée, qui peut être terminée alors que la session est depuis longtemps en *idle in transaction* ou *idle*. Si les requêtes font couramment plus de 1 ko, il faudra augmenter `track_activity_query_size` pour qu'elles ne soient pas tronquées.

Certains champs de cette vue ne sont renseignés que si `track_activities` est à `on` (valeur par défaut).

6.5.2 Arrêter une requête ou une session

- Annuler une requête
  - `pg_cancel_backend (pid int)`
  - `pg_ctl kill INT pid` (éviter)
  - `kill -SIGINT pid`, `kill -2 pid` (éviter)
- Fermer une connexion
  - `pg_terminate_backend(pid int, timeout bigint)`
  - `pg_ctl kill TERM pid` (éviter)
  - `kill -SIGTERM pid`, `kill -15 pid` (éviter)
- Jamais `kill -9` ou `kill -SIGKILL` !!

Les fonctions `pg_cancel_backend` et `pg_terminate_backend` sont le plus souvent utilisées. Le paramètre est le numéro du processus auprès de l'OS.

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.

L'utilisateur, connecté au processus de PID 10901 comme l'indique la fonction `pg_backend_pid`, exécute une très grosse insertion :

```
SELECT pg_backend_pid();

 pg_backend_pid
----------------
          10901

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 :

```
SELECT pg_cancel_backend(10901);

 pg_cancel_backend
-------------------
 t
```

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 devra se conclure par un `ROLLBACK` à cause de l'erreur. À noter cependant qu'il n'est pas possible d'annuler une transaction qui n'exécute rien à ce moment. En conséquence, `pg_cancel_backend` ne suffit pas pour parer à une session en statut `idle in transaction`.

Il est possible d'aller plus loin en supprimant la connexion d'un utilisateur. Cela se fait avec la fonction `pg_terminate_backend` qui se manie de la même manière :

```
SELECT pid, datname, usename, application_name,state
FROM pg_stat_activity WHERE backend_type = 'client backend' ;

 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   13267 | b1      | u1        | psql             | idle
   10901 | b1      | guillaume | psql             | active

SELECT pg_terminate_backend(13267);

 pg_terminate_backend
----------------------
 t

SELECT pid, datname, usename, application_name, state
     FROM pg_stat_activity WHERE backend_type='client backend';

 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   10901 | b1      | guillaume | psql             | active
```

L'utilisateur de la session supprimée verra un message d'erreur au prochain ordre qu'il enverra. `psql` se reconnecte automatiquement mais cela n'est pas forcément le cas d'autres outils client.

```
SELECT 1 ;

FATAL:  terminating connection due to administrator command
la connexion au serveur a été coupée de façon inattendue
        Le serveur s'est peut-être arrêté anormalement avant ou durant le
        traitement de la requête.
La connexion au serveur a été perdue. Tentative de réinitialisation : Succès.
Temps : 7,309 ms
```

Par défaut, `pg_terminate_backend` renvoie `true` dès qu'il a pu envoyer le signal, sans tester son effet. À partir de la version 14, il est possible de préciser une durée comme deuxième argument de `pg_terminate_backend`. Dans l'exemple suivant, on attend 2 s (2000 ms) avant de constater, ici, que le processus visé n'est toujours pas arrêté, et de renvoyer `false` et un avertissement :

```
# SELECT pg_terminate_backend (178896,2000) ;

WARNING:  backend with PID 178896 did not terminate within 2000 milliseconds

 pg_terminate_backend
----------------------
 f
```

Ce message ne veut pas dire que le processus ne s'arrêtera pas finalement, plus tard.

Depuis la ligne de commande du serveur, un `kill <pid>` (c'est-à-dire `kill -SIGTERM` ou `kill -15`) a le même effet qu'un `SELECT pg_terminate_backend (<pid>)`. Cette méthode n'est pas recommandée car il n'y a pas de vérification que vous tuez bien un processus **postgres**. `pg_ctl` dispose d'une action `kill` pour envoyer un signal à un processus. Malheureusement, là-aussi, `pg_ctl` ne fait pas de différence entre les processus postgres et les autres processus.

N'utilisez jamais `kill -9 <pid>` (ou `kill -SIGKILL`), ou (sous Windows) `taskkill /f /pid <pid>` pour tuer une connexion : l'arrêt est alors brutal, et le processus principal n'a aucun moyen de savoir pourquoi. Pour éviter une corruption de la mémoire partagée, il va arrêter et redémarrer immédiatement tous les processus, déconnectant tous les utilisateurs au passage !

L'utilisation de `pg_terminate_backend()` et `pg_cancel_backend()` n'est disponible que pour les utilisateurs appartenant au même rôle que l'utilisateur à déconnecter, les utilisateurs membres du rôle `pg_signal_backend` et bien sûr les superutilisateurs.

6.5.3 pg_stat_ssl

Quand le SSL est activé sur le serveur, cette vue indique pour chaque connexion cliente les informations suivantes :

- SSL activé ou non
- Version SSL
- Suite de chiffrement
- Nombre de bits pour algorithme de chiffrement
- Compression activée ou non
- Distinguished Name (DN) du certificat client

La définition de la vue est celle-ci :

```
\d pg_stat_ssl
```

```
                  Vue « pg_catalog.pg_stat_ssl »
    Colonne     |  Type   | Collationnement | NULL-able | Par défaut
----------------+---------+-----------------+-----------+------------
 pid            | integer |                 |           |
 ssl            | boolean |                 |           |
 version        | text    |                 |           |
 cipher         | text    |                 |           |
 bits           | integer |                 |           |
 compression    | boolean |                 |           |
 client_dn      | text    |                 |           |
```

