Module H1

Supervision de PostgreSQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Supervision de PostgreSQL

Module H1

TITRE : Supervision de PostgreSQL
SOUS-TITRE : Module H1

REVISION: 22.09
DATE: 02 septembre 2022

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

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

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

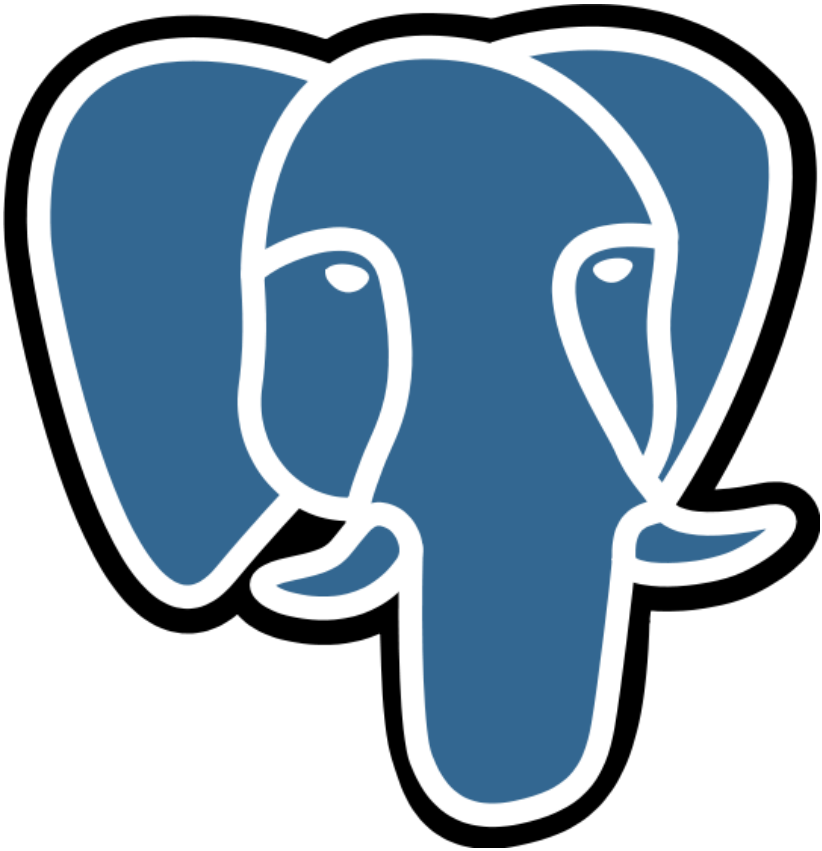
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

Table des Matières

1 SUPERVISION



1.1 INTRODUCTION

- Deux types de supervision
  - occasionnelle
  - automatique
- Superviser PostgreSQL et le système
- Pour PostgreSQL, statistiques et traces

Superviser un serveur de bases de données consiste à superviser le SGBD 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 SGBD. 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 survient, savoir comment interpréter les informations fournies par le système et par PostgreSQL) et comment mettre en place une supervision automatique (pour des alertes ou la supervision à long terme).

1.1.1 MENU

- Politique de supervision
- Supervision de PostgreSQL
- Traces : configuration & analyses
- Statistiques d'activité

1.2 POLITIQUE DE SUPERVISION

- Pour quoi ?
- Pour qui ?
- Quels critères ?
- Quels outils

Il n'existe pas qu'une seule supervision. Suivant la personne concernée par la supervision et son objectif, les critères de la supervision seront différents.

Lors de la mise en place de la supervision, il est important de se demander l'objectif de cette supervision, à qui elle va servir, les critères qui importent à cette personne.

Répondre à ces questions permettra de mieux choisir l'outil de supervision à mettre en place, ainsi que sa configuration.

1.2.1 OBJECTIFS DE LA SUPERVISION

- Améliorer/mesurer les performances
- Améliorer l'applicatif
- Anticiper/prévenir les incidents
- Réagir vite en cas de crash

Généralement, les administrateurs mettant en place la supervision veulent pouvoir anticiper les problèmes, qu'ils soient matériels, de performance, de qualité de service, etc.

Améliorer les performances du SGBD sans connaître les performances globales du système est très difficile. Si un utilisateur se plaint d'une perte de performance, pouvoir corroborer ses dires avec des informations provenant du système de supervision aide à s'assurer qu'il y a bien un problème de performances et peut fréquemment aider à résoudre ce problème. De plus, il est important de pouvoir mesurer les gains de performances.

Une supervision des traces de PostgreSQL permet aussi d'améliorer les applications qui utilisent une base de données. Toute requête en erreur est tracée dans les journaux applicatifs, ce qui permet de trouver rapidement les problèmes que les utilisateurs rencontrent.

Un suivi régulier de la volumétrie ou du nombre de connexions permet de prévoir les évolutions nécessaires du matériel ou de la configuration : achat de matériel, création d'index, amélioration de la configuration.

Prévenir les incidents peut se faire en ayant une sonde de supervision des erreurs disques par exemple. La supervision permet aussi d'anticiper les problèmes de configuration. Par exemple, surveiller le nombre de sessions ouvertes sur PostgreSQL permet de s'assurer que ce nombre n'approche pas trop du nombre maximum de sessions configuré avec le paramètre `max_connections` dans le fichier `postgresql.conf`.

Enfin, une bonne configuration de la supervision implique d'avoir configuré finement la gestion des traces de PostgreSQL. Avoir un bon niveau de trace (autrement dit : ni trop, ni pas assez) permet de réagir rapidement après un crash.

1.2.2 ACTEURS CONCERNÉS

- Développeur
  - correction et optimisation de requêtes
- Administrateur de bases de données
  - surveillance, performance
  - mise à jour
- Administrateur système
  - surveillance, qualité de service

Il y a trois types d'acteurs concernés par la supervision.

Le développeur doit pouvoir visualiser l'activité de la base de données. Il peut ainsi comprendre l'impact du code applicatif sur la base. De plus, le développeur est intéressé par la qualité des requêtes que son code exécute. Donc des traces qui ramènent les requêtes en erreur et celles qui ne sont pas performantes sont essentielles pour ce profil.

L'administrateur de bases de données a besoin de surveiller les bases pour s'assurer de la qualité de service, pour garantir les performances et pour réagir rapidement en cas de problème. Il doit aussi faire les mises à jours mineures dès qu'elles sont disponibles.

Enfin, l'administrateur système doit s'assurer de la présence du service. Il doit aussi s'assurer que le service dispose des ressources nécessaires, en terme de processeur (donc de puissance de calcul), de mémoire et de disque (notamment pour la place disponible).

1.2.3 EXEMPLES D'INDICATEURS - SYSTÈME D'EXPLOITATION

- Charge CPU
- Entrées/sorties disque
- Espace disque
- Sur-activité et non-activité du serveur
- Temps de réponse
- Outils Unix habituels :
  - `top`, `atop`, `free`, `df`, `vmstat`, `sar`, `iotop`

Voici quelques exemples d'indicateurs intéressants à superviser pour la partie du système d'exploitation.

