Module H2

# Analyses et diagnostics


25.03

# Table des matières

## Sur ce document

| | |
|---|---|
| **Formation** | Module H2 |
| **Titre** | Analyses et diagnostics |
| **Révision** | 25.03 |
| **PDF** | https://dali.bo/h2_pdf |
| **EPUB** | https://dali.bo/h2_epub |
| **HTML** | https://dali.bo/h2_html |
| **Slides** | https://dali.bo/h2_slides |

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

## Chers lectrices & lecteurs,

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

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

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

## À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

## Remerciements

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

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, 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

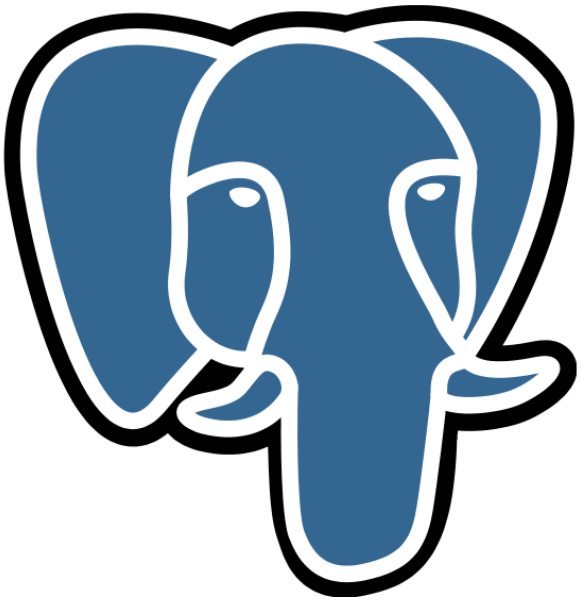
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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/ Analyses et diagnostics



## 1.1 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).

### 1.1.1 Menu

- Supervision occasionnelle système
  - Linux
  - Windows
- Supervision occasionnelle PostgreSQL
- Outils

## 1.2 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.

### 1.2.1 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 don-

nées 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.

