Table des matières

- Les processus de PostgreSQL

# Les processus de PostgreSQL

*Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 112 (Janvier 2009). Il est disponible maintenant sous licence Creative Commons.*


PostgreSQL n'est pas multi-threadé. Lors de sa conception, les développeurs de PostgreSQL ont préféré utiliser une approche basée sur les processus, et non pas sur les threads. Cela a une implication importante, par exemple pour la gestion de la mémoire. Mais même sans cela, il est nécessaire de bien comprendre l'intérêt des différents processus pour bien gérer son serveur.

L'outil permettant de lancer PostgreSQL s'appelle pg_ctl. Il est généralement appelé par le script de démarrage compris dans le répertoire /etc/init.d, mais il est aussi exécutable manuellement. Ce programme exécute un autre programme, appelé postgres. Il s'agit du processus père de tous les autres processus du serveur PostgreSQL. Très souvent, le nom postmaster est utilisé. Le programme existe même dans les paquets de PostgreSQL, mais il s'agit souvent que d'un lien symbolique vers l'exécutable postgres. Voici ce qu'il se passe quand on exécute la commande pg_ctl :

```
guillaume@laptop$ ps xfo args | grep [p]ostgres
guillaume@laptop$ pg_ctl start
serveur en cours de démarrage
LOG:  le système de bases de données a été arrêté à 2008-10-16 16:22:10 CEST
LOG:  lancement du processus autovacuum
LOG:  le système de bases de données est prêt pour accepter les connexions
guillaume@laptop$ ps xfo args | grep [p]ostgres
/opt/postgresql-8.3/bin/postgres
 \_ postgres: writer process
 \_ postgres: wal writer process
 \_ postgres: autovacuum launcher process
 \_ postgres: stats collector process
```

Nous apercevons par défaut cinq processus : postgres, « writer process », « wal writer process », « autovacuum launcher process » et « stats collector process ». Nous allons détailler chacun de ces processus ainsi que quelques autres.

## Processus d'écoute des connexions

Le premier processus est considéré comme le démon principal de PostgreSQL. Il est le père de tous les autres processus du serveur PostgreSQL. Son but principal est d'écouter toutes les connexions entrantes, soit par la socket soit par le port TCP/IP. Il est souvent dénommé postmaster.

Au démarrage, il charge la configuration, réalise tout un ensemble de tests, supprime des fichiers temporaires qui auraient été laissés dans le répertoire des données de PostgreSQL lors d'une précédente exécution, crée un fichier verrou dans le répertoire des données, alloue la mémoire partagée, initialise certains structures et ainsi de suite. S'il ne peut obtenir cette mémoire, le serveur complet s'arrête. Cela peut arriver par manque mémoire ou par une mauvaise configuration des paramètres système SHMALL et SHMMAX. Dans ce cas, le système d'exploitation refuse d'accorder la mémoire partagée demandée par postmaster.

Par contre, si l'allocation réussit, il va lancer les différents services nécessaire à PostgreSQL dans cet ordre :

- le processus de gestion des journaux applicatifs (si activé) ;
- le processus de collecte des statistiques (si activé) ;
- le « autovacuum launcher » (si activé) ;
- le processus d'écriture en tâche de fond ;
- le processsus d'écriture des journaux de transactions en tâche de fond.

Quand une demande de connexion arrive, un processus fils est immédiatement créé. Ce dernier est responsable de l'identification et l'authentification du client et, si tout va bien, est chargé de la communication client/serveur.

Bien qu'allouer par lui, postmaster n'utilise pas la mémoire partagée. Cela le rend très robuste, et permet d'en faire le père de tous les processus, même dans les cas où il aurait été plus logique que le père soit un autre processus. En effet, en cas de crash des processus fils, il est capable de nettoyer le système en supprimant la mémoire partagée.

## Processus d'écriture en tâche de fond

## Écriture des fichiers de données

Ce processus, disponible depuis la version 8.0, s'occupe de l'écriture des blocs modifiés en mémoire cache dans les fichiers de données.