La charge CPU (processeur) est importante. Elle peut expliquer pourquoi des requêtes, auparavant rapides, deviennent lentes. Cependant, la suractivité comme la non-activité sont un problème. En fait, si le service est tombé, le serveur sera en sous-activité, ce qui est un excellent indice.

Les entrées/sorties disque permettent de montrer un souci au niveau du système disque. Par exemple, PostgreSQL peut écrire trop à cause d'une mauvaise configuration des journaux de transactions, ou des fichiers temporaires issus de requêtes lourdes ou mal écrites ; ou il peut lire trop de données par manque de cache en RAM, ou de requêtes mal écrites.

L'espace disque est essentiel à surveiller. PostgreSQL ne propose rien pour cela, il faut donc le faire au niveau système. L'espace disque peut poser problème s'il manque, surtout si cela concerne la partition des journaux de transactions.

Unix possède de nombreux outils pour surveiller les différents éléments du système. Les grands classiques sont `top` et ses innombrables clones comme `atop` pour le CPU, `free` pour la RAM, `df` pour l'espace disque, `vmstat` pour la mémoire virtuelle, `iotop` pour les entrées/sorties, ou `sar` (généraliste). Sous Windows, les premiers outils sont bien sûr le Gestionnaire des tâches, et Process Monitor[2] des outils Sysinternals.

Il est conseillé d'avoir une requête étalon dont la durée d'exécution sera testée de temps à autre pour détecter les moments problématiques sur le serveur.

1.2.4 EXEMPLES D'INDICATEURS - BASE DE DONNÉES

- Nombre de connexions
- Requêtes lentes et/ou fréquentes
- Ratio d'utilisation du cache
- Verrous
- Volumétries
- ...

Il existe de nombreux indicateurs intéressant sur les bases : nombre de connexions (en faisant par exemple la différence entre connexions inactives, actives, en attente de verrous), nombre de requêtes lentes et/ou fréquentes, volumétrie (en taille, en nombre de lignes), des ratios (utilisation du cache par exemple)...

---

[2] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

1.3 SUPERVISION DE POSTGRESQL

- Supervision occasionnelle
  - sur incident…
- Supervision automatique
  - remonter des informations rapidement
  - archiver, suivre les tendances

La supervision occasionnelle est la conséquence d'une plainte d'un utilisateur : on se contente de réagir à un problème. C'est généralement insuffisant.

Il est important de mettre en place une solution de supervision automatique. Le but est de récupérer périodiquement des données statistiques sur les objets et sur l'utilisation du serveur pour avoir des graphes de tendance, et recevoir des alertes quand des seuils sont dépassés.

1.3.1 INFORMATIONS INTERNES

- PostgreSQL propose :
  - statistiques d'activité
  - traces
- …mais rien pour les historiser

PostgreSQL propose deux canaux d'informations : les statistiques d'activité (à ne pas confondre avec les statistiques sur les données, à destination de l'optimiseur de requêtes) et les traces applicatives (ou « logs »), souvent dans un fichier comme `postgresql.log` (le nom exact varie avec la distribution et l'installation).

PostgreSQL stocke un ensemble d'informations (métadonnées des schémas, informations sur les tables et les colonnes, données de suivi interne, etc.) dans des tables systèmes qui peuvent être consultées par les administrateurs. PostgreSQL fournit également des vues combinant des informations puisées dans différentes tables systèmes. Ces vues simplifient le suivi de l'activité de la base.

PostgreSQL est aussi capable de tracer un grand nombre d'informations qui peuvent être exploitées pour surveiller l'activité de la base de données.

Pour pouvoir mettre en place un système de supervision automatique, il est essentiel de s'assurer que les statistiques d'activité et les traces applicatives sont bien configurées et il faut aussi leur associer un outil permettant de sauvegarder les données, les alertes et de les historiser.

1.3.2 OUTILS EXTERNES

- Pour conserver les informations
- ...et exécuter automatiquement des actions
  - graphiques (Munin, Zabbix...)
  - envoi d'alertes (Nagios, tail_n_mail)

Pour récupérer et enregistrer les informations statistiques, les historiser, envoyer des alertes, il faut faire appel à un outil externe. Cela peut être un outil très simple comme munin ou un outil très complexe comme Nagios ou Zabbix.

1.3.3 CHECK_PGACTIVITY

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - nombreuses métriques remontées
- Développé au départ par Dalibo
  - utilisable indépendamment
- https://github.com/OPMDG/check_pgactivity

Le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios (entre autres).

Au départ, Dalibo utilisait une sonde nommmée `check_postgres` et participait activement à son développement, avec même un commiter dans le projet. Cependant, rapidement, nous nous sommes aperçus que nous ne pouvions pas aller aussi loin que nous le souhaitions. C'est à ce moment que, dans le cadre de sa R&D, Dalibo a conçu `check_pgactivity`.

La plupart des sondes de `check_postgres` y ont été réimplémentées. Le script corrige certaines sondes existantes et en fournit de nouvelles répondant mieux aux besoins de notre supervision, avec notamment un nombre plus important de métriques de performances. Il renvoie directement des ratios par rapport à la valeur précédente plutôt que des valeurs fixes. Pour les besoins les plus simples, le script peut être utilisé de façon autonome, sans nécessité d'installer toute l'infrastructure d'un outil comme Nagios, Icinga ou Grafana.

La supervision d'un serveur PostgreSQL passe par la surveillance de sa disponibilité, des indicateurs sur son activité, l'identification des besoins de maintenance, et le suivi de la réplication le cas échéant. Ci-dessous figurent les sondes `check_pgactivity` à mettre en

place sur ces différents aspects. Le site du projet contient toute la documentation de chaque sonde.

Disponibilité :

- `connection` : réalise un test de connexion pour vérifier que le serveur est accessible ;
- `backends` : compte le nombre de connexions au serveur comparé au paramètre `max_connections` ;
- `backend_status` : permet d'obtenir des statistiques plus précises sur l'état des connexions clientes et d'être alerté lorsqu'un certain nombre de connexions clientes sont dans un état donné (*waiting*, *idle in transaction*…) ;
- `uptime` : détecte un redémarrage du serveur ou du rechargement de la configuration.

Vacuum :

- `autovacuum` : suit le fonctionnement de l'autovacuum et des tâches en cours (`VACUUM`, `ANALYZE`, `FREEZE`…) ;
- `table_bloat` : vérifie le volume de données « mortes » et la fragmentation des tables ;
- `btree_bloat` : vérifie le volume de données « mortes » et la fragmentation des index - par rapport à `check_postgres`, le calcul est séparé entre tables et index ;
- `last_analyze` : vérifie si le dernier analyze (relevé des statistiques relatives aux calculs des plans d'exécution) est trop ancien ;
- `last_vacuum` : vérifie si le dernier vacuum (relevé des espaces réutilisables dans les tables) est trop ancien.