### 1.2.2 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`.

### 1.2.3 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**).

### 1.2.4 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*.

### 1.2.5 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
```

### 1.2.6 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].

### 1.2.7 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

## 1.3 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.

### 1.3.1 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.

### 1.3.2 Windows - Process Monitor

- Surveillance des processus
- Filtres
- Récupération de la ligne de commande, identificateur de session et utilisateur
- https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

[2] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tasklist

Process Monitor permet de lister les appels système des processus, comme le montre la copie d'écran ci-dessous :

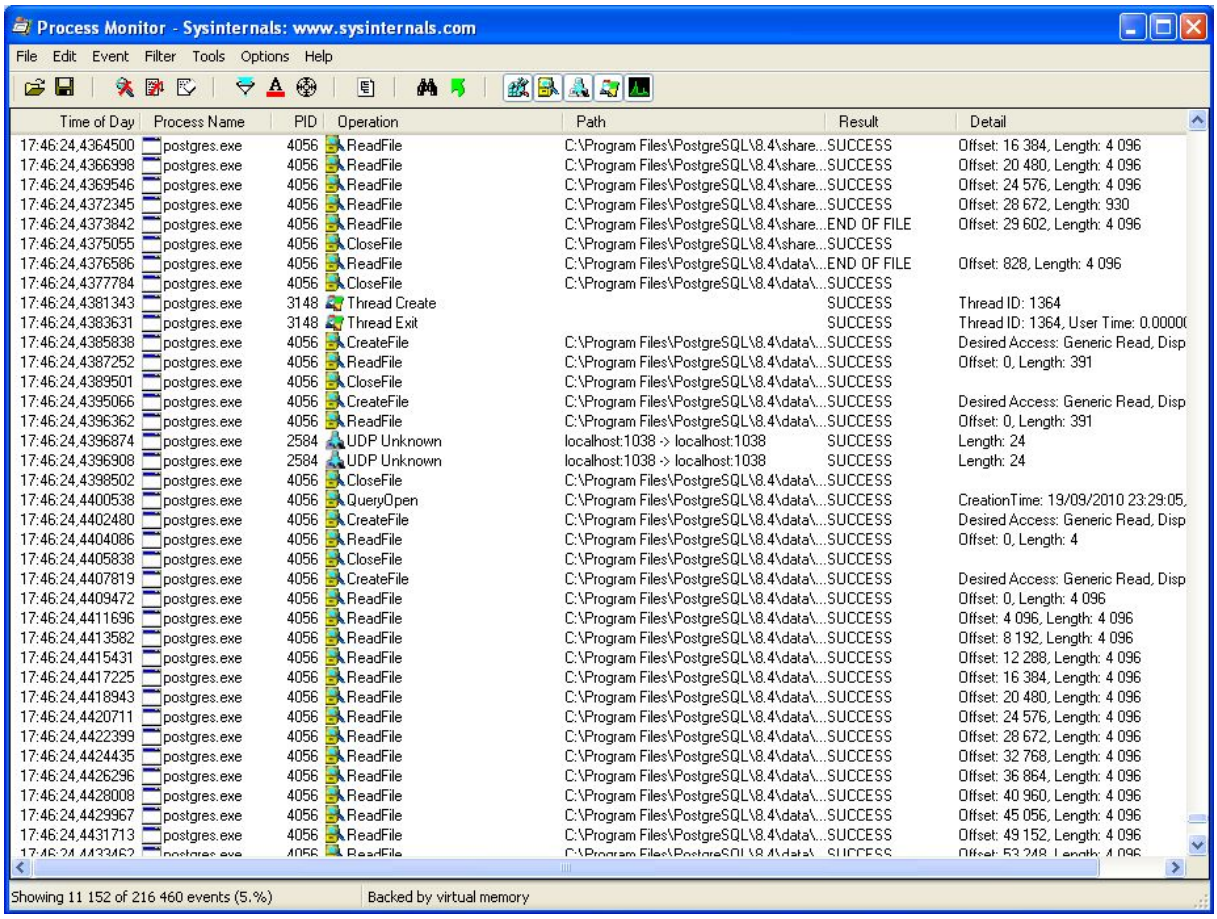



**Figure 1/ .1:** Process Monitor

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.

### 1.3.3 Windows - Process Explorer

- Semblable à `top`
- https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

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.

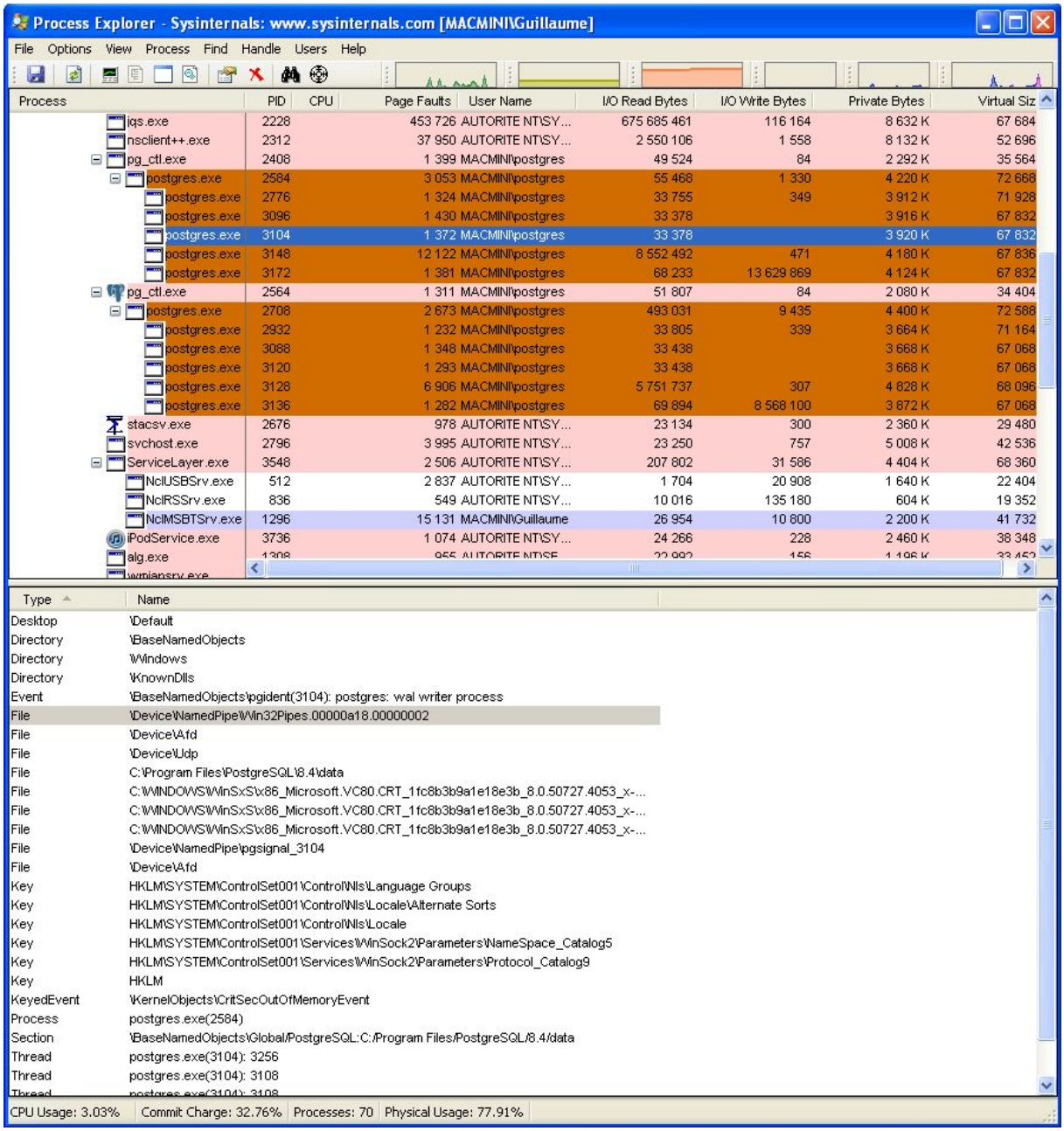



**Figure 1/ .2:** Process Explorer

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 :

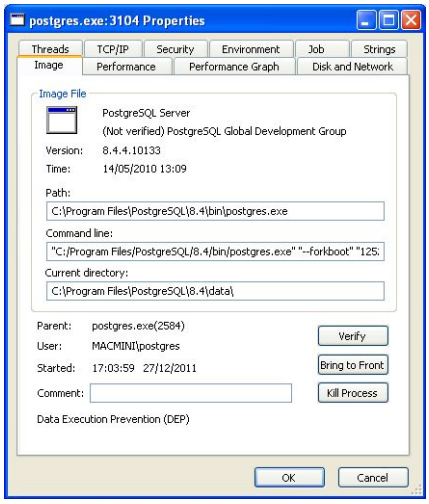



**Figure 1/ .3:** Process Explorer

- le deuxième, « Performances » fournit des informations textuelles sur les performances :

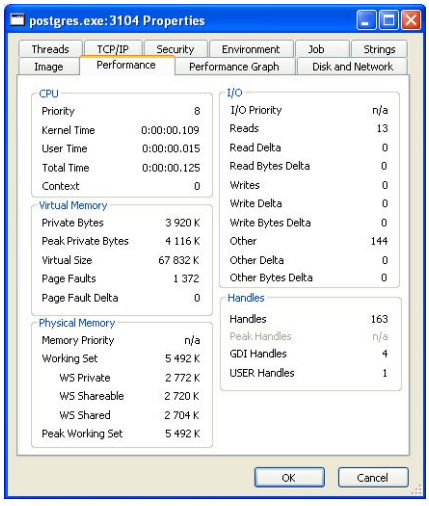



**Figure 1/ .4:** Process Explorer

- le troisième affiche quelques graphes :

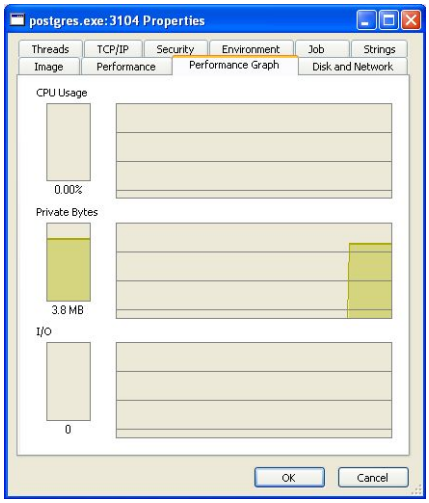



**Figure 1/ .5:** Process Explorer

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.

### 1.3.4 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.

## 1.4 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.

### 1.4.1 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.

## 1.5 GÉRER LES CONNEXIONS

- qui est connecté ?
- qui fait quoi ?
- qui est bloqué ?
- qui bloque les autres ?
- comment arrêter une requête ?

### 1.5.1 Vue pg_stat_activity

- Liste des processus
  - sessions (*backends*)
  - processus en tâche de fond
- Requête en cours/dernière exécutée
  - `query_id`
- *idle in transaction* : attention !
- Sessions en attente : *wait_events*

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche. Elle donne la liste des processus en cours sur l'instance, en incluant entre autres :

- le numéro de processus sur le serveur (`pid`) ;
- la base de données, le nom dutilisateur, l'adresse et le port du client ;
- les dates de début d'ordre, de transaction ou de session ;
- son statut (active ou non) ;
- la requête en cours, ou la dernière requête si la session ne fait rien ;
- le nom de l'application s'il a été renseigné avec le paramètre `application_name` ;
- le type de processus : session d'un utilisateur (*client backend*), processus interne…