```
client_serial | numeric |                 |          |
issuer_dn     | text    |                 |          |
```

- `pid` : numéro du processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client ;
- `ssl` : ssl activé ou non ;
- `version` : version ssl utilisée, *null* si ssl n'est pas utilisé ;
- `cipher` : suite de chiffrement utilisée, *null* si ssl n'est pas utilisé ;
- `bits` : nombre de bits de la suite de chiffrement, *null* si ssl n'est pas utilisé ;
- `compression` : compression activée ou non, *null* si ssl n'est pas utilisé ;
- `client_dn` : champ *Distinguished Name (DN)* du certificat client, *null* si aucun certificat client n'est utilisé ou si ssl n'est pas utilisé ;
- `client_serial` : numéro de série du certificat client, *null* si aucun certificat client n'est utilisé ou si ssl n'est pas utilisé ;
- `issuer_dn` : champ *Distinguished Name (DN)* du constructeur du certificat client, *null* si aucun certificat client n'est utilisé ou si ssl n'est pas utilisé ;

6.6 VERROUS

- Visualisation des verrous en place
- Tous types de verrous sur objets
- Complexe à interpréter
  - verrous sur enregistrements pas directement visibles
  - voir l'article détaillé[6] sur la base de connaissance Dalibo.

La vue `pg_locks` 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 cible (quand verrou de type `page` ou `tuple`).
- `tuple` : le numéro de l'enregistrement cible (quand verrou de type `tuple`).
- `virtualxid` : le numéro de la transaction virtuelle cible (quand verrou de type `virtualxid`).
- `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`).
- `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`).

6.6.1 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 nécessitent 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.

6.6.2 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.

6.7 SURVEILLER L'ACTIVITÉ SUR LES TABLES

- Quelle taille font mes objets ?
- Quel est leur taux de fragmentation ?
- Comment sont-ils accédés ?

6.7.1 Obtenir la taille des objets

- Pour une table :
  - `pg_relation_size` : *heap*
  - `pg_table_size` : + TOAST + divers
- Index : `pg_indexes_size`
- Table + index : `pg_total_relation_size`
- Plus lisibles avec `pg_size_pretty`

Une table comprend différents éléments : la partie principale ou *main* (ou *heap*) ; pas toujours la plus grosse ; des objets techniques comme la *visibility map* ou la *Free Space Map* ou l'*init* ; parfois des données dans une table TOAST associée ; et les éventuels index. La « taille » de la table dépend donc de ce que l'on entend précisément.

`pg_relation_size` donne la taille de la relation, par défaut de la partie *main*, mais on peut demander aussi les parties techniques. Elle fonctionne aussi pour la table TOAST si l'on a son nom ou son OID.

`pg_total_relation_size` fournit la taille totale de tous les éléments, dont les index et la partie TOAST.

`pg_table_size` renvoie la taille de la table avec le TOAST et les parties techniques, mais sans les index (donc essentiellement les données).

`pg_indexes_size` calcule la taille totale des index d'une table.

Toutes ces fonctions acceptent en paramètre soit un OID soit le nom en texte.

Voici un exemple d'une table avec deux index avec les quatre fonctions :

```
CREATE UNLOGGED TABLE donnees_aleatoires (
                            i int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
                            a text);
```

```
-- 6000 lignes de blancs
INSERT INTO donnees_aleatoires (a)
SELECT repeat (' ',2000) FROM generate_series (1,6000);

-- Pour la Visibility Map
VACUUM donnees_aleatoires ;

SELECT  pg_relation_size('donnees_aleatoires'), -- partie 'main'
        pg_relation_size('donnees_aleatoires', 'vm')   AS "pg_relation_size (,vm)",
        pg_relation_size('donnees_aleatoires', 'fsm')  AS "pg_relation_size (,fsm)",
       pg_relation_size('donnees_aleatoires', 'init') AS "pg_relation_size (,init)",
        pg_table_size   ('donnees_aleatoires'),
        pg_indexes_size ('donnees_aleatoires'),
        pg_total_relation_size('donnees_aleatoires')
\gx
```

```
-[ RECORD 1 ]------------+---------
pg_relation_size         | 12288000
pg_relation_size (,vm)   | 8192
pg_relation_size (,fsm)  | 24576
pg_relation_size (,init) | 0
pg_table_size            | 12337152
pg_indexes_size          | 163840
pg_total_relation_size   | 12500992
```

La fonction `pg_size_pretty` est souvent utilisée pour renvoyer un texte plus lisible :

```
SELECT  pg_size_pretty(pg_relation_size('donnees_aleatoires'))
            AS pg_relation_size,
        pg_size_pretty(pg_relation_size('donnees_aleatoires', 'vm'))
            AS "pg_relation_size (,vm)",
        pg_size_pretty(pg_relation_size('donnees_aleatoires', 'fsm'))
            AS "pg_relation_size (,fsm)",
        pg_size_pretty(pg_relation_size('donnees_aleatoires', 'init'))
            AS "pg_relation_size (,init)",
        pg_size_pretty(pg_table_size('donnees_aleatoires'))
            AS pg_table_size,
        pg_size_pretty(pg_indexes_size('donnees_aleatoires'))
            AS pg_indexes_size,
        pg_size_pretty(pg_total_relation_size('donnees_aleatoires'))
            AS  pg_total_relation_size
\gx
```

```
-[ RECORD 1 ]------------+-----------
pg_relation_size         | 12 MB
pg_relation_size (,vm)   | 8192 bytes
pg_relation_size (,fsm)  | 24 kB
pg_relation_size (,init) | 0 bytes
pg_table_size            | 12 MB
pg_indexes_size          | 160 kB
pg_total_relation_size   | 12 MB
```

Ajoutons des données peu compressibles pour la partie TOAST :

```
\COPY donnees_aleatoires(a) FROM PROGRAM 'cat /dev/urandom|tr -dc A-Z|fold -bw
↪  5000|head -n 5000' ;
```

```
VACUUM ANALYZE donnees_aleatoires ;
```

```
SELECT
    oid AS table_oid,
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid,
    reltoastrelid::regclass::text AS toast_table,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table",
    pg_size_pretty(pg_relation_size(c.oid, 'main')) AS "  Heap",
    pg_size_pretty(pg_relation_size(c.oid, 'vm')) AS "  VM",
    pg_size_pretty(pg_relation_size(c.oid, 'fsm')) AS "  FSM",
    pg_size_pretty(pg_relation_size(c.oid, 'init')) AS "  Init",
    pg_size_pretty(pg_total_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname = 'donnees_aleatoires'
\gx
```