Activité :

- `locks` : permet d'obtenir des statistiques plus détaillées sur les verrous obtenus et tient notamment compte des spécificités des *predicate locks* du niveau d'isolation `SERIALIZABLE` ;
- `wal_files` : compte le nombre de segments du journal de transaction présents dans le répertoire `pg_wal` ;
- `longest_query` : permet d'être alerté si une requête est en cours d'exécution depuis plus d'un certain temps ;
- `oldest_xact` : permet d'être alerté si une transaction est ouverte depuis un certain temps sans être utilisée ;
- `oldest_2pc` : calcule l'âge de la plus ancienne transaction préparée (*two-phase commit transaction*) ;
- `oldest_xmin` : repère la plus ancienne transaction de chaque base, et ce à quoi elle est liée (requête, slot…) ;
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL ;

- `hit_ratio` : calcule le *hit ratio* (utilisation du cache de PostgreSQL) ;
- `commit_ratio` : calcule la proportion de `COMMIT` et `ROLLBACK` ;
- `checksum_errors` : détecte l'apparition d'erreurs de sommes de contrôle (à partir de PostgreSQL 12) ;
- `database_size` : suit la volumétrie des bases et leurs variations ;
- `max_freeze_age` : calcule l'âge des plus vieilles lignes stockées dans chaque base pour suivre le bon passage des `VACUUM FREEZE` ;
- `stat_snapshot_age` : calcule l'âge des statistiques d'activité pour repérer un blocage du collecteur ;
- `temp_files` : suivi des fichiers temporaires.

Configuration :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut ;
- `minor_version` : détecte les instances n'ayant pas la dernière version mineure ;
- `settings` : repère un changement des paramètres ;
- `invalid_indexes` : repérer tout index invalide ;
- `pgdata_permission` : vérifie les droits sur `PGDATA` pour éviter un blocage au redémarrage ;
- `table_unlogged` : remonte le nombre de tables *unlogged* ;
- `extensions_versions` : détecte les extensions à mettre à jour.

Réplication & archivage :

- `archiver` : compte le nombre de segments du journal de transaction en attente d'archivage ;
- `archive_folder` : vérifie qu'il n'y a pas de journal manquant dans les archives de sauvegarde PITR ;
- `hot_standby_delta` : calcule le délai de réplication entre un serveur primaire et un serveur secondaire ;
- `is_master` / `is_hot_standby` : vérifie que l'instance est bien démarrée en lecture/écriture, ou une instance secondaire ;
- `is_replay_paused` : vérifie si la réplication est en pause ;
- `replication_slots` : calcule la volumétrie conservée pour chaque slot de réplication.

Sauvegarde physique et logique :

- `backup_label_age` : calcule l'âge du fichier `backup_label` (sauvegardes PITR exclusives) ;
- `pg_dump_backup` : contrôle l'âge et la variation de taille des sauvegardes logiques.

1.3.4 CHECK_POSTGRES

- Script de monitoring PostgreSQL pour Nagios ou MRTG
- Quelques autres sondes
- https://bucardo.org/wiki/Check_postgres

Le script de monitoring `check_postgres` est la première sonde à avoir été écrite pour PostgreSQL. Comme vu précédemment, `check_pgactivity` est un remplaçant plus fonctionnel, donnant plus de métriques, sur un nombre plus limité de sondes.

`check_postgres` évolue cependant toujours et est intéressant dans certains cas : alerte pour les triggers désactivés, taille des relations, estimation du retard en réplication logique (native et Slony), comparaison de schémas, détection de proximité du wraparound. Elle intègre aussi beaucoup de sondes pour l'outil de pooling pgBouncer.

1.4 TRACES

- Configuration
  - traces peu fournies par défaut
- Récupération
  - des problèmes importants
  - des requêtes lentes/fréquentes
- Outils externes de classement

La première information que fournit PostgreSQL sur son activité sort dans les traces. Chaque requête en erreur génère une trace indiquant la requête erronée et l'erreur. Chaque problème de lecture ou d'écriture de fichier génère une trace. En fait, tout problème détecté par PostgreSQL fait l'objet d'un message dans les traces. PostgreSQL peut aussi y envoyer d'autres messages suivant certains événements, comme les connexions, l'activité de processus système en tâche de fond, etc.

Nous allons donc aborder la configuration des traces (où tracer, quoi tracer, quel niveau d'informations). Nous verrons au passage nombre d'informations intéressantes à récupérer. Enfin, nous verrons quelques outils permettant de traiter automatiquement les fichiers de trace.

1.4.1 CONFIGURATION DES TRACES : PRINCIPES

- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
  - durée, fichiers temporaires…
- Tracer certains comportements
  - erreurs

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas à la fois trop lourdes (pour ne pas être submergé par les informations) et incomplètes (il manque des informations). Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

Suivant la configuration réalisée, les journaux applicatifs peuvent contenir quantité d'informations importantes. La plus fréquemment recherchée est la durée d'exécution des requêtes. L'intérêt principal est de récupérer les requêtes les plus lentes. L'autre information importante concerne les messages d'erreur, de niveau PANIC en premier lieu. Ces messages indiquent un état anormal du serveur qui s'est soldé par un arrêt brutal. Ce genre de problème est anormal et doit être surveillé.

1.4.2 ÉVÉNEMENTS EXCEPTIONNELS TRACÉS

- Crash de PostgreSQL :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

- Rechargement de la configuration :

```
LOG:  received SIGHUP, reloading configuration files
```

- Envoi immédiat d'une alerte
- Outil : tail_n_mail

Les messages `PANIC` sont très importants. Généralement, vous ne les verrez pas au moment où ils se produisent. Un crash va survenir et vous allez chercher à comprendre ce qui s'est passé. Il est possible à ce moment-là que vous trouviez dans les traces des messages `PANIC`, comme celui-ci :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

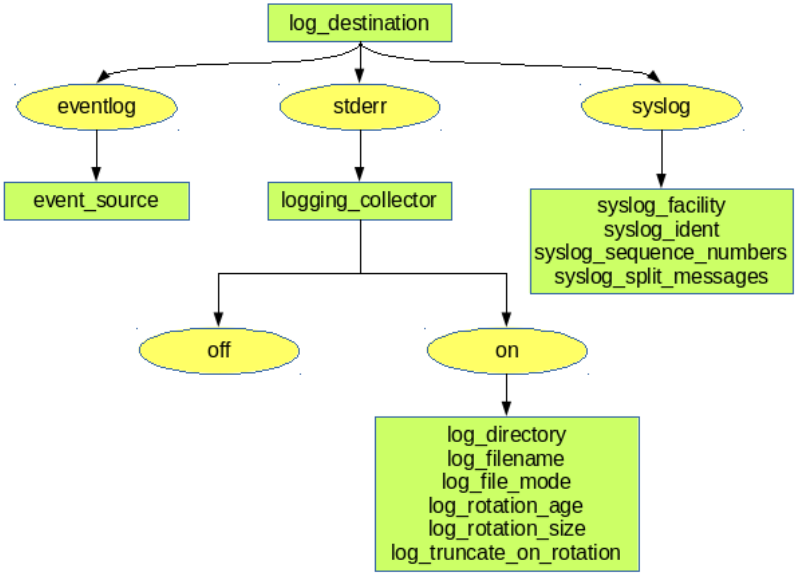
Là, le problème est très simple : PostgreSQL n'arrive pas à créer un journal de transactions à cause d'un manque d'espace sur le disque. Du coup, le système ne peut plus fonctionner, il panique et s'arrête.