Avant la version 8.0, les processus postgres étaient chargés des écritures. Cela se passait ainsi : quand un processus avait besoin de place dans le cache disque et que les blocs utilisables étaient des blocs modifiées, il effectuait d'abord l'écriture des blocs dans les fichiers de données, puis pouvait utiliser les blocs libérés. Du coup, la requête réellement exécutée mettait beaucoup de temps à donner un résultat.

Pour améliorer l'interactivité, il a été décidé de créer un processus chargé des écritures. Ce processus, aussi appelé bgwriter pour background writer, va de temps à autre enregistrer les blocs modifiés du cache disque dans les fichiers de données. Cette action d'enregistrement est généralement déclenchée par un checkpoint. Dans le meilleur des cas, toutes les écritures passeront par ce nouveau processus mais il faut savoir que les processus postgres peuvent toujours écrire si le processus d'écriture en tâche de fond n'arrive pas à tenir le rythme.

Le processus bgwriter est exécuté au lancement du serveur PostgreSQL et vit jusqu'à l'arrêt du serveur. Si ce processus meurt de façon inattendue, le processus postmaster traite cet événement comme la mort inattendue d'un processus postgres : arrêt du serveur PostgreSQL en urgence. Au prochain démarrage, une procédure de restauration aura lieu. Quelques variables permettent de configurer le moment de l'exécution d'un checkpoint :

- checkpoint_timeout indique la durée maximale sans CHECKPOINT ;
- checkpoint_segments indique le nombre maximum de journaux de transactions utilisés sans CHECKPOINT.

Par défaut, il y a un checkpoint au pire toutes les cinq minutes et à chaque fois que trois journaux de transactions ont été entièrement utilisés. Il s'est avéré par la suite que le checkpoint générait des pics importants d'activité. D'autres paramètres ont fait leur apparition pour permettre de lisser son activité. Par exemple, en 8.3, le paramètre checkpoint_completion_target permet de faire en sorte que les écritures se fassent sur X% du temps entre deux checkpoints. Par défaut à 0,5 (soit 50 %), il est possible d'augmenter ce paramètre jusqu'à 0,9 (90 %) pour diluer encore plus les écritures. Plutôt que d'écrire tous les blocs modifiés en un seul coup, on peut faire en sorte que, une fois une limite dépassée, un certain temps d'attente est respecté. Le délai est configuré avec le paramètre bgwriter_delay (200 ms par défaut). Quant à la limite, elle est calculée par rapport à deux paramètres : bgwriter_lru_maxpages, nombre maximum de blocs écrit sur disque et bgwriter_lru_multiplier, un facteur que le processus multiplie au nombre de blocs moyens réclamés lors des précédentes utilisations.

Le nombre de tampons sales écrits à chaque tour est basé sur le nombre de nouveaux tampons qui ont été requis par les processus serveur lors des derniers tours. Le besoin récent moyen est multiplié par bgwriter_lru_multiplier pour arriver à une estimation du nombre de tampons nécessaire au prochain tour. Les tampons sales sont écrits pour qu'il y ait ce nombre de tampons propres, réutilisables. (Néanmoins, au maximum bgwriter_lru_maxpages tampons sont écrits par tour.) De ce fait, une configuration de 1.0 représente une politique d'écriture « juste à temps » d'exactement le nombre de tampons prédits. Des valeurs plus importantes fournissent une protection contre les pics de demande, alors qu'une valeur plus petite laisse intentionnellement des écritures aux processus postgres. La valeur par défaut est de 2. En cas de modification, il faut redémarrer le serveur.

Pour mieux étudier l'impact des checkpoints et l'intérêt ou non de ce processus, il est possible de tracer toute son activité grâce au paramètre log_checkpoint.

La première ligne indique le type de checkpoint : suite à un arrêt (« shutdown »), forcé (« immediate force wait »), dû au checkpoint_timeout (« time »), dû au checkpoint_segments (« xlog »). Par exemple, voici la trace pour la création d'une base de donnée :

```
LOG:  checkpoint starting: immediate force wait
LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 transaction log file(s) added, 0 removed, 0 recycled; write=0.000 s, sync=0.000 s, total=0.003 s
```

Lors de la création, un CHECKPOINT est forcé, ce qui se voit avec la première ligne (« immediate force »). Si ensuite nous insérons beaucoup de données dans une table de cette nouvelle base :

```
LOG:  checkpoint starting: xlog
LOG:  checkpoint complete: wrote 2438 buffers (79.4%); 0 transaction log file(s) added, 0 removed, 0 recycled; write=3.839 s, sync=0.489 s, total=6.701 s
```

La première ligne indique un CHECKPOINT suite à un dépassement de la valeur du paramètre checkpoint_segments (« xlog » sur la première ligne des traces).