```
-[ RECORD 1 ]------+-------------------------
table_oid          | 4200073
table              | public.donnees_aleatoires
reltoastrelid      | 4200076
toast_table        | pg_toast.pg_toast_4200073
nb_lignes_estimees | 6000
 Table             | 40 MB
  Heap             | 12 MB
  VM               | 8192 bytes
  FSM              | 24 kB
  Init             | 0 bytes
  Toast            | 28 MB
 Index             | 264 kB
Total              | 41 MB
```

Le wiki[7] contient d'autres exemples, notamment sur le calcul de la taille totale d'une table partitionnée.

[7] https://wiki.postgresql.org/wiki/Disk_Usage

6.7.2 Mesurer la fragmentation des objets

- Fragmentation induite par MVCC
  - tables et index
- Mesure précise de la fragmentation :
  - extension `pgstattuple`
- Estimer la fragmentation :
  - pgsql-bloat-estimation[8]
  - `pgstattuple_approx()` (tables)
  - supervision avec `check_pgactivity`

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.), puis des ralentissements de la base de données. Il est donc nécessaire de pouvoir détecter les cas où la base présente une fragmentation trop élevée.

La fragmentation recouvre deux types d'espaces : les lignes mortes à nettoyer, et l'espace libre et utilisable, parfois excessif.

Estimation rapide :

`pg_stat_user_tables.n_dead_tup` à une valeur élevée est déjà un indicateur qu'un `VACUUM` est nécessaire.

De manière plus complète, les requêtes de Jehan-Guillaume de Rorthais dans le dépôt indiqué ci-dessus 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, c'est-à-dire un espace (mort ou réutilisable) excessif

Attention : il s'agit seulement d'une estimation de la fragmentation d'une table. Les statistiques (`ANALYZE`) doivent être fraîches. Dans certains cas, l'estimation n'est pas très précise. Par contre elle est très rapide.

Calcul précis :

Pour mesurer très précisément la fragmentation d'une table ou d'un index, il faut installer l'extension pgstattuple[9]. Celle-ci par contre est susceptible de lire toute la table, ce qui est donc long.

Il existe une fonction `pgstattuple()` pour les tables et index, et une fonction `pgstatindex()` plus précise pour les index.

[9] https://docs.postgresql.fr/current/pgstattuple.html

Une autre fonction, `pgstattuple_approx()`, se base sur la *visibility map* et la *Free Space Map*. Elle ne fonctionne que pour les tables. Elle est moins précise mais plus rapide que `pgstattuple()`, mais reste plus lente que l'estimation basée sur les statistiques.

Exemple :

Les ordres ci-dessous génèrent de la fragmentation dans une table de 42 Mo dont on efface 90 % des lignes :

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;
DROP TABLE IF EXISTS demo_bloat ;
CREATE TABLE demo_bloat (i integer, filler char(10) default ' ');
-- désactivation de l'autovacuum pour l'exemple
ALTER TABLE demo_bloat SET (autovacuum_enabled=false);
-- insertion puis suppression de 90% des lignes
INSERT INTO demo_bloat SELECT i FROM generate_series(1, 1000000) i ;
DELETE FROM demo_bloat WHERE i < 900000 ;

SELECT * FROM pg_stat_user_tables WHERE relname ='demo_bloat';

-[ RECORD 1 ]-------+-----------
relid               | 10837034
schemaname          | public
relname             | demo_bloat
seq_scan            | 1
seq_tup_read        | 1000000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 1000000
n_tup_upd           | 0
n_tup_del           | 899999
n_tup_hot_upd       | 0
n_live_tup          | 100001
n_dead_tup          | 899999
n_mod_since_analyze | 1899999
n_ins_since_vacuum  | 1000000
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

`n_dead_tup` (lignes mortes) est ici très élevé.

L'estimation retournée par la requête d'estimation proposée plus haut est ici très proche de la réalité car les statistiques sont fraîches :

```
ANALYZE demo_bloat ;
\x on
\i ./pgsql-bloat-estimation/table/table_bloat.sql

(…)
-[ RECORD 41 ]---+------------------------
```

```
current_database | postgres
schemaname       | public
tblname          | demo_bloat
real_size        | 44285952
extra_size       | 39870464
extra_pct        | 90.02959674435812
fillfactor       | 100
bloat_size       | 39870464
bloat_pct        | 90.02959674435812
(…)
```

Le *bloat* et l'espace « en trop » (*extra*) sont tous les deux à 90 % car le *fillfactor* est de 100 %.

Avec `pgstattuple()`, les colonnes `free_space` et `free_percent` donnent la taille et le pourcentage d'espace libre :

```
SELECT * FROM pgstattuple ('demo_bloat') \gx
```

```
-[ RECORD 1 ]------+-------
table_len          | 44285952
tuple_count        | 100001
tuple_len          | 3900039
tuple_percent      | 8.81
dead_tuple_count   | 899999
dead_tuple_len     | 35099961
dead_tuple_percent | 79.26
free_space         | 134584
free_percent       | 0.3
```

Il n'y a presque pas d'espace libre (*free*) car beaucoup de lignes sont encore mortes (`dead_tuple_percent` indique 79 % de lignes mortes).

La fonction d'approximation est ici plus rapide (deux fois moins de blocs lus dans ce cas précis) pour le même résultat :

```
SELECT * FROM pgstattuple_approx ('demo_bloat') \gx
```

```
-[ RECORD 1 ]--------+-------------------
table_len            | 44285952
scanned_percent      | 100
approx_tuple_count   | 100001
approx_tuple_len     | 3900039
approx_tuple_percent | 8.80649240644076
dead_tuple_count     | 899999
dead_tuple_len       | 35099961
dead_tuple_percent   | 79.2575510175326
approx_free_space    | 134584
approx_free_percent  | 0.3038977235941546
```

Si on nettoie la table, on retrouve 90 % d'espace réellement libre :

```
VACUUM demo_bloat;

SELECT * FROM pgstattuple('demo_bloat');
```

```
-[ RECORD 1 ]------+-------
table_len          | 44285952
```

```
tuple_count        | 100001
tuple_len          | 3900039
tuple_percent      | 8.81
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 39714448
free_percent       | 89.68
```

(La fonction approximative renverra presque les mêmes chiffres :

```
SELECT * FROM pgstattuple_approx('demo_bloat');
```

```
-[ RECORD 1 ]--------+-------------------
table_len            | 44285952
scanned_percent      | 0
approx_tuple_count   | 100001
approx_tuple_len     | 4584480
approx_tuple_percent | 10.351996046059934
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 39701472
approx_free_percent  | 89.64800395394006
```

Le résultat de la requête d'estimation ne changera pas, indiquant toujours 90 % de *bloat*.

Le choix de la bonne requête dépendra de ce que l'on veut. Si l'on cherche juste à savoir si un `VACUUM FULL` est nécessaire, l'estimation suffit généralement et est très rapide. Si l'on suspecte que l'estimation est fausse et que l'on a plus de temps, les deux fonctions de `pgstattuple` sont plus précises.

6.7.3 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 (y compris ceux comptés dans `n_tup_hot_upd` et `n_tup_newpage_upd`) ou supprimés ;
- `n_tup_hot_upd` : nombre d'enregistrements mis à jour par mécanisme HOT (c'est-à-dire chaînés au sein d'un même bloc) ;
- `n_tup_newpage_upd` : nombre de mises à jour ayant nécessité d'aller écrire la nouvelle ligne dans un autre bloc, faute de place dans le bloc d'origine (à partir de PostgreSQL 16) ;
- `n_live_tup` : estimation du nombre d'enregistrements « vivants » ;
- `n_dead_tup` : estimation du nombre d'enregistrements « morts » (supprimés mais non nettoyés) depuis le dernier `VACUUM` ;
- `n_mod_since_analyze` : nombre d'enregistrements modifiés depuis le dernier `ANALYZE` ;
- `n_ins_since_vacuum` : estimation du nombre d'enregistrements insérés depuis le dernier `VACUUM` ;
- `last_vacuum` : timestamp du dernier `VACUUM` ;
- `last_autovacuum` : timestamp du dernier `VACUUM` automatique ;
- `last_analyze` : timestamp du dernier `ANALYZE` ;
- `last_autoanalyze` : timestamp du 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.

Contrairement aux autres colonnes, les colonnes `n_live_tup`, `n_dead_tup` et `n_mod_since_analyze` sont des estimations. Leur valeurs changent au fur et à mesure de l'exécution de commandes `INSERT`, `UPDATE`, `DELETE`. Elles sont aussi recalculées complètement lors de l'exécution d'un `VACUUM` et d'un `ANALYZE`. De ce fait, leur valeur peut changer entre deux `VACUUM` même si aucune écriture de ligne n'a eu lieu.

6.7.4 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 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.

6.7.5 Vues pg_statio_user_tables & pg_statio_user_indexes

- Opérations au niveau bloc
- Demandés au système ou trouvés dans le cache de PostgreSQL
- Pour calculer des hit ratios :