Un outil comme tail_n_mail peut aider à détecter automatiquement ce genre de problème et à envoyer un mail à la personne d'astreinte. Il n'est d'ailleurs pas si rare que PostgreSQL, après un problème grave, redémarre si vite qu'il n'y a aucune conséquence visible sérieuse, et que ce genre de détection automatique soit le seul symptôme d'un problème.

Un autre événement à suivre est le changement de la configuration du serveur, et surtout la valeur des paramètres modifiés. PostgreSQL envoie un message niveau `LOG` lorsque la configuration est relue. Il indique aussi les nouvelles valeurs des paramètres, ainsi que les paramètres modifiés qu'il n'a pas pu prendre en compte (cela peut arriver pour tous les paramètres exigeant un redémarrage du serveur).

Là-aussi, tail_n_mail est l'outil adapté pour être prévenu dès que la configuration du serveur est relue.

1.4.3 OÙ TRACER ?



1.4.4 CONFIGURATION DE LA DESTINATION DES TRACES

- `log_destination` : `stderr` / `syslog` / `csvlog` / `eventlog`
- `logging_collector` : géré par PostgreSQL (Red Hat)
  - `log_directory`, `log_filename`, `log_file_mode`
  - `log_rotation_age`, `log_rotation_size`, `log_truncate_on_rotation`
- Sinon : si `off`, penser à `logrotate` (Debian)
- `syslog` (Unix)
  - `syslog_facility`, `syslog_ident`
  - `syslog_sequence_numbers`, `syslog_split_messages`
- `eventlog` (Windows) : `event_source`

PostgreSQL peut envoyer les traces sur plusieurs destinations selon la valeur de `log_destination` :

stderr et csvlog

`stderr` (valeur par défaut, y compris sur Debian & Red Hat) et `csvlog`, disponibles sur toutes les plateformes, correspondent à la sortie des erreurs.

La différence entre les deux réside dans le format des traces (texte simple ou CSV). (Noter qu'il existe une extension jsonlog[3] , par Michaël Paquier, qui offre en plus le format JSON).

Les paramètres suivants sont spécifiques à `stderr` et `csvlog`.

`logging_collector` indique ensuite si PostgreSQL doit s'occuper lui-même de la gestion des fichiers de logs.

S'il est à `on` (défaut sous Red Hat/CentOS/Rocky Linux) :

- `log_directory` désigne l'emplacement des journaux applicatifs. Par défaut, il est dans le répertoire de l'instance (sous-répertoire `log`, ou `pg_log` jusqu'en version 9.6 comprise) ;
- `log_filename` indique le nom des fichiers. Sa valeur varie suivant la distribution :
  - `postgresql-%Y-%m-%d_%H%M%S.log` est le défaut des versions compilées (avec rotation quotidienne) ;
  - `postgresql-%a.log` est le défaut sur Red Hat/CentOS/Rocky Linux : on obtient une rotation quotidienne sur une semaine (`postgresql-Mon.log`, `postgresql-Tue.log`, etc.) ;
- `log_file_mode` précise les droits sur les fichiers (`0600` par défaut, les réservant à l'utilisateur sous lequel l'instance tourne). Une rotation est configurable suivant la

[3] https://github.com/michaelpq/pg_plugins/tree/master/jsonlog

taille (`log_rotation_size`) et la durée de vie (`log_rotation_age`, souvent `1d`, à garder en cohérence avec `log_filename`) ;

- `log_truncate_on_rotation` à `on` entraîne l'effacement de tout fichier dont le nom serait réutilisé, ce qui est en général une bonne idée si les mêmes noms de fichiers sont réutilisés.

Par contre, sous Debian, `logging_collector` est par défaut à `off`, les paramètres ci-dessus sont ignorés et la gestion des logs est gérée par le système d'exploitation :

- les traces vont dans `/var/log/postgresql/`, donc hors du PGDATA ;
- elles sont nommées en fonction de l'instance, de son nom et du nom fourni lors de la création avec l'outil `pg_ctlcluster` (donc pour l'instance installée par défaut, en version 14, le fichier sera `postgresql-14-main.log`), sauf à modifier le `pg_ctl.conf` de l'instance ;
- la rotation des fichiers est gérée par `logrotate` (comme les autres fichiers de log), et paramétrée dans `/etc/logrotate.d/postgresql-common` avec par défaut une rotation sur 10 jours.

Une instance compilée n'utilise pas non plus le *logging collector*.

syslog

`syslog` fonctionne uniquement sur un serveur Unix et est intéressant pour centraliser la configuration des traces.

Dans cette configuration, il reste à définir le niveau avec `syslog_facility`, et l'identification du programme avec `syslog_ident`. Les valeurs par défaut de ces deux paramètres sont généralement bonnes. Il est intéressant de modifier ces valeurs surtout si plusieurs instances de PostgreSQL sont installées sur le même serveur car cela permet de différencier leur traces.

`syslog_sequence_numbers` préfixe chaque message d'un numéro de séquence incrémenté automatiquement, pour éviter le message `--- last message repeated N times ---`, utilisé par un grand nombre d'implémentations de syslog. Ce comportement est activé par défaut. Quant à `syslog_split_messages`, s'il est activé, les messages envoyés à syslog sont divisés par lignes, elles-mêmes divisées pour tenir sur 1024 octets. Attention à ce que `syslog` soit configuré pour accepter des débits élevés quand on tient à tout tracer.

eventlog

`eventlog` est disponible uniquement sur Windows et alimente le journal des événements. Pour identifier les messages de PostgreSQL, il faut aussi renseigner `event_source`[4] , qui

[4] https://docs.postgresql.fr/current/event-log-registration.html

par défaut est à « PostgreSQL ».

1.4.5 NIVEAU DES TRACES

- `log_min_messages`
  - défaut : `panic` / `fatal` / `log` / `error` / `warning`
- `log_min_error_statement`
  - défaut : `error` (ou `warning`)
- `log_error_verbosity`
  - `default` / `terse` / `verbose`

`log_min_messages` est le paramètre à configurer pour avoir plus ou moins de traces. Par défaut, PostgreSQL enregistre tous les messages de niveau `panic`, `fatal`, `log`, `error` et `warning`. Cela peut sembler beaucoup mais, dans les faits, c'est assez discret. Cependant, il est possible de descendre le niveau ou de l'augmenter.

`log_min_error_statement` indique à partir de quel niveau la requête est elle-aussi tracée. Par défaut, la requête n'est tracée que si une erreur est détectée. Généralement, ce paramètre n'est pas modifié, sauf dans un cas précis. Les messages d'avertissement (niveau `warning`) n'indiquent pas la requête qui a généré l'affichage du message. Cela est assez important, notamment dans le cadre de l'utilisation d'antislash dans les chaînes de caractères. On verra donc parfois un abaissement au niveau `warning` pour cette raison.