En dehors du paramètre log_checkpoint, il est malgré tout possible d'avoir plus d'informations en passant au niveau de trace DEBUG1. À ce niveau, les changements forcés de journaux de transactions sont clairement précisés.

Enfin, il réalise aussi un autre nettoyage. Lorsqu'un journal de transactions est archivé, il reste dans le répertoire pg_xlog/archive_status un fichier du nom du journal de transactions mais avec une extension .done. Le processus bgwriter peut supprimer ce fichier, et traiter le journal de transactions (soit en le renommant soit en le supprimant). Au niveau mémoire, ce processus utilise un peu de mémoire partagée, la quantité dépendant essentiellement de la taille de la mémoire cache de PostgreSQL. Sur un serveur avec 24 Mo de mémoire cache (la valeur maximale par défaut), la mémoire allouée est de 29 Ko.

## Écriture des journaux de transactions

Le « wal writer process » est apparu en 8.3 pour gérer la même activité d'écriture en tâche de fond, mais cette fois du côté des journaux de transactions. Le but est toujours de décharger les processus postgres de ce travail. Il enregistre les modifications dans les journaux de transactions au moment du COMMIT ou si les informations à enregistrer ne tiennent plus dans le cache. Ce cache spécifique aux journaux de transactions permet d'éviter les écritures tant que le COMMIT n'est pas reçu. Cependant, comme tout cache, il a une limite paramétré avec la variable wal_buffers. En cas de débordement du cache, le « wal writer process » écrit les données sur disque.

De plus, dans le cas où l'enregistrement asynchrone est activé (paramètre asynchronous_commit), il garantit que l'enregistrement se fait au plus tard au bout de trois fois le délai indiqué par le paramètre wal_writer_delay.

# Processus de collecte des statistiques

Ce processus est activé par défaut en 8.3. Contrairement aux deux précédents types de processus, il peut être désactivé grâce au paramètre track_activities.

Le but de ce processus est de récupérer certaines informations des processus postgres. Il s'agit des statistiques sur le nombre de lignes lues, insérées, modifiées et supprimées. Ce sont aussi des statistiques sur le nombre de blocs disque lus ou écrits. Bref, toutes sortes d'informations sur l'activité du serveur sont comptés à condition que le paramètre track_count soit activé.

Ces informations sont récupérées via un port UDP configuré en mode non bloquant, ce qui permet, en cas de retard du collecteur, que les messages soient

ignorés et ne ralentissent pas le serveur PostgreSQL. Aucun processus postgres ne doit être ralenti, et encore pire bloqué, par l'envoi des informations de statistique. Mais du coup, cela sous-entend que les statistiques ne sont pas forcément exactes.

Les données sont temporairement enregistrées dans le fichier global/pgstat.tmp. On y trouve les statistiques du processus d'écriture en tâche de fond, mais aussi des statistiques sur les bases et relations. Une fois l'enregistrement terminé, le fichier est renommé en global/pgstat.stat. Le fichier est d'autant plus gros que le nombre de bases et relations est important. Le fichier peut devenir très gros si un grand nombre de tables temporaires est créé. Cela génère une activité non négligeable étant donné que le fichier est écrit à chaque travail du collecteur de statistiques, c'est-à-dire toutes les 500 ms.

*Note :* Une solution à ce problème sera proposée en 8.4. Le fichier pgstat.stat sera créé dans un répertoire qui pourra être placé dans un disque RAM. Ce répertoire sera personnalisable via un paramètre du fichier de configuration postgresql.conf. Il sera copié dans le répertoire global à l'arrêt du serveur et récupéré à cet endroit au prochain démarrage.