```
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…
-[ 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…
-[ 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…
-[ 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…
-[ 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             |
```

Les textes des requêtes sont tronqués à 1024 caractères : c'est un problème courant. Il est conseillé de monter le paramètre `track_activity_query_size` à plusieurs kilooctets.

Cette vue fournit aussi les *wait events*, qui indiquent ce qu'une session est en train d'attendre. Cela peut être très divers et inclut la levée d'un verrou sur un objet, celle d'un verrou interne, la fin d'une entrée-sortie… L'absence de *wait event* indique que la requête s'exécute. À noter qu'une session avec un *wait event* peut rester en statut `active`.

Les détails sur les champs `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente) sont disponibles dans le tableau des événements d'attente[3]. de la documentation.

À partir de PostgreSQL 17, la vue `pg_wait_events` peut être directement jointe à `pg_stat_activity`, et son champ `description` évite d'aller voir la documentation :

```
SELECT datname, application_name, pid,
       wait_event_type, wait_event, query, w.description
FROM pg_stat_activity a
  LEFT OUTER JOIN pg_wait_events w
  ON (a.wait_event_type = w.type AND a.wait_event = w.name)
WHERE backend_type='client backend'
AND   wait_event IS NOT NULL
ORDER BY wait_event DESC LIMIT 4 \gx

-[ RECORD 1 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786146
wait_event_type  | LWLock
wait_event       | WALWriteLock
```

[3] https://docs.postgresql.fr/current/monitoring-stats.html#WAIT-EVENT-TABLE