`log_error_verbosity` vaut `default`, qui convient généralement. Si une requête est tracée, plusieurs lignes peuvent apparaître, chacune de niveau `DETAIL`, `HINT`, `QUERY` ou `CONTEXT`. La valeur `terse` les masque. À l'inverse, `verbose` rajoute encore d'autres informations sur le code source d'origine.

1.4.6 TRACER LES REQUÊTES ET LEUR DURÉE

- Toutes les requêtes :
  - `log_min_duration_statement` (ex : `1s`)
  - ou `log_statement` + `log_duration`
- Extrait aléatoire :
  - `log_transaction_sample_rate`
  - `log_statement_sample_rate` + `log_min_duration_sample`

Pour répérer les problèmes de performances, il est intéressant de pouvoir tracer les requêtes et leur durée d'exécution. PostgreSQL propose deux solutions à cela.

log_statement & log_duration :

La première solution disponible concerne les paramètres `log_statement` et `log_duration`. Le premier permet de tracer toute requête exécutée si la requête correspond au filtre indiqué par le paramètre :

- `none` : aucune requête n'est tracée ;
- `ddl` : seules les requêtes DDL (autrement dit de changement de structure) sont tracées ;
- `mod` : seules les requêtes de changement de structure et de données sont tracées ;
- `all` : toutes les requêtes sont tracées.

Le paramètre `log_duration` est un simple booléen. S'il vaut `true` ou `on`, chaque requête exécutée envoie en plus un message dans les traces indiquant la durée d'exécution de la requête. Évidemment, il vaut mieux alors configurer `log_statement` à `all`, ou il sera impossible de dire à quelles requêtes les temps correspondent.

Donc pour tracer toutes les requêtes et leur durée d'exécution, une solution serait de réaliser la configuration suivante :

```
log_statement = 'all'
log_duration = on
```

Une requête générera deux entrées dans les traces, de cette façon :

```
2019-01-28 15:43:27.993 CET [25575] LOG:  statement: SELECT * FROM pg_stat_activity;
2019-01-28 15:43:27.999 CET [25575] LOG:  duration: 7.093 ms
```

log_min_duration_statement :

Il est préférable de désactiver ces deux paramètres et de préférer `log_min_duration_statement`. Son but est d'abord de cibler les requêtes lentes, par exemple celles qui prennent plus de deux secondes à s'exécuter :

```
log_min_duration_statement = '2s'
```

La requête et la durée d'exécution seront alors tracées dans le même message :

```
2019-01-28 15:49:56.193 CET [32067] LOG:
  duration: 2906.270 ms
  statement: insert into t1 select i, i from generate_series(1, 200000) as i;
```

En plus de la trace par `log_min_duration_statement`, rien n'interdit de tracer des requêtes sensibles, notamment le DDL :

```
log_statement = 'ddl'
```

Échantillonnage :

Quelle que soit la méthode, tracer toutes les requêtes peut poser problème pour de simples raisons de volumétrie du fichier de traces. Même s'il est possible de configurer finement la durée à partir de laquelle une requête est tracée, il faut bien comprendre que plus la durée minimale est importante, plus la vision des performances est partielle. Passeront ainsi « sous le radar » des requêtes relativement rapides mais très nombreuses qui, ensemble, peuvent représenter l'essentiel de la charge.

Cela étant dit, laisser 0 en permanence n'est pas recommandé. Il est préférable de configurer ce paramètre à une valeur plus importante en temps normal pour détecter seulement les requêtes longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

Une nouvelle fonctionnalité a donc été ajoutée : tracer une certaine proportion des requêtes ou des transactions.

`log_transaction_sample_rate`, à partir de PostgreSQL 12, indique une proportion de **transactions** à tracer. Par exemple, en le configurant à `0.01`, toutes les requêtes d'un centième des transactions, choisies au hasard, seront tracées.

De manière similaire, à partir de PostgreSQL 13, `log_statement_sample_rate` indique la proportion de **requêtes** à tracer, parmi celles durant plus d'une certaine durée, à indiquer dans `log_min_duration_sample` :

```
log_min_duration_sample = '10ms'
log_statement_sample_rate = 0.01
```

Évidemment, une requête dépassant la durée de `log_min_duration_statement` sera toujours tracée.

1.4.7 CONFIGURATION : TRACER CERTAINS COMPORTEMENTS

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits` (mini 1s)
- `log_recovery_conflict_waits` (v14+)

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements. Le paramétrage par défaut est peu bavard, et il est généralement conseillé d'activer tous les paramètres qui suivent.

`log_connections` et son pendant `log_disconnections`, à `on`, permettent de suivre qui se (dé)connecte, depuis où, et durant combien de temps :

```
2019-01-28 13:34:32 CEST LOG:  connection received: host=[local]
2019-01-28 13:34:32 CEST LOG:  connection authorized: user=u1 database=b1
…
2016-09-01 13:34:35 CEST LOG:  disconnection: session time: 0:01:04.634
                               user=u1 database=b1 host=[local]
```

Il est possible de récupérer cette durée de session pour calculer leur durée moyenne. Cette information est importante pour savoir si un outil de pooling de connexions a un intérêt.

`log_autovacuum_min_duration` équivaut à `log_min_duration_statement`, mais pour l'autovacuum. Le but est de tracer son activité, au-delà d'une certaine durée, de vérifier qu'il passe suffisamment fréquemment et rapidement.

`log_checkpoints` à `on` ajoute un message dans les traces pour indiquer qu'un checkpoint commence ou se termine, auquel cas s'ajoutent des statistiques :

```
2019-01-28 13:34:17 CEST LOG: checkpoint starting: xlog
2019-01-28 13:34:20 CEST LOG: checkpoint complete:
  wrote 13115 buffers (80.0%);
  0 WAL file(s) added, 0 removed, 0 recycled;
  write=3.007 s, sync=0.324 s, total=3.400 s;
  sync files=16, longest=0.285 s, average=0.020 s;
  distance=404207 kB, estimate=404207 kB
```

Le message indique donc en plus le nombre de blocs écrits sur disque, le nombre de journaux de transactions ajoutés, supprimés et recyclés. Il est rare que des journaux soient ajoutés, ils sont plutôt recyclés. Des journaux sont supprimés quand il y a eu une très grosse activité qui a généré plus de journaux que d'habitude. Les statistiques incluent aussi la durée des écritures, de la synchronisation sur disque, la durée totale, etc. Le plus important est de pouvoir vérifier que l'écriture des checkpoints est bien régulière (essentiellement périodique).

`log_lock_waits` à `on` permet de tracer les attentes de verrous (par exemple, un `UPDATE` bloqué par un autre `UPDATE`, un `SELECT` bloqué par `TRUNCATE` ou un `VACUUM FULL`, etc…) Lorsque l'attente dépasse la durée indiquée par le paramètre `deadlock_timeout` (1 seconde par défaut), un message est enregistré, comme dans cet exemple :

```
2019-01-28 13:38:40 CEST LOG:  process 15976 still waiting for
                               AccessExclusiveLock on relation 26160 of
                               database 16384 after 1000.123 ms
2019-01-28 13:38:40 CEST STATEMENT:  DROP TABLE t1;
```

Ici, un `DROP TABLE` attend depuis 1 seconde de pouvoir poser un verrou exclusif sur une relation.