Les processus postgres font appel au collecteur pour fournir les informations sur les tables accédées, lues, etc. L'opération VACUUM fait de même pour indiquer les bases de données et les relations supprimées. Dans le cas des bases, l'ordre DROP DATABASE envoie déjà ce message au collecteur, mais ce n'est malheureusement pas le cas avec un DROP TABLE. Cela explique que, en l'absence de VACUUM et si la base utilise beaucoup de tables temporaires, ces dernières sont toujours présentes dans le fichier pgstat.stat, sans aucune raison.

Ce processus est exécuté au lancement du serveur PostgreSQL et vit jusqu'à l'arrêt du serveur. Si ce processus meurt de façon inattendue, le processus postmaster tente de le relancer plusieurs fois si nécessaire.

Question mémoire, il n'utilise pas la mémoire partagée mais comprend quelques structures qu'il conserve tout au long de son exécution. La taille de ces structures dépend directement du nombre de processus car une structure est alloué par processus. Pour donner une idée de la quantité de mémoire impliquée, cela représente environ 128 Ko de mémoire partagée allouée jusqu'à cent processus.

## Processus de maintenance des tables

Ce démon a pour but de procéder au nettoyage des tables si elles ont eu une activité suffisante pour nécessiter cette opération. Depuis la version 8.3, il existe le démon principal, dénommé « autovacuum launcher », et des processus secondaires, appelés « autovacuum worker ».

L' « autovacuum launcher » est exécuté dès le démarrage par le processus postmaster. Son exécution durera jusqu'à l'arrêt complet du serveur PostgreSQL. Si ce processus meurt de façon inattendue, le processus postmaster tente de le relancer plusieurs fois si nécessaire.

Au démarrage, il commence par allouer un petit espace de mémoire partagée entre tous les processus autovacuum. Cet espace mémoire est vraiment restreint en utilisation aux processus autovacuum et a une taille très limitée (144 octets sur un serveur tout juste installé). Cette taille dépend du paramètre autovacuum_max_workers.

L'allocation mémoire réalisée, il s'endort. Il se réveille à intervalle régulier dépendant du paramètre autovacuum_naptime. Lors de ce réveil, il indique la base de données à traiter dans l'espace en mémoire partagée qu'il a alloué à son lancement et demande au processus postmaster d'exécuter un processus « autovacuum worker ». Il ne peut exécuter lui-même un « autovacuum_worker » car il est moins robuste que le processus postmaster, notamment parce qu'il utilise la mémoire partagée où peut survenir une corruption.

Lorsqu'un nouveau processus autovacuum est exécuté, il détecte la présence de la mémoire partagée liée aux processus autovacuum et comprend du coup qu'il est un « autovacuum worker ». Il se connecte à la base de données indiquée dans la mémoire partagée, recherche les tables et index à traiter, les traite puis quitte en envoyant un signal SIGUSR1 au processus « autovacuum launcher ». Ainsi, il prévient ce processus pour que ce dernier puisse demander l'exécution d'un nouvel « autovacuum worker » (utile quand on a atteint le maximum de processus de ce type et qu'on est malgré tout en attente pour en exécuter un autre).

Plusieurs « autovacuum worker » peuvent fonctionner en même temps, avec un maximum dépendant du paramètre autovacuum_max_workers. Plusieurs peuvent travailler en même temps sur la même base. Ils ne se bloqueront pas sur la même table ou sur le même index car ils indiquent en mémoire partagée la table (ou l'index) sur lesquels ils travaillent. Les coûts associés à l'autovacuum sont balancés entre chaque « autovacuum worker » (voir pour cela les paramètres autovacuum_vacuum_cost_delay et autovacuum_vacuum_cost_limit). Lorsqu'un « autovacuum worker » exécute réellement un VACUUM, une mémoire supplémentaire est allouée, dont la taille dépend du paramètre maintenance_work_mem.

Des traces sont disponibles pour ce processus. Le paramètre log_autovacuum_min_duration permet d'enclencher des traces pour tous les processus autovacuum dont la durée d'exécution dépasse la durée indiquée par le paramètre. Des traces supplémentaires sont disponibles au niveau DEBUG1 (une trace indiquant la base de donnée traitée par un « autovacuum worker »), au niveau DEBUG2 (pour comprendre le calcul des coûts associés au VACUUM) et au niveau DEBUG3 (indiquant le résultat des opérations VACUUM et ANALYZE).