```
idx_blks_hit::float / (idx_blks_read + idx_blks_hit)
```

Voici la description des différentes colonnes de `pg_statio_user_tables` :

```
# \d pg_statio_user_tables
                Vue « pg_catalog.pg_statio_user_tables »
     Colonne      |  Type  | Collationnement | NULL-able | Par défaut
------------------+--------+-----------------+-----------+------------
 relid            | oid    |                 |           |
```

```
 schemaname      | name   |                 |           |
 relname         | name   |                 |           |
 heap_blks_read  | bigint |                 |           |
 heap_blks_hit   | bigint |                 |           |
 idx_blks_read   | bigint |                 |           |
 idx_blks_hit    | bigint |                 |           |
 toast_blks_read | bigint |                 |           |
 toast_blks_hit  | bigint |                 |           |
 tidx_blks_read  | bigint |                 |           |
 tidx_blks_hit   | bigint |                 |           |
```

- `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` :

```
# \d pg_statio_user_indexes

             Vue « pg_catalog.pg_statio_user_indexes »
    Colonne     |  Type  | Collationnement | NULL-able | Par défaut
----------------+--------+-----------------+-----------+------------
 relid          | oid    |                 |           |
 indexrelid     | oid    |                 |           |
 schemaname     | name   |                 |           |
 relname        | name   |                 |           |
 indexrelname   | name   |                 |           |
 idx_blks_read | bigint |                 |           |
 idx_blks_hit  | bigint |                 |           |
```

- `indexrelid`, `indexrelname` : `OID` et nom de 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 (pour ne pas faire une division entière qui renverrait souvent 0), et que le `CASE` est destiné à éviter une division par zéro.

6.7.6 Vue pg_stat_io

Vue synthétique des opérations disques selon :

- le type de backend
  - backend, autovacuum, checkpointer…
- le type d'objet
  - table ou table temporaire
- le contexte
  - normal, vacuum, bulkread/bulkwrite…

Penser à activer `track_io_timing`

La nouvelle vue `pg_stat_io` permet d'obtenir des informations sur les opérations faites sur disques. Il y a différents compteurs : *reads* (lectures), *writes* (écritures), *read_time* et *write_time* (durées associées aux précédents), *extends* (extensions de fichiers), *hits* (lecture en cache de PostgreSQL), *evictions* (éviction du cache), etc. Ils sont calculés pour chaque combinaison de type de backend, objet I/O cible et contexte I/O. Les définitions des colonnes et des compteurs peuvent être trouvées dans la la documentation officielle[10].

Comme la plupart des vues statistiques, les données sont cumulatives. Une remise à zéro s'effectue avec :

```
SELECT pg_stat_reset_shared ('io');
```

Les champs `*_time` ne sont alimentés que si le paramètre `track_io_timing` a été activé. Ne sont pas tracées certaines opérations qui ne passent pas par le cache disque, comme les déplacements de table entre tablespace.

Exemples :

Si nous voulons connaître les opérations qui ont les durées de lectures hors du cache les plus longues :

```
SELECT backend_type, object, context, reads, read_time
FROM   pg_stat_io
ORDER BY read_time DESC NULLS LAST LIMIT 3 ;
```

[10] https://docs.postgresql.fr/16/monitoring-stats.html#MONITORING-PG-STAT-IO-VIEW

```
   backend_type     |  object  | context  |   reads   |   read_time
--------------------+----------+----------+-----------+---------------
 client backend     | relation | normal   | 640840357 | 738717779.603
 autovacuum worker  | relation | vacuum   | 117320999 |  16634388.118
 background worker  | relation | bulkread |  44481246 |   9749622.473
```

Le résultat indique que ce temps est essentiellement dépensé par des backends client, sur des tables non temporaires, dans un contexte « normal » (via les *shared buffers*). La présence de *reads* massifs indique peut-être des *shared buffers* trop petits (si les requêtes sont optimisées).

Une requête similaire pour les écritures est :

```
SELECT backend_type, object, context, writes, round(write_time) AS write_time
FROM pg_stat_io ORDER BY write_time DESC NULLS LAST LIMIT 3 ;
```

```
   backend_type     |  object  | context | writes | write_time
--------------------+----------+---------+--------+------------
 checkpointer       | relation | normal  | 435117 |      14370
 background writer  | relation | normal  |  74684 |       1049
 client backend     | relation | vacuum  |  25941 |        123
```

Ici, les écritures sont faites essentiellement par les checkpoints, accessoirement le `background writer`, ce qui est idéal.

Par contre, si la même requête renvoie ceci :

```
SELECT backend_type, object, context, writes, round(write_time) AS write_time
FROM pg_stat_io ORDER BY write_time DESC NULLS LAST LIMIT 5 ;
```