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur certains objets, ce qui peut diminuer fortement les performances. Ces messages peuvent permettre d'analyser la cause première d'une accumulation de verrous, à condition que les requêtes soient tracées.

En version 14 apparaît le paramètre `log_recovery_conflict_waits`. Ce dernier, une fois activé, permet de tracer toute attente due à un conflit de réplication. Il n'est donc valable et pris en compte que sur un serveur secondaire.

1.4.8 REPÉRER LES FICHIERS TEMPORAIRES

- Exemple :

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp9894.0",
      size 26927104
```

- `log_temp_files` à activer !
- Alerte : problème potentiel de performances

Quand PostgreSQL ne peut effectuer un tri en mémoire, il le fait sur disque dans un fichier temporaire, ce qui est beaucoup plus lent qu'en mémoire, même avec un SSD. Typiquement, cela concerne le tri de données et le hachage, quand la valeur du paramètre `work_mem` ne permet pas de tout faire en mémoire. Cela ne sera pas forcément gênant pour une grosse requête ponctuelle, mais, répétés, ces fichiers peuvent avoir un impact sur la performance du système. Ils sont parfois inévitables quand on brasse beaucoup de données.

Être averti lors de la création de ce type de fichiers peut être intéressant, mais ils sont parfois trop fréquents pour que ce soit réaliste. Il est préférable de faire analyser après coup un fichier de traces pour savoir combien de fichiers temporaires ont été créés, et de quelles tailles. Cela peut mener à vérifier les requêtes exécutées, les optimiser, vérifier la configuration, réviser la valeur de `work_mem`…

Le paramètre `log_temp_files` à 0 permet de tracer toutes les créations de fichiers temporaires, comme ici :

```
2019-01-28 13:41:11 CEST LOG:  temporary file: path
                               "base/pgsql_tmp/pgsql_tmp15617.1",
                               size 59645952
```

Pour le même tri, il peut y avoir de nombreux fichiers temporaires. De plus, la requête est aussi tracée, et si elle est longue et fréquente, le volume de traces peut être conséquent.

1.4.9 CONFIGURATION : DIVERS

- `log_line_prefix`
  - Conseillé : `%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h`
- `lc_messages` = `C`
- `log_timezone` = `'Europe/Paris'`

Le paramètre `log_line_prefix` permet d'ajouter un préfixe à une trace. Le défaut (`'%m [%p] '`, soit horodatage et numéro de processus), est insuffisant :

```
2021-02-05 14:12:12.343 UTC [2917] LOG:  duration: 3.276 ms
                          statement: SELECT count(*) FROM pgbench_branches ;
```

Il est conseillé de rajouter le nom de l'application cliente, le nom de l'utilisateur, le nom de la base, etc. Une valeur habituellement conseillée pour pgBadger[5] , pour une sortie vers `stderr`, est :

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
```

ce qui nous donnera ce genre de traces :

```
2021-02-05 14:30:01 UTC [3006]: user=durand,db=bench,app=test,client=[local]
    LOG:  duration: 0.184 ms  statement: SELECT count(*) FROM pgbench_branches ;
```

Pour une sortie vers `syslog`, l'horodatage est inutile :

```
log_line_prefix = 'user=%u,db=%d,app=%a,client=%h '
```

Par défaut, les traces sont enregistrées dans la locale par défaut du serveur. Des traces en français peuvent présenter certains intérêts pour des débutants, mais ont plusieurs gros inconvénients : un moteur de recherche renverra beaucoup moins de résultats avec des traces en français qu'en anglais, et les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il vaut mieux préciser systématiquement :

```
lc_messages = 'C'
```

Quant à `log_timezone`, il permet de choisir le fuseau horaire pour l'horodatage des traces. C'est inestimable quand on administre différents serveurs dispersés sur la planète.

```
log_timezone = 'UTC'

log_timezone = 'Europe/Paris'
```

---

[5] https://pgbadger.darold.net/documentation.html#POSTGRESQL-CONFIGURATION

1.5 OUTILS D'ANALYSE DES TRACES

- Beaucoup d'outils existent
  - en temps réel / rétro-analyse
  - généralistes / spécifiques PostgreSQL
- Exemples :
  - pgBadger
  - logwatch
  - tail_n_mail

Il existe de nombreux programmes qui analysent les traces. On peut distinguer deux catégories :

- ceux qui le font en temps réel ;
- ceux qui le font après coup (de la rétro-analyse en fait).

Mais également :

- ceux généralistes, connaissant plus ou moins bien beaucoup d'outils et logiciels ;
- ceux dédiés à PostgreSQL.

L'analyse en temps réel des traces permet de réagir rapidement à certains messages. Par exemple, il est important d'avoir une réaction rapide à l'archivage échoué d'un journal de transactions, ainsi qu'en cas de manque d'espace disque. Dans cette catégorie, il existe des outils généralistes comme logwatch[6] , et des outils spécifiques pour PostgreSQL comme tail_n_mail[7] .

L'analyse après coup permet une analyse plus fine, se terminant généralement par un rapport en HTML, parfois avec des graphes. Cette analyse plus fine nécessite des outils spécialisés. Il en a existé plusieurs qui ne sont plus maintenus. La référence dans le domaine est pgBadger[8] .

---

[6] https://sourceforge.net/projects/logwatch/
[7] https://bucardo.org/tail_n_mail/
[8] https://pgbadger.darold.net/

1.5.1 PGBADGER

- Site officiel : https://pgbadger.darold.net/
- Licence : PostgreSQL
- Analyse des traces de durée d'exécution des requêtes
- Analyse des traces du VACUUM, des connexions, des checkpoints
- Compatible syslog, stderr, csvlog

Gilles Darold a créé pgBadger, un analyseur des journaux applicatifs de PostgreSQL. Il permet de générer des rapports détaillés depuis ceux-ci. pgBadger est très souvent utilisé pour déterminer les requêtes à améliorer en priorité pour accélérer son application basée sur PostgreSQL. C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL, au point qu'il est cité dans le manuel de PostgreSQL.

pgBadger est écrit en Perl et est facilement extensible si vous avez besoin de rapports spécifiques.

Il est conçu pour traiter rapidement de gros fichiers de traces avec une mémoire réduite, mais permet d'exploiter plusieurs CPU pour accélérer considérablement l'analyse.