```
query            | UPDATE pgbench_accounts SET abalance = abalance + 4055 WHERE…
description      | ø
-[ RECORD 2 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786190
wait_event_type  | IO
wait_event       | WalSync
query            | UPDATE pgbench_accounts SET abalance = abalance + -1859  WHERE…
description      | Waiting for a WAL file to reach durable storage
-[ RECORD 3 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786145
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 3553 WHERE…
description      | Waiting for a read from a relation data file
-[ RECORD 4 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786143
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 1929 WHERE…
description      | Waiting for a read from a relation data file
```

Le processus de la ligne 2 attend une synchronisation sur disque du journal de transaction (WAL), et les deux suivants une lecture d'un fichier de données. (La description vide en ligne 1 est un souci de la version 17.2).

Pour entrer dans le détail des champs liés aux connexions :

- `backend_type` est le type de processus : on filtrera généralement sur `client backend`, mais on y trouvera aussi des processus de tâche de fond comme `checkpointer`, `walwriter`, `autovacuum launcher` et autres processus de PostgreSQL, ou encore des *workers* lancés par des extensions ;
- `datname` est le nom de la base à laquelle la session est connectée, et `datid` est son identifiant (OID) ;
- `pid` est le processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client, qui durera le temps de la session (sauf parallélisation) ;
- `usename` est le nom de l'utilisateur connecté, et `usesysid` est son OID dans `pg_roles` ;
- `application_name` est un nom facultatif, et il est recommandé que l'application cliente le renseigne autant que possible avec `SET application_name TO 'nom_outil_client'` ;
- `client_addr` est l'adresse IP du client connecté (`NULL` si connexion sur socket Unix), et `client_hostname` est le nom associé à cette IP, renseigné uniquement si `log_hostname` a été passé à `on` (cela peut ralentir les connexions à cause de la résolution DNS) ;
- `client_port` est le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP.

Une requête parallélisée occupe plusieurs processus, et apparaîtra sur plusieurs lignes de `pid` différents. Le champ `leader_pid` indique le processus principal. Les autres processus disparaîtront dès la requête terminée.

Pour les champs liés aux durées de session, transactions et requêtes :

- `backend_start` est le timestamp de l'établissement de la session ;
- `xact_start` est le timestamp de début de la transaction ;
- `query_start` est le timestamp de début de la requête en cours, ou de la dernière requête exécutée ;
- `status` vaut soit `active`, soit `idle` (la session ne fait rien) soit `idle in transaction` (en attente pendant une transaction) ;
- `backend_xid` est l'identifiant de la transaction en cours, s'il y en a une ;
- `backend_xmin` est l'horizon des transactions visibles, et dépend aussi des autres transactions en cours.

Rappelons qu'une session durablement en statut `idle in transaction` bloque le fonctionnement de l'autovacuum car `backend_xmin` est bloqué. Cela peut mener à des tables fragmentées et du gaspillage de place disque.

Depuis PostgreSQL 14, `pg_stat_activity` peut afficher un champ `query_id`, c'est-à-dire un identifiant de requête normalisée (dépouillée des valeurs de paramètres). Il faut que le paramètre `compute_query_id` soit à `on` ou `auto` (le défaut, et alors une extension peut l'activer). Ce champ est utile pour retrouver une requête dans la vue de l'extension `pg_stat_statements`, par exemple.

Certains champs de cette vue ne sont renseignés que si le paramètre `track_activities` est à `on` (valeur par défaut, qu'il est conseillé de laisser ainsi).

À noter qu'il ne faut pas interroger `pg_stat_activity` au sein d'une transaction, son contenu pourrait sembler figé.

### 1.5.2 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.

### 1.5.3 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 ;

## 1.6 VERROUS

- Visualisation des verrous en place
- Tous types de verrous sur objets
- Complexe à interpréter
  - verrous sur enregistrements pas directement visibles
  - https://kb.dalibo.com/verrouillage

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`).

### 1.6.1 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.

### 1.6.2 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.

## 1.7 SURVEILLER L'ACTIVITÉ SUR LES TABLES

- Quelle taille font mes objets ?
- Quel est leur taux de fragmentation ?
- Comment sont-ils accédés ?

### 1.7.1 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[4] contient d'autres exemples, notamment sur le calcul de la taille totale d'une table partitionnée.

[4] https://wiki.postgresql.org/wiki/Disk_Usage

### 1.7.2 Mesurer la fragmentation des objets

- Fragmentation induite par MVCC
  - tables et index
- Mesure précise de la fragmentation :
  - extension `pgstattuple`
- Estimer la fragmentation :
  - https://github.com/ioguix/pgsql-bloat-estimation
  - `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[5]. Celle-ci par contre est susceptible de lire toute la table, ce qui est donc long.

[5] https://docs.postgresql.fr/current/pgstattuple.html

Il existe une fonction `pgstattuple()` pour les tables et index, et une fonction `pgstatindex()` plus précise pour les index.

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.

### 1.7.3 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.

### 1.7.4 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.

### 1.7.5 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.

### 1.7.6 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[6].

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 :

[6] https://docs.postgresql.fr/16/monitoring-stats.html#MONITORING-PG-STAT-IO-VIEW

```
SELECT backend_type, object, context, reads, read_time
FROM   pg_stat_io
ORDER BY read_time DESC NULLS LAST LIMIT 3 ;
```

```
   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 …`).

## 1.8 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

### 1.8.1 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`.

### 1.8.2 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.

### 1.8.3 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.

### 1.8.4 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.

### 1.8.5 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.

### 1.8.6 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.

### 1.8.7 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.

### 1.8.8 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.

### 1.8.9 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.

## 1.9 PROGRESSION DE CERTAINES COMMANDES

| Opération | Vue de suivi | PostgreSQL |
|---|---|---|
| `VACUUM` | `pg_stat_progress_vacuum` | 9.6 |
| `ANALYZE` | `pg_stat_progress_analyze` | 13 |
| `CLUSTER` | `pg_stat_progress_cluster` | 12 |
| `VACUUM FULL` | `pg_stat_progress_cluster` | 12 |
| `CREATE INDEX` | `pg_stat_progress_create_index` | 12 |
| `BASE BACKUP` | `pg_stat_progress_basebackup` | 13 |
| `COPY` | `pg_stat_progress_copy` | 14 |

Il est 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 :

```
TABLE pg_stat_progress_vacuum \gx
```

```
-[ RECORD 1 ]------+--------------
pid                | 2603780
datid              | 1308955
datname            | pgbench_100
relid              | 1308962
phase              | scanning heap
heap_blks_total    | 163935
heap_blks_scanned  | 3631
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes   | 0
num_dead_item_ids  | 0
indexes_total      | 0
indexes_processed  | 0
```

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.