```
   backend_type     |  object  |  context  |  writes  | write_time
--------------------+----------+-----------+----------+------------
 client backend     | relation | normal    | 82556667 |    3770829
 autovacuum worker  | relation | vacuum    | 94262005 |    1847367
 checkpointer       | relation | normal    | 74210966 |     632146
 client backend     | relation | bulkwrite | 47901524 |     206759
 background writer  | relation | normal    | 10315801 |     147621
```

on en déduit que les backends écrivent beaucoup par eux-mêmes, un peu plus en nombre d'écritures que le `checkpointer`. Cela suggère que le `background writer` n'est pas assez agressif. Noter que les *autovacuum workers* procèdent aussi eux-mêmes à leurs écritures. Enfin le contexte *bulkwrite* indique l'utilisation de modes d'écritures en masse (par exemple des `CREATE TABLE … AS …`).

6.8 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

6.8.1 Trace des requêtes exécutées

- `log_min_duration_statements` = <temps minimal d'exécution>
  - `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
- `log_min_duration_sample` = <temps minimal d'exécution>
  - `log_statement_sample_rate` et/ou `log_transaction_sample_rate`
  - trace d'un ratio des requêtes
- 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.

Cependant, tracer toutes les requêtes peut poser problème. Le contournement habituel est de ne tracer que les requêtes dont l'exécution est supérieure à une certaine durée, mais cela cache tout le restant du trafic qui peut être conséquent et avoir un impact sur les performances globales du système. En version 13, une nouvelle fonctionnalité a été ajoutée : tracer un certain ratio de requêtes ou de transactions.

Si `log_statement_sample_rate` est configuré à une valeur strictement supérieure à zéro, la valeur correspondra au pourcentage de requêtes à tracer. Par exemple, en le configuration à 0,5, une requête sur deux sera tracée. Les requêtes réellement tracées dépendent de leur durée d'exécution. Cette durée doit être supérieure ou égale à la valeur du paramètre `log_min_duration_sample`.

Ce comportement est aussi disponible pour les transactions. Pour cela, il faut configurer le paramètre `log_transaction_sample_rate`.

6.8.2 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.

6.8.3 pg_stat_statements

- Ajoute la vue statistique `pg_stat_statements`
- Les requêtes sont normalisées
- Indique les requêtes exécutées
  - avec la durée d'exécution, l'utilisation du cache, etc.

Contrairement à pgBadger, `pg_stat_statements` 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`. Les requêtes sont normalisées (reconnues comme identiques même avec des paramètres différents), et identifiables grâce à un `queryid`. Une même requête peut apparaître sur plusieurs lignes de `pg_stat_statements` pour des bases et utilisateurs différents. Par contre, l'utilisation de schémas, implicitement ou pas, force un `queryid` différent.

L'installation et quelques exemples de requêtes sont proposés dans https://dali.bo/x2_html#pg_stat_statements.

Voici un exemple de requête sur la vue `pg_stat_statements` :

```
SELECT * FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 3 ;
```

```
-[ RECORD 1 ]---+----------------------------------------------------------
userid          | 10
dbid            | 63781
toplevel        | t
queryid         | -1739183385080879393
query           | UPDATE branches SET bbalance = bbalance + $1 WHERE bid = $2;
plans           | 0
[...]
calls           | 3000
total_exec_time | 20.716706
[...]
rows            | 3000
[...]
-[ RECORD 2 ]---+----------------------------------------------------------
userid          | 10
dbid            | 63781
toplevel        | t
queryid         | -1737296385080879394
query           | UPDATE tellers SET tbalance = tbalance + $1 WHERE tid = $2;
plans           | 0
[...]
calls           | 3000
```

```
total_exec_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.

6.8.4 Vue pg_stat_statements - métriques 1/5

Métriques intéressantes :

- Durée d'exécution :
  - `total_exec_time`
  - `min_exec_time` / `max_exec_time`
  - `stddev_exec_time`
  - `mean_exec_time`
- Avant la version 13, les colonnes n'avaient pas `_exec` dans leur nom
- Nombre de lignes retournées : `rows`

`pg_stat_statements` apporte des statistiques sur les durées d'exécutions des requêtes normalisées. Notamment :

- `total_exec_time` : temps d'exécution total ;
- `min_exec_time` et `max_exec_time` : durées d'exécution minimale et maximale d'une requête normalisée ;
- `mean_exec_time` : durée moyenne d'exécution ;
- `stddev_exec_time` : écart-type de la durée d'exécution, une métrique intéressante pour identifier une requête dont le temps d'exécution varie fortement ;
- `rows` : nombre total de lignes retournées.

6.8.5 Vue pg_stat_statements - métriques 2/5

- Durée d'optimisation (v13+) :
  - `total_plan_time`
  - `min_plan_time` / `max_plan_time`
  - `stddev_plan_time`
  - `mean_plan_time`

`pg_stat_statements` apporte des statistiques sur les durées d'optimisation des requêtes normalisées. Ainsi, `total_plan_time` indique le cumul d'optimisation total. `min_plan_time` et `max_plan_time` représentent respectivement la durée d'optimisation minimale et maximale d'une requête normalisée. La colonne `mean_plan_time` donne la durée moyenne d'optimisation alors que la colonne `stddev_plan_time` donne l'écart-type de la durée d'optimisation. Cette métrique peut être intéressante pour identifier une requête dont le temps d'optimisation varie fortement.

Toutes ces colonnes ne sont disponibles qu'à partir de la version 13.

6.8.6 Vue pg_stat_statements - métriques 3/5

- Accès à la mémoire partagée
  - `shared_blks_hit/read/dirtied/written`
- Accès à la mémoire de la session (tables temporaires…)
  - `local_blks_hit/read/dirtied/written`
- Lecture/écriture de fichiers temporaires
  - `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` : nombre de blocs lus directement dans le cache de PostgreSQL ;
- `shared_blks_read` : blocs lus demandés au système d'exploitation (donc lus sur le disque ou dans le cache du système) ;
- `shared_blks_dirtied` : nouveaux blocs « sales » générés par la requête par des mises à jour, insertions, suppressions, `VACUUM`…, et sans compter ceux qui l'étaient déjà auparavant ; ces blocs seront écrits sur disque ultérieurement ;
- `shared_blks_written` : blocs directements écrits sur disque, ce qui peut arriver s'il n'y a plus de place en mémoire partagée (un processus *backend* peut nettoyer des pages *dirty* sur disque pour libérer des pages en mémoire partagée, certaines commandes peuvent être plus agressives).

Des métriques similaires sont `local_blks_*` pour les accès à la mémoire du *backend*, pour les objets temporaires (tables temporaires, index sur tables temporaires…). Ces derniers ne nécessitent pas d'être partagés avec les autres sessions.

Les métriques `temp_blks_read` et `temp_blks_written` correspondent au nombre de blocs lus et écris depuis le disque dans des fichiers temporaires. Cela survient par exemple lorsqu'un tri ou le retour d'une fonction multiligne ne rentre pas dans le `work_mem`.

Les métriques finissant par `_time` sont des cumuls des durées de lectures et écritures des accès sur disques. Il faut activer le `track_io_timing` pour qu'elles soient remplies.

6.8.7 Vue pg_stat_statements - métriques 4/5

- Journaux de transactions (v13+) :
  - `wal_records`
  - `wal_fpi`
  - `wal_bytes`

`pg_stat_statements` apporte des statistiques sur les écritures dans les journaux de transactions. `wal_records`, `wal_fpi`, `wal_bytes` correspondent respectivement au nombre d'enregistrements, au nombre de *Full Page Images* (blocs entiers, de 8 ko généralement, écrits intégralement quand un bloc est écrit pour la première fois après un checkpoint), et au nombre d'octets écrits dans les journaux de transactions lors de l'exécution de cette requête.