1.5.2 PGBADGER : EXEMPLE DE RAPPORT

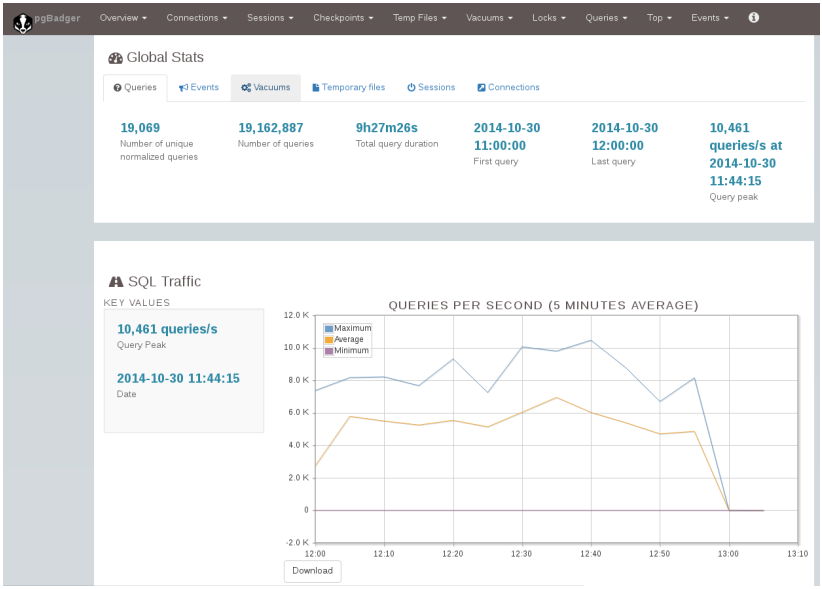


Figure 1: Capture pgBadger

1.5.3 UTILISER PGBADGER

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations
  - sur les requêtes (normalisées) et leur durée d'exécution
  - sur les connexions et sessions
  - sur les checkpoints
  - sur l'autovacuum
  - sur les verrous
  - etc.
- Génération d'un rapport HTML très détaillé

pgBadger s'utilise en ligne de commande : il suffit de lui fournir le ou les fichiers de trace à analyser et il rend un rapport HTML sur les requêtes exécutées, sur les connexions, sur

les bases, etc. Le rapport est très complet. Les graphes sont zoomables. Les requêtes sont reformatées pour plus de lisibilité.

1.5.4 CONFIGURER POSTGRESQL POUR PGBADGER

- Minimum :
  - `log_destination`, `log_line_prefix` et `lc_messages=C`
- Base :
  - `log_connections`, `log_disconnections`
  - `log_checkpoints`, `log_lock_waits`, `log_temp_files`
  - `log_autovacuum_min_duration`
- Pour un audit :
  - `log_min_duration_statement = 0` (attention !)

pgBadger a besoin d'un minimum d'informations dans les traces : timestamp (`%t`), pid (`%p`) et numéro de ligne dans la session (`%l`). Il n'y a pas de conseil particulier sur la destination des traces (en dehors de `eventlog` que pgBadger ne sait pas traiter). De même, le préfixe des traces est laissé au choix de l'utilisateur. Dans certains cas, il faudra le préciser à pgBadger avec l'option `--prefix` (par exemple si la valeur de `log_line_prefix` a changé entre le début et la fin du fichier). Une configuration courante et vraiment informative est par exemple :

```
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Noter que même sans cela, pgBadger essaie de récupérer les informations sur les adresses IP, les utilisateurs connectés, les bases de données à partir des traces sur les connexions. Ajouter le joker `%e` (code SQLState) permet d'obtenir un tableau sur les erreurs.

La langue des traces doit être l'anglais (`lc_messages` à `C`), ce qui de toute manière est la valeur conseillée.

Pour tracer les requêtes, il est préférable de passer par `log_min_duration_statement` plutôt que `log_statement` et `log_duration` : pgBadger fera plus facilement l'association entre chaque requête et sa durée :

```
log_min_duration_statement = 0
log_statement = none
log_duration = off
```

Comme déjà dit plus haut, `log_min_duration_statement = 0` peut générer un énorme volume de traces et n'est souvent configuré ainsi que le temps d'un audit. Avec une valeur supérieure, pgBadger ne verra absolument pas les requêtes les plus rapides.

Il est aussi conseillé de tirer parti d'autres informations dans les traces :

- `log_checkpoints` pour des statistiques sur les checkpoints ;
- `log_connections` et `log_disconnections` pour des informations sur les connexions et déconnexions ;
- `log_lock_waits` pour des statistiques sur les verrous en attente ;
- `log_temp_files` pour des statistiques sur les fichiers temporaires ;
- `log_autovacuum_min_duration` pour des statistiques sur l'activité de l'autovacuum.

1.5.5 OPTIONS DE PGBADGER

- Génération :

```
pgbadger ... -o rapport.html postgresql-Mon.log postgresql-Tue.log …
```

- Très nombreuses options, dont :
  - `--outfile`
  - `--prefix`
  - `--begin` / `--end`
  - `--dbname`, `--dbuser`, `--dbclient`, `--appname`
  - `--jobs`

Pour l'utilisation la plus simple, il suffit de fournir au script en paramètre les noms des différents fichiers de traces, éventuellement compressés, pour obtenir le rapport sur tous les événements qu'il y trouvera.

Il existe énormément d'options[9] , et Gilles Darold en rajoute fréquemment. L'aide fournie sur le site web officiel les cite intégralement.

Parmi les plus utiles, `--outfile` permet d'indiquer le nom du rapport (par défaut `out.html`).

`--prefix` permet de préciser une valeur de `log_line_prefix` si la détection automatique échoue.

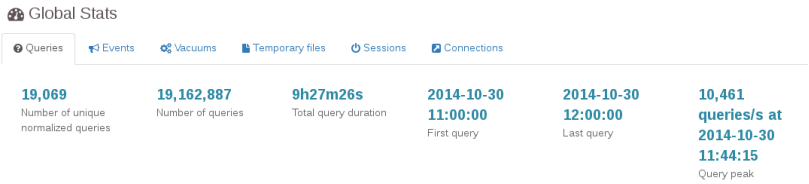
Le rapport peut être restreint à une tranche horaire précise avec `--begin` et `--end` (par exemple `--begin '2019-04-01 16:00:00`).

Pour mieux cibler le rapport, il est possible de restreindre l'analyse à un utilisateur (`--dbuser`), une base de données (`--dbname`), une machine cliente (`--dbclient`), ou une application (`--appname`, si elle définit bien `application_name` à la connexion).

`--jobs` permet de paralléliser la lecture des traces sur plusieurs processeurs. Attention, de très gros fichiers les satureront probablement plusieurs minutes.

[9] https://pgbadger.darold.net/documentation.html#SYNOPSIS

1.5.6 PGBADGER : EXEMPLE 1



Au tout début du rapport, pgBadger donne des statistiques générales sur les fichiers de traces.

Dans les informations importantes se trouve le nombre de requêtes normalisées. En fait, des requêtes telles que :

```
SELECT * FROM utilisateurs WHERE id = 1;
```

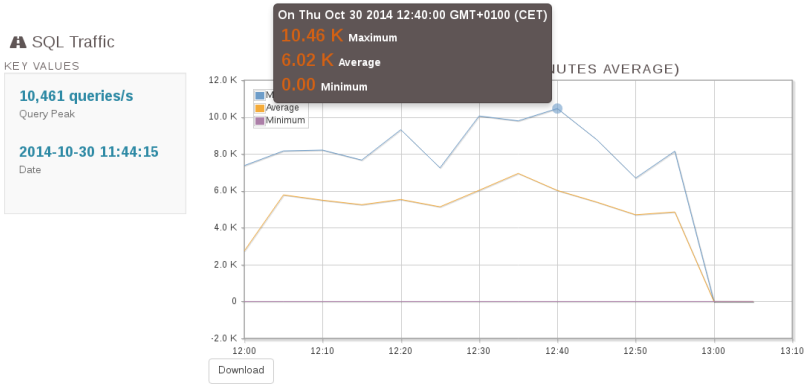
et