Noter que le suivi du `VACUUM` dans les index (les deux derniers champs) nécessite au moins PostgreSQL 17. C'est souvent la partie la plus longue d'un `VACUUM`.

Au fil des versions de PostgreSQL, sont apparues des vues similaires pour suivre les ordres `ANALYZE`, `VACUUM FULL`, `CLUSTER`, les (ré)indexations, les *base backups* (par exemple avec `pg_basebackup`), ou des insertions avec `COPY`.

Ces vues n'affichent que les opérations en cours, elles n'historisent rien. Si aucun de ces ordres n'est en cours, elles n'afficheront rien.

## 1.10 PROGRESSION D'UNE REQUÊTE

– Hélas, non

Hélas, PostgreSQL ne permet pas suivre le déroulement d'une requête. Même avec `auto_explain`, il faut attendre sa fin pour avoir le plan d'exécution.

Cette lacune est effectivement gênante. Il existe des extensions ou projets plus ou moins expérimentaux, ou avec un impact notable en performance.

## 1.11 SURVEILLER LES ÉCRITURES

- Quelle quantité de données sont écrites ?
- Quel canal d'écriture est utilisé ?

### 1.11.1 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.

### 1.11.2 Vues pg_stat_bgwriter & pg_stat_checkpointer

- `pg_stat_bgwriter`
- `pg_stat_checkpointer` (v17)
- Activité des écritures dans les fichiers de données
- Visualisation du volume d'allocations et d'écritures

Cette vue ne comporte qu'une seule ligne.

`pg_stat_bgwriter` stocke les statistiques d'écriture des buffers des *background writer*, et du *checkpointer* (jusqu'en version 16 incluse) et des sessions elles-mêmes. On peut ainsi voir si les *backends* écrivent beaucoup ou peu. À partir de PostgreSQL 17, apparaît `pg_stat_checkpointer` qui reprend les champs sur les *checkpoints* et en ajoute quelques-uns. Cette vue permet de vérifier que les *checkpoints* sont réguliers, donc peu gênants.

Exemple (version 17) :

```
TABLE pg_stat_bgwriter \gx

-[ RECORD 1 ]----+------------------------------
buffers_clean    | 3004
maxwritten_clean | 26
buffers_alloc    | 24399160
stats_reset      | 2024-11-05 15:12:27.556173+01

TABLE pg_stat_checkpointer \gx

-[ RECORD 1 ]-------+------------------------------
num_timed           | 282
num_requested       | 2
restartpoints_timed | 0
restartpoints_req   | 0
restartpoints_done  | 0
write_time          | 605908
sync_time           | 3846
buffers_written     | 20656
stats_reset         | 2024-11-05 15:12:27.556173+01
```

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 `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') ;
SELECT pg_stat_reset_shared('checkpointer') ;
```

## 1.12 SURVEILLER L'ARCHIVAGE ET LA RÉPLICATION

- Sauvegarde PITR & *log shipping* :
  - `pg_stat_archiver`
- Réplication :
  - `pg_stat_replication`
  - `pg_stat_database_conflicts`

### 1.12.1 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 ;
```

### 1.12.2 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`.

## 1.13 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.

### 1.13.1 pg_activity

- `top` pour PostgreSQL
- Libre, script en python
- Affiche :
  - les requêtes en cours
  - les sessions bloquées
  - les sessions bloquantes
- https://github.com/dalibo/pg_activity/

`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
```

### 1.13.2 pgBadger

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations sur les requêtes
- Génération d'un rapport HTML très détaillé
- https://pgbadger.darold.net/

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.

### 1.13.3 pgCluu

- Outils de collectes de métriques de performances
  - https://github.com/darold/pgcluu
  - 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

### 1.13.4 PostgreSQL Workload Analyzer

- Objectif : identifier les requêtes coûteuses
  - sans devoir accéder aux logs
  - quasi en temps-réel
- Background worker
  - dépendant de `pg_stat_statements`
- https://github.com/powa-team

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

## 1.14 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.

### 1.14.1 Questions

N'hésitez pas, c'est le moment !

## 1.15 QUIZ

**https://dali.bo/h2_quiz**

## 1.16 TRAVAUX PRATIQUES : ANALYSE DE TRACES AVEC PGBADGER

### 1.16.1 Installation

**But** : Installation & utilisation de pgBadger

#### 1.16.1.1 Installer pgBadger

On peut installer pgBadger soit depuis les dépôts du PGDG, soit depuis le site de l'auteur https://pgbadger.darold.net/.

Le plus simple reste le dépôt du PGDG associé à la distribution :

```
$ sudo dnf install pgbadger
```

Comme Gilles Darold fait évoluer le produit régulièrement, il n'est pas rare que le dépôt Github soit plus à jour, et l'on peut préférer cette source. La release 11.8 est la dernière au moment où ceci est écrit.