On peut ainsi suivre les requêtes créant de nombreux journaux.

6.8.8 Vue pg_stat_statements - métriques 5/5

- JIT (v15+)
  - `jit_functions`
  - `jit_generation_time`
  - etc

`pg_stat_statements` apporte des statistiques sur les durées d'optimisation via JIT. Toutes les informations fournies par un `EXPLAIN ANALYZE` sont disponibles dans cette vue. Cette métrique peut être intéressante pour comprendre si JIT améliore bien la durée d'exécution des requêtes.

Liste des colonnes disponibles :

- `jit_functions`
- `jit_generation_time`
- `jit_inlining_count`
- `jit_inlining_time`
- `jit_optimization_count`
- `jit_optimization_time`
- `jit_emission_count`
- `jit_emission_time`

Toutes ces colonnes ne sont disponibles qu'à partir de la version 15.

6.8.9 Requêtes bloquées

- Vue `pg_stat_activity`
  - colonnes `wait_event` et `wait_event_type`
- Vue `pg_locks`
  - colonne `granted`
  - colonne `waitstart` (v14+)
- 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`.

C'est une vue globale à l'instance :

```
\d pg_locks
```

```
                      Vue « pg_catalog.pg_locks »
      Colonne       |   Type                     | … | NULL-able | Par défaut
--------------------+----------------------------+---+-----------+------------
 locktype           | text                       |   |           |
 database           | oid                        |   |           |
 relation           | oid                        |   |           |
 page               | integer                    |   |           |
 tuple              | smallint                   |   |           |
 virtualxid         | text                       |   |           |
 transactionid      | xid                        |   |           |
 classid            | oid                        |   |           |
 objid              | oid                        |   |           |
 objsubid           | smallint                   |   |           |
 virtualtransaction | text                       |   |           |
 pid                | integer                    |   |           |
 mode               | text                       |   |           |
 granted            | boolean                    |   |           |
 fastpath           | boolean                    |   |           |
 waitstart          | timestamp with time zone   |   |           |
```

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 :

```
SELECT pid, query FROM pg_stat_activity
WHERE wait_event_type = 'Lock' AND backend_type='client backend' ;
```

```
  pid  |     query
-------+----------------
 17396 | drop table t2;
```

Pour savoir de quel verrou a besoin le processus 17396, il faut interroger la vue `pg_locks` :

```
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
```

Le processus 17396 attend un verrou sur la relation 24581. Reste à savoir qui dispose d'un verrou sur cet objet :

```
SELECT locktype, relation, pid, mode, granted FROM pg_locks
WHERE relation=24581 AND granted ;
```

```
 locktype | relation |  pid  |      mode       | granted
----------+----------+-------+-----------------+---------
 relation |    24581 | 17276 | AccessShareLock | t
```

Il s'agit du processus 17276. Et que fait ce processus ?

```
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;
```

Nous retrouvons bien notre session en transaction.

Depuis PostgreSQL 9.6, on peut aller plus vite, avec la fonction `pg_blocking_pids()`, qui renvoie les PID des sessions bloquant une session particulière.

```
SELECT pid, pg_blocking_pids(pid)
FROM pg_stat_activity WHERE wait_event IS NOT NULL ;
```

```
  pid  | pg_blocking_pids
-------+------------------
 17396 | {17276}
```

Le processus 17276 bloque bien le processus 17396.

Depuis la version 14, la colonne `waitstart` de la vue `pg_locks` indique depuis combien de temps la session est en attente du verrou.

6.9 PROGRESSION D'UNE REQUÊTE

- API de progression de requêtes
- Utilisé par les commandes SQL
  - `VACUUM` avec `pg_stat_progress_vacuum`
  - `ANALYZE` avec `pg_stat_progress_analyze`
  - `CLUSTER` et `VACUUM FULL` avec `pg_stat_progress_cluster`
  - `CREATE INDEX` et `REINDEX` avec `pg_stat_progress_create_index`
  - `COPY` avec `pg_stat_progress_copy`
- Utilisé par la commande de réplication
  - `BASE BACKUP` avec `pg_stat_progress_basebackup`

La version 9.6 implémente une API pour surveiller la progression de l'exécution d'une requête. Cette API est utilisée par différentes commandes.

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 86 665 blocs (soit 68 % de la table), et en a traité 86 664.

Cette API a ensuite été utilisée pour implémenter avec la version 12 le suivi de l'exécution d'un `CLUSTER` et d'un `VACUUM FULL` avec `pg_stat_progress_cluster`, et celui d'un `CREATE INDEX` et d'un `REINDEX` avec `pg_stat_progress_create_index`. La version 13 a ajouté le suivi d'un `ANALYZE` avec la vue `pg_stat_progress_analyze`. Elle a aussi ajouté le suivi de la commande de réplication `BASE BACKUP` avec `pg_stat_progress_basebackup`. Enfin, la version 14 ajoute le suivi de la commande `COPY` avec la vue `pg_stat_progress_copy`.

6.10 SURVEILLER LES ÉCRITURES

- Quelle quantité de données sont écrites ?
- Quel canal d'écriture est utilisé ?

6.10.1 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.

6.10.2 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.

Certaines colonnes indiquent l'activité du `checkpointer`, afin de vérifier que celui-ci effectue surtout des écritures périodiques, donc bien lissées dans le temps. Les deux premières colonnes notamment

permettent de vérifier que la configuration de `checkpoint_segments` ou `max_wal_size` n'est pas trop basse par rapport au volume d'écriture que subit la base.

- `checkpoints_timed` : nombre de checkpoints déclenchés par `checkpoint_timeout` (périodiques) ;
- `checkpoints_req` : nombre de checkpoints déclenchés par atteinte de `max_wal_size`, donc sous forte charge ;
- `checkpoint_write_time` : temps passé par le `checkpointer` à écrire des données ;
- `checkpoint_sync_time` : temps passé à s'assurer que les écritures ont été synchronisées sur disque lors des checkpoints.

Le `background writer` est destiné à nettoyer le cache de PostgreSQL en complément du `checkpointer`, pour éviter que les backends (processus clients) écrivent eux-mêmes, faute de bloc libérable dans le cache. Il allège aussi la charge du `checkpointer`. Il a des champs dédiés :

- `buffers_checkpoint` : nombre de blocs écrits par `checkpointer` ;
- `buffers_clean` : nombre de blocs écrits par le `background writer` ;
- `maxwritten_clean` : nombre de fois où le `background writer` s'est arrêté pour avoir atteint la limite configurée par `bgwriter_lru_maxpages` ;
- `buffers_backend` : nombre de blocs écrits par les processus backends (faute de buffer disponible en cache) ;
- `buffers_backend_fsync` : nombre de blocs synchronisés par les backends ;
- `buffers_alloc` : nombre de blocs alloués dans les *shared buffers*.