## Processus d'archivage

Désactivé par défaut, ce processus a pour but de gérer l'archivage des journaux de transactions. L'activation se fait de façon simple : il suffit de configurer archive_mode à on. Ceci fait, après un redémarrage de PostgreSQL, un processus nommé « archiver process » apparaît. Il n'a pas de consommation mémoire importante et il n'utilise pas non plus la mémoire partagée. Il sera là jusqu'à l'arrêt du serveur PostgreSQL. Si jamais il meurt de façon inattendue, le serveur tentera plusieurs fois de le relancer. Il communique avec les autres processus au moyen de signaux. SIGHUP par exemple lui fait recharger la configuration.

S'il reçoit le signal SIGUSR1 ou au plus tard au bout de 60 secondes, il lit le contenu du répertoire pg_xlog/archive_status. Il récupère le fichier d'extension .ready le plus ancien, en déduit le nom du journal de transactions à archiver et exécute la commande d'archivage. Si cette dernière renvoie le code de statut 0 (signifiant la réussite de l'opération), il renomme le fichier .ready en .done et se rendort.

Le nom du processus affiché par ps dépend essentiellement de son état :

- « archiver process » au départ ;
- « archiving %s », lors de l'archivage du journal de transactions %s ;
- « failed on %s » si l'archivage a échoué pour le journal %s ;
- « last was %s » dès qu'un journal de transactions a été archivé avec succès.

En dehors du nom du processus, il est possible de détailler l'exécution de ce processus. Le niveau de traces DEBUG3 indique le nom de la commande d'archivage en cours d'exécution alors que le niveau DEBUG1 signale un journal de transactions archivé avec succès.

## Processus de gestion des journaux applicatifs

Disponible depuis la version 8.0 mais désactivé par défaut, ce processus sert à rediriger les messages de la sortie standard vers les journaux applicatifs gérés par PostgreSQL. Il apparaît sous le nom de « logger process ». Il utilise très peu de mémoire et se détache en plus de la mémoire partagée. La communication avec les autres processus se fait par l'intermédiaire d'un tube.

Il réagit notamment à deux signaux : SIGHUP pour lui demander de relire la configuration et SIGUSR1 pour demander une rotation du journal applicatif. Dans le cas du SIGHUP, le journal est changé si le répertoire des journaux applicatifs ou le modèle de nom des journaux a changé.

Ce processus est exécuté au lancement du serveur PostgreSQL et vit jusqu'à l'arrêt du serveur. Si ce processus meurt de façon inattendue, le processus postmaster tente de le relancer plusieurs fois si nécessaire.

## Processus de communication client/serveur

Ce sont les plus nombreux. Chaque processus postgres s'occupe de la communication entre le client qui s'est connecté et le serveur, que ce client soit l'outil psql, l'outil de sauvegarde pg_dump ou n'importe quel autre application capable de se connecter à une base de données PostgreSQL. Cependant, il n'y en aura jamais plus de max_connections.

Ils utilisent principalement la mémoire partagée pour stocker les pages des tables et index qu'ils utilisent. Ils utilisent aussi une partie de mémoire qui leur est propre, par exemple pour les tris ou pour les créations d'index.

Lorsqu'un client se connecte, le premier travail du processus postgres est de s'assurer de l'identité du client. Ceci fait, il effectue si nécessaire son authentification. Ensuite, il est en attente des requêtes du client. À réception d'une requête, il procède à son analyse et à son exécution. Pour cela, il peut avoir besoin de lire des tables qu'il placera dans la mémoire partagée. Une fois le résultat obtenu, il envoit le résultat au client. Il envoie les traces au collecteur de traces, récupère les statistiques qu'il fournir au collecteur de statistiques.

## Conclusion

Nous avons vu tous les processus/démons gérés par PostgreSQL. La plupart ne sont présents qu'en un exemplaire. Seul celui qui se charge des communications avec les clients peut apparaître plusieurs fois. Ils utilisent principalement la mémoire partagée, le reste dépend de leur travail mais ne représente pas beaucoup.