```
$ wget https://github.com/darold/pgbadger/archive/v11.8.tar.gz
$ tar xvf v11.8.tar.gz
```

Dans le répertoire `pgbadger-11.8`, il n'y a guère que le script `pgbadger` dont on ait besoin, et que l'on placera par exemple dans `/usr/local/bin`.

On peut même utiliser un simple `git clone` du dépôt. Il n'y a pas de phase de compilation.

#### 1.16.1.2 Récupérer les traces à analyser

Elles sont disponibles sur : https://public.dalibo.com/workshop/workshop_supervision/logs_postgresql.tgz.

L'archive contient 9 fichiers de traces de 135 Mo chacun :

```
$ tar xzf logs_postgresql.tgz
$ cd logs_postgresql
$ du -sh *
135M    postgresql-11-main.1.log
135M    postgresql-11-main.2.log
135M    postgresql-11-main.3.log
135M    postgresql-11-main.4.log
135M    postgresql-11-main.5.log
135M    postgresql-11-main.6.log
135M    postgresql-11-main.7.log
135M    postgresql-11-main.8.log
135M    postgresql-11-main.9.log
```

### 1.16.2 Générer et étudier des rapports pgBadger

**But** : Apprendre à générer et analyser des rapports pgBadger.

#### 1.16.2.1 Premier rapport

Créer un premier rapport sur le premier fichier de traces : `pgbadger -j 4 postgresql-11-main.1.log`.

Lancer tout de suite en arrière-plan la création du rapport complet : `pgbadger -j 4 --outfile rapport_complet`

Pendant ce temps, ouvrir le fichier *out.html* dans votre navigateur. Parcourir les différents onglets et graphiques. Que montrent les onglets *Connections* et *Sessions* ?

Que montre l'onglet *Checkpoints* ?

Que montre l'onglet *Temp Files* ?

Que montre l'onglet *Vacuums* ?

Que montre l'onglet *Locks* ?

Que montre l'onglet *Queries* ?

Que montre l'onglet *Top* dans *Time consuming queries* et *Normalized slowest queries* ? Quelle est la différence entre les différents ensemble de requêtes présentés ?

#### 1.16.2.2 Étude du rapport complet

Une fois la génération de `rapport_complet.html` terminée, l'ouvrir. Chercher à quel moment et sur quelle base sont apparus principalement des problèmes d'attente de verrous.

Créer un rapport `rapport_bank.html` ciblé sur les 5 minutes avant et après 16h50, pour cette base de données. Retrouver les locks et identifier la cause du verrou dans les requêtes les plus lentes.

> Nous voulons connaître plus précisément les requêtes venant de l'IP 192.168.0.89 et avoir une vue plus fine des graphiques. Créer un rapport `rapport_host_89.html` sur cette IP avec une moyenne par minute.

#### 1.16.2.3 Mode incrémental de pgBadger

> Créer un rapport incrémental (sans HTML) dans `/tmp/incr_report` à partir du premier fichier avec : `pgbadger -j 4 -I --noreport -O /tmp/incr_report/ postgresql-11-main.1.log`
> Que contient le répertoire ?

> Quelle est la taille de ce rapport incrémental ?

> Ajouter les rapports incrémentaux avec le rapport HTML sur les 2 premiers fichiers de traces. Quel rapport obtient-on ?

## 1.17 TRAVAUX PRATIQUES : ANALYSE DE TRACES AVEC PGBADGER (SOLUTION)

### 1.17.1 Installation

Voir l'énoncé plus haut.

### 1.17.2 Générer et étudier des rapports pgBadger

#### 1.17.2.1 Premier rapport

> Créer un premier rapport sur le premier fichier de traces : `pgbadger -j 4 postgresql-11-main.1.log`.

Nous allons commencer par créer un premier rapport à partir du premier fichier de logs. L'option `-j` est à fixer à votre nombre de processeurs :

```
$ pgbadger -j 4 postgresql-11-main.1.log
```

Le fichier de rapport *out.html* est créé dans le répertoire courant. Avant de l'ouvrir dans le navigateur, lançons la création du rapport complet :

> Lancer tout de suite en arrière-plan la création du rapport complet : `pgbadger -j 4 --outfile rapport_complet`

La ligne de commande suivante génère un rapport sur tous les fichiers disponibles :