Les colonnes `buffers_clean` (à comparer à `buffers_checkpoint` et `buffers_backend`) et `maxwritten_clean` permettent de vérifier que la configuration 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, les backends vont se mettre à écrire eux-mêmes sur le disque et `buffers_backend` va augmenter. Ce dernier cas n'est pas inquiétant s'il est ponctuel (gros import), mais ne doit pas être fréquent en temps normal, toujours dans le but de lisser les écritures sur le disque.

Il faut toutefois prendre tout cela 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`.

Toutes ces statistiques sont cumulatives. Le champs `stats_reset` indique la date de remise à zéro de cette vue. Pour demander la réinitialisation, utiliser :

```
SELECT pg_stat_reset_shared('bgwriter') ;
```

6.11 SURVEILLER L'ARCHIVAGE ET LA RÉPLICATION

- Sauvegarde PITR & *log shipping* :
  - `pg_stat_archiver`
- Réplication :
  - `pg_stat_replication`
  - `pg_stat_database_conflicts`

6.11.1 pg_stat_archiver

- Bon fonctionnement de l'archivage
- Quand et combien d'erreurs d'archivages se sont produites

Cette vue ne comporte qu'une seule ligne.

- `archived_count` : nombre de WAL archivés ;
- `last_archived_wal` : nom du dernier fichier WAL dont l'archivage a réussi ;
- `last_archived_time` : date du dernier archivage réussi ;
- `failed_count` : nombre de tentatives d'archivages échouées ;
- `last_failed_wal` : nom du dernier fichier WAL qui a rencontré des problèmes d'archivage ;
- `last_failed_time` : date de la dernière tentative d'archivage échouée ;
- `stats_reset` : date de remise à zéro de cette vue statistique.

Cette vue peut être spécifiquement remise à zéro par l'appel à la fonction `pg_stat_reset_shared('archiver')`.

On peut facilement s'en servir pour déterminer si l'archivage fonctionne bien :

```
SELECT case WHEN (last_archived_time > last_failed_time)
    THEN 'OK' ELSE 'KO' END FROM pg_stat_archiver ;
```

6.11.2 pg_stat_replication & pg_stat_database_conflicts

- `pg_stat_replication` :
  - État des serveurs secondaires (*streaming*)
  - Mesure du lag
- `pg_stat_database_conflicts` :
  - nombre de conflits de réplication
  - par type

`pg_stat_replication` permet de suivre les différentes étapes de la réplication.

```
select * from pg_stat_replication \gx
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 16028
usesysid         | 10
usename          | postgres
application_name | secondaire
client_addr      | 192.168.74.16
client_hostname  | *NULL*
client_port      | 52016
backend_start    | 2019-10-28 19:00:16.612565+01
backend_xmin     | *NULL*
state            | streaming
sent_lsn         | 0/35417438
write_lsn        | 0/35417438
flush_lsn        | 0/35417438
replay_lsn       | 0/354160F0
write_lag        | 00:00:00.002626
flush_lag        | 00:00:00.005243
replay_lag       | 00:00:38.09978
sync_priority    | 1
sync_state       | sync
reply_time       | 2019-10-28 19:04:48.286642+0
```

- `pid` : numéro de processus du backend discutant avec le serveur secondaire ;
- `usesysid`, `usename` : OID et nom de l'utilisateur utilisé pour se connecter en streaming replication ;
- `application_name` : *application_name* de la chaîne de connexion du serveur secondaire ; Peut être paramétré dans le paramètre `primary_conninfo` du serveur secondaire, surtout utilisé dans le cas de la réplication synchrone ;
- `client_addr` : adresse IP du secondaire (s'il n'est pas sur la même machine, ce qui est vraisemblable) ;
- `client_hostname` : nom d'hôte du secondaire (si `log_hostname` à `on`) ;

- `client_port` : numéro de port TCP auquel est connecté le serveur secondaire ;
- `backend_start` : timestamp de connexion du serveur secondaire
- `backend_xmin` : l'horizon `xmin` renvoyé par le standby ;
- `state` : `startup` (en cours d'initialisation), `backup` (utilisé par `pg_basebackup`), `catchup` (étape avant streaming, rattrape son retard), `streaming` (on est dans le mode streaming, les nouvelles entrées de journalisation sont envoyées au fil de l'eau) ;
- `sent_lsn` : l'adresse jusqu'à laquelle on a envoyé le contenu du WAL à ce secondaire ;
- `write_lsn` l'adresse jusqu'à laquelle ce serveur secondaire a écrit le WAL sur disque ;
- `flush_lsn` : l'adresse jusqu'à laquelle ce serveur secondaire a synchronisé le WAL sur disque (l'écriture est alors garantie) ;
- `replay_lsn` : l'adresse jusqu'à laquelle le serveur secondaire a rejoué les informations du WAL (les données sont donc visibles jusqu'à ce point, par requêtes, sur le secondaire) ;
- `write_lag` : durée écoulée entre la synchronisation locale sur disque et la réception de la notification indiquant que le standby l'a écrit (mais ni synchronisé ni appliqué) ;
- `flush_lag` : durée écoulée entre la synchronisation locale sur disque et la réception de la notification indiquant que le standby l'a écrit et synchronisé (mais pas appliqué) ;
- `replay_lag` : durée écoulée entre la synchronisation locale sur disque et la réception de la notification indiquant que le standby l'a écrit, synchronisé et appliqué ;
- `sync_priority` : dans le cas d'une réplication synchrone, la priorité de ce serveur (un seul est synchrone, si celui-ci tombe, un autre est promu). Les 3 valeurs 0 (asynchrone), 1 (synchrone) et 2 (candidat) sont traduites dans `sync_state` ;
- `reply_time` : date et heure d'envoi du dernier message de réponse du standby.

`pg_stat_database_conflicts` suit les conflits entre les données provenant du serveur principal et les sessions en cours sur le secondaire :

```
 \d pg_stat_database_conflicts

            Vue « pg_catalog.pg_stat_database_conflicts »
     Colonne      |  Type   | Collationnement | NULL-able | Par défaut
------------------+---------+-----------------+-----------+------------
 datid            | oid     |                 |           |
 datname          | name    |                 |           |
 confl_tablespace | bigint  |                 |           |
 confl_lock       | bigint  |                 |           |
 confl_snapshot   | bigint  |                 |           |
 confl_bufferpin  | bigint  |                 |           |
 confl_deadlock   | bigint  |                 |           |
```

- `datid`, `datname` : l'OID et le nom de la base ;
- `confl_tablespace` : requêtes annulées pour rejouer un `DROP TABLESPACE` ;
- `confl_lock` : requêtes annulées à cause de `lock_timeout` ;
- `confl_snapshot` : requêtes annulées à cause d'un *snapshot* (instantané) trop vieux ; dû à des données supprimées sur le primaire par un VACUUM, rejouées sur le secondaire et y supprimant des données encore nécessaires pour des requêtes (on peut faire disparaître totalement ce cas en activant `hot_standby_feedback`) ;

- `confl_bufferpin` : requêtes annulées à cause d'un `buffer pin`, c'est-à-dire d'un bloc de cache mémoire en cours d'utilisation dont avait besoin la réplication. Ce cas est extrêmement rare : il faudrait un buffer pin d'une durée comparable à `max_standby_archive_delay` ou `max_standby_streaming_delay`. Or ceux-ci sont par défaut à 30 s, alors qu'un buffer pin dure quelques microsecondes ;
- `confl_deadlock` : requêtes annulées à cause d'un deadlock entre une session et le rejeu des transactions (toujours au niveau des buffers). Hautement improbable aussi.

Il est à noter que la version 14 permet de tracer toute attente due à un conflit de réplication. Il suffit pour cela d'activer le paramètre `log_recovery_conflict_waits`.

6.12 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, `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.

6.12.1 pg_activity

- `top` pour PostgreSQL
- Libre, script en python
- Affiche :
  - les requêtes en cours
  - les sessions bloquées
  - les sessions bloquantes
- Dépôt github[11]

`pg_activity` est un projet libre qui apporte une fonctionnalité équivalent à `top`, mais appliqué à PostgreSQL. Il affiche trois écrans qui affichent chacun les requêtes en cours, les sessions bloquées et les sessions bloquantes, avec possibilité de tris, de changer le délai de rafraîchissement, de mettre en pause, d'exporter les requêtes affichées en CSV, etc…

Pour afficher toutes les informations, y compris au niveau système, l'idéal est de se connecter en **root** et superutilisateur **postgres** :

```
sudo -u postgres pg_activity -U postgres
```

6.12.2 pgBadger

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations sur les requêtes
- Génération d'un rapport HTML très détaillé
- Site officiel[12]

pgBadger est un projet sous licence BSD très actif. Le site officiel se trouve sur https://pgbadger.darold.net/.

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.

6.12.3 pgCluu

- Outils de collectes de métriques de performances
  - Dépôt github[13]
  - génère un rapport HTML complet
- Différents aspects mesurés :
  - informations sur le système
  - consommation des ressources CPU, RAM, I/O
  - utilisation de la base de données

6.12.4 PostgreSQL Workload Analyzer

- Objectif : identifier les requêtes coûteuses
  - sans devoir accéder aux logs
  - quasi en temps-réel
- Background worker
  - dépendant de `pg_stat_statements`
- Site officiel[14]

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 ~28 000 `COPY`, ~11 000 `LOCK`
- dont 5048 requêtes applicatives

6.13 CONCLUSION

- Un système est pérenne s'il est bien supervisé
- Les systèmes de supervision automatique ont souvent besoin d'être complétés
- 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.

6.13.1 Questions

N'hésitez pas, c'est le moment !

6.14 QUIZ

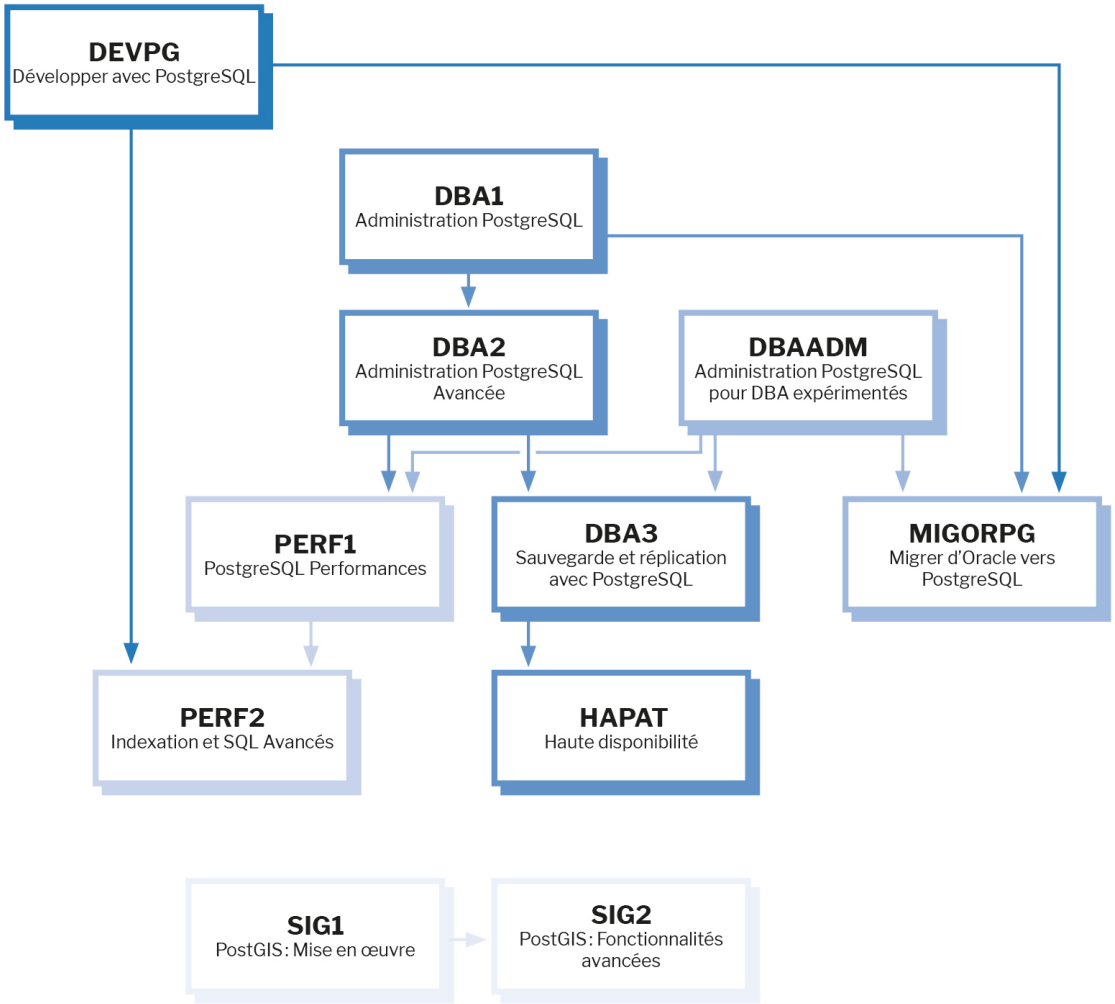
https://dali.bo/h2_quiz

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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