```
$ pgbadger -j 4 --outfile rapport_complet.html postgresql-11-main.*.log
```

> Pendant ce temps, ouvrir le fichier *out.html* dans votre navigateur. Parcourir les différents onglets et graphiques. Que montrent les onglets *Connections* et *Sessions* ?

On peut observer dans les sections *Connections* et *Sessions* un nombre de sessions et de connexions proches. Chaque session doit ouvrir une nouvelle connexion. Ceci est assez coûteux, un processus et de la mémoire devant être alloués.

> Que montre l'onglet *Checkpoints* ?

La section *Checkpoints* indique les écritures des *checkpointers* et *background writer*. Ils ne s'apprécient que sur une durée assez longue.

> Que montre l'onglet *Temp Files* ?

La section *Temp Files* permet, grâce au graphique temporel, de vérifier si un ralentissement de l'instance est corrélé à un volume important d'écriture de fichiers temporaires. Le rapport permet également de lister les requêtes ayant généré des fichiers temporaires. Suivant les cas, on pourra tenter une optimisation de la requête ou bien un ajustement de la mémoire de travail, `work_mem`.

> Que montre l'onglet *Vacuums* ?

La section *Vacuums* liste les différentes tables ayant fait l'objet d'un `VACUUM`.

> Que montre l'onglet *Locks* ?

Le section *Locks* permet d'obtenir les requêtes normalisées ayant le plus fait l'objet d'attente sur verrou. Le rapport pgBadger ne permet pas toujours de connaître la raison de ces attentes.

> Que montre l'onglet *Queries* ?

La section *Queries* fournit une connaissance du type d'activité sur chaque base de données : *application web*, OLTP, *data warehouse*. Elle permet également, si le paramètre `log_line_prefix` le précise bien, de connaître la répartition des requêtes selon la base de données, l'utilisateur, l'hôte ou l'application.

> Que montre l'onglet *Top* dans *Time consuming queries* et *Normalized slowest queries* ? Quelle est la différence entre les différents ensemble de requêtes présentés ?

La section *Top* est très intéressante. Elle permet de lister les requêtes les plus lentes unitairement, mais surtout celles ayant pris le plus de temps, en cumulé et en moyenne par requête.

Avoir fixé le paramètre `log_min_duration_statement` à 0 permet de lister toutes les requêtes exécutées. Une requête peut ne mettre que quelques dizaines de millisecondes à s'exécuter et sembler unitairement très rapide. Mais si elle est lancée des millions de fois par heure, elle peut représenter une charge très conséquente. Elle est donc la première requête à optimiser.

Par comparaison, une grosse requête lente passant une fois par jour participera moins à la charge de la machine, et sa durée n'est pas toujours réellement un problème.

#### 1.17.2.2 Étude du rapport complet

> Une fois la génération de `rapport_complet.html` terminée, l'ouvrir. Chercher à quel moment et sur quelle base sont apparus principalement des problèmes d'attente de verrous.

La vue des verrous nous informe d'un problème sur la base de données *bank* vers 16h50.

> Créer un rapport `rapport_bank.html` ciblé sur les 5 minutes avant et après 16h50, pour cette base de données. Retrouver les locks et identifier la cause du verrou dans les requêtes les plus lentes.

Nous allons réaliser un rapport spécifique sur cette base de données et cette période :

```
$ pgbadger -j 4 --outfile rapport_bank.html --dbname bank        \
   --begin "2018-11-12 16:45:00" --end "2018-11-12 16:55:00"  \
   postgresql-11-main.*.log
```

L'onglet *Top* affiche moins de requête, et la requête responsable du verrou de 16h50 saute plus rapidement aux yeux que dans le rapport complet :

```
VACUUM ( FULL, FREEZE);
```

> Nous voulons connaître plus précisément les requêtes venant de l'IP 192.168.0.89 et avoir une vue plus fine des graphiques. Créer un rapport `rapport_host_89.html` sur cette IP avec une moyenne par minute.

Nous allons créer un rapport en filtrant par client et en calculant les moyennes par minute (le défaut est de 5) :

```
$ pgbadger -j 4 --outfile rapport_host_89.html --dbclient 192.168.0.89 \
   --average 1 postgresql-11-main.*.log
```

Il est également possible de filtrer par application avec l'option `--appname`.

#### 1.17.2.3 Mode incrémental de pgBadger

Les fichiers de logs sont volumineux. On ne peut pas toujours conserver un historique assez important. pgBadger peut parser les fichiers de log et stocker les informations dans des fichiers binaires. Un rapport peut être construit à tout moment en précisant les fichiers binaires à utiliser.

> Créer un rapport incrémental (sans HTML) dans `/tmp/incr_report` à partir du premier fichier avec : `pgbadger -j 4 -I --noreport -O /tmp/incr_report/ postgresql-11-main.1.log`
> Que contient le répertoire ?

Le résultat est le suivant :

```
$ mkdir /tmp/incr_report
$ pgbadger -j 4 -I --noreport -O /tmp/incr_report/ postgresql-11-main.1.log

$ tree /tmp/incr_report
/tmp/incr_report
├── 2018
│   └── 11
│       └── 12
│           ├── 2018-11-12-25869.bin
│           ├── 2018-11-12-25871.bin
│           ├── 2018-11-12-25872.bin
│           └── 2018-11-12-25873.bin
└── LAST_PARSED

3 directories, 5 files
```

Le fichier `LAST_PARSE` stocke la dernière ligne analysée :

```
$ cat /tmp/incr_report/LAST_PARSED
2018-11-12 16:36:39 141351476   2018-11-12 16:36:39 CET [17303]: user=banquier,
  db=bank,app=gestion,client=192.168.0.84 LOG:  duration: 0.2
```

Dans le cas d'un fichier de log en cours d'écriture, pgBadger commencera son analyse suivante à partir de cette date.

> Quelle est la taille de ce rapport incrémental ?

Le fichier `postgresql-11-main.1.log` occupe 135 Mo. On peut le compresser pour le réduire à 7 Mo. Voyons l'espace occupé par les fichiers incrémentaux de pgBadger :

```
$ mkdir /tmp/incr_report
$ pgbadger -j 4 -I --noreport -O /tmp/incr_report/ postgresql-11-main.1.log
$ du -sh /tmp/incr_report/
340K    /tmp/incr_report/
```

On pourra reconstruire à tout moment les rapports avec la commande :

```
$ pgbadger -I -O /tmp/incr_report/ --rebuild
```

Ce mode permet de construire des rapports réguliers, journaliers et hebdomadaires. Vous pouvez vous référer à la documentation[7] pour en savoir plus sur ce mode incrémental.

> Ajouter les rapports incrémentaux avec le rapport HTML sur les 2 premiers fichiers de traces. Quel rapport obtient-on ?

Il suffit d'enlever l'option `--noreport` :

```
$ pgbadger -j 4 -I  -O /tmp/incr_report/ postgresql-11-main.1.log
 ↪  postgresql-11-main.2.log
[==========================>] Parsed 282702952 bytes of 282702952 (100.00%),
                             queries: 7738842, events: 33
LOG: Ok, generating HTML daily report into /tmp/incr_report//2018/11/12/...
LOG: Ok, generating HTML weekly report into /tmp/incr_report//2018/week-46/...
LOG: Ok, generating global index to access incremental reports...
```

Les rapports obtenus sont ici quotidiens et hebdomadaires :

```
$ tree /tmp/incr_report
/tmp/incr_report
├── 2018
│   ├── 11
│   │   └── 12
│   │       ├── 2018-11-12-14967.bin
│   │       ├── 2018-11-12-17227.bin
│   │       ├── 2018-11-12-18754.bin
│   │       ├── 2018-11-12-18987.bin
│   │       ├── 2018-11-12-18993.bin
│   │       ├── 2018-11-12-18996.bin
│   │       ├── 2018-11-12-19002.bin
│   │       ├── 2018-11-12-22821.bin
│   │       ├── 2018-11-12-3633.bin
│   │       ├── 2018-11-12-3634.bin
```

[7] http://pgbadger.darold.net/documentation.html#INCREMENTAL-REPORTS

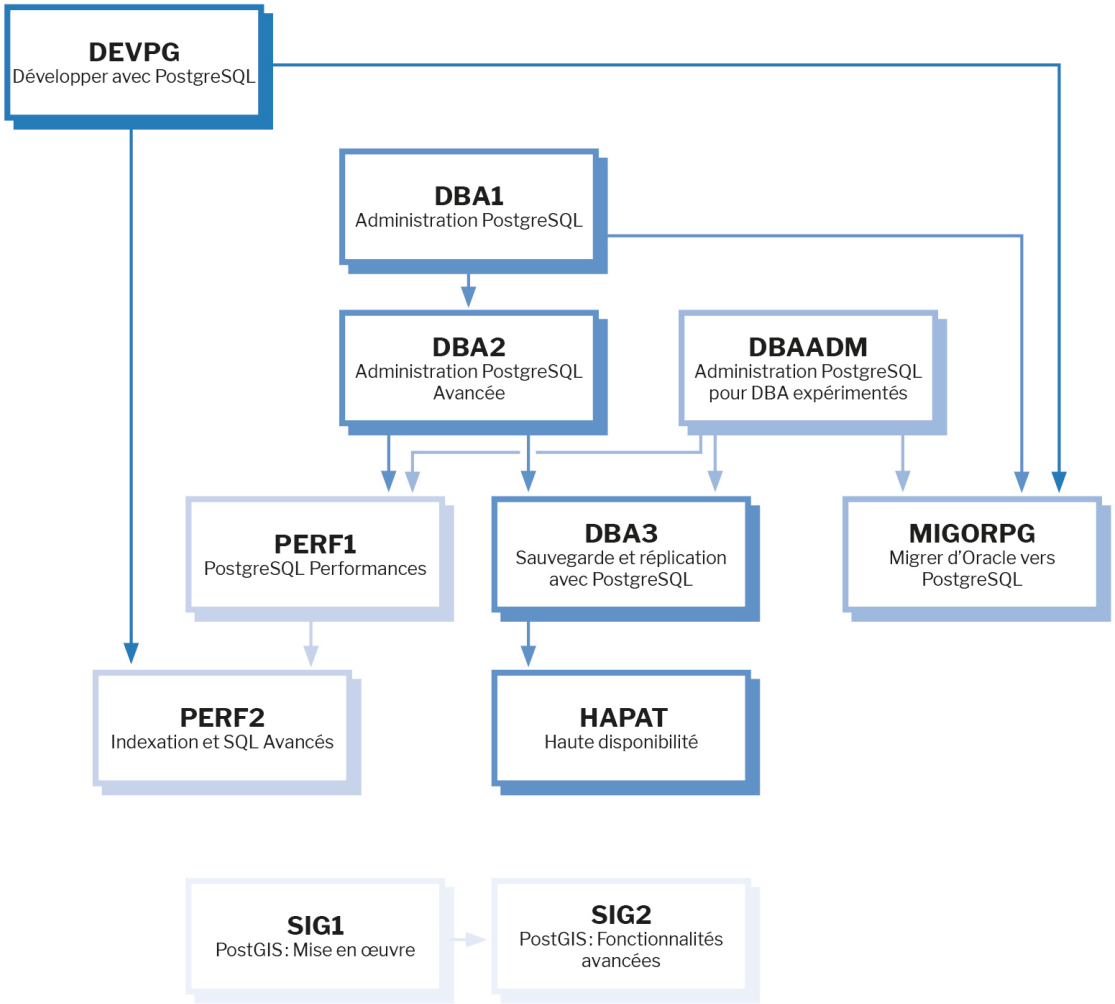
```
│   │       ├── 2018-11-12-3635.bin
│   │       ├── 2018-11-12-3636.bin
│   │       └── index.html
│   └── week-46
│       └── index.html
├── index.html
└── LAST_PARSED
```

# 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.