```
SELECT * FROM utilisateurs WHERE id = 2;
```

sont différentes car elles ne vont pas récupérer la même fiche utilisateur. Cependant, en enlevant la partie constante, les requêtes sont identiques. La seule différence est la ligne récupérée mais pas la requête. pgBadger est capable de faire cette différence. Toute constante, qu'elle soit de type numérique, textuelle, horodatage ou booléenne, peut être supprimée de la requête. Dans l'exemple ci-dessus, pgBadger a comptabilisé environ 19 millions de requêtes, mais seulement 19 069 requêtes différentes après normalisation. Ceci est important dans le fait où nous n'allons pas devoir travailler sur plusieurs millions de requêtes mais « seulement » sur 19 000.

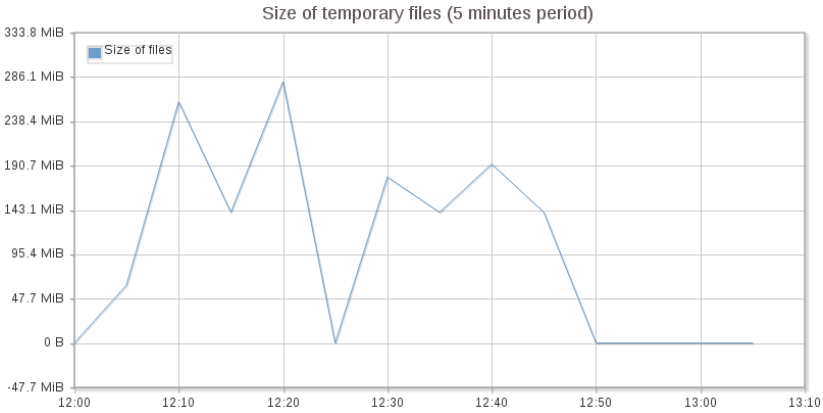
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 9 heures d'exécution de requêtes. Cependant, les traces ne couvrent que 11 h à 12 h, soit une heure. Cela indique que le serveur est assez sollicité. Il est fréquent que la durée d'exécution sérielle des requêtes soit plusieurs fois plus importantes que la durée des traces.

1.5.7 PGBADGER : EXEMPLE 2



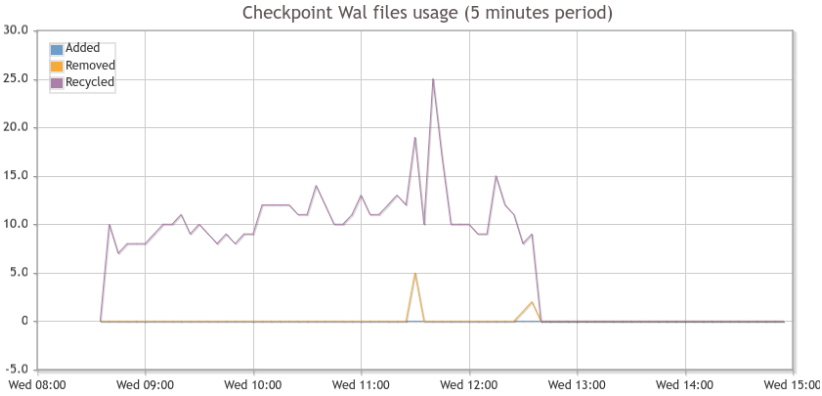
Ce graphe indique le nombre de requêtes par seconde : en fait, environ 33 requêtes/s en pointe.

1.5.8 PGBADGER : EXEMPLE 3



Ce graphe affiche en vert le nombre de fichiers temporaires créés sur la durée des traces. La ligne bleue correspond à la taille des fichiers. Nous remarquons ainsi la création à peu près régulière de fichiers temporaires, à raison d'environ 200 Mo par tranche de 5 minutes.

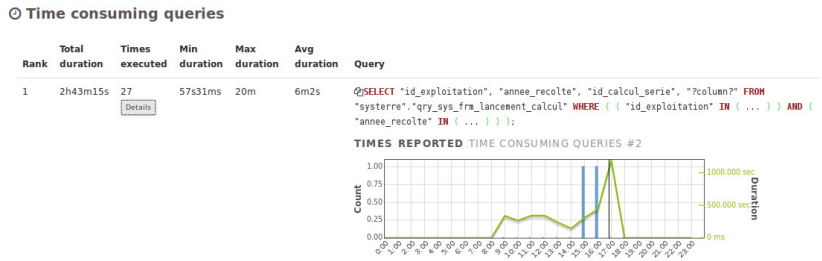
1.5.9 PGBADGER : EXEMPLE 4



De grosses écritures peuvent mener à un nombre important de journaux. Cette courbe montre une création relativement régulière de journaux. En fait, il s'agit uniquement de recyclage de journaux existants : PostgreSQL n'a pas à en créer et en initialiser en masse. Le pic n'est que de 5 journaux de 16 Mo à la minute.

Plus bas dans l'onglet *Checkpoints* figurent des informations sur l'écart (en octets) entre deux checkpoints, la durée d'écriture, la durée de synchronisation sur le disque, et le nombre d'avertissements sur des checkpoints non périodiques.

1.5.10 PGBADGER : EXEMPLE 5



Le plus important est certainement l'onglet *Top/Time consuming queries*. Il liste les requêtes qui ont pris le plus de temps, que ce soit parce que les requêtes en question sont vraiment très lentes ou parce qu'elles sont exécutées un très grand nombre de fois.

Ci-dessus n'est affichée que la première requête de la liste. Le bouton *Details* permet d'afficher la courbe indiquant les heures d'occurrences et les durées.

Nous remarquons d'ailleurs dans cet exemple qu'avec la seule requête affichée, nous arrivons à un total de 2 h 43. Le premier exemple nous indique que l'exécution sérielle des requêtes aurait pris 9 heures. En ne travaillant que sur elle, nous travaillons en fait sur presque le tiers du temps total d'exécution des requêtes comprises dans les traces.

Les autres requêtes les plus consommatrices ne sont pas listées ici, mais il est peu probable que l'on ait à travailler sur les 19 000 requêtes normalisées. Il est fréquent que l'essentiel de la charge soit contenu dans les quelques premières requêtes du tableau. Ce sont évidemment les premières cibles pour une tentative d'optimisation : réécriture, modification du paramétrage, index dédiés…

1.5.11 LOGWATCH

- Outil externe écrit en Perl
  - https://sourceforge.net/projects/logwatch/
  - Licence MIT
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs
- Exemple :

```
/usr/sbin/logwatch --detail Med --service postgresql --range All
```

Surveiller ses journaux applicatifs (ou fichiers de trace) est une activité nécessaire mais bien souvent rébarbative. De nombreux programmes existent pour nous faciliter la tâche, mais le propre de ces programmes est d'être exhaustif. De plus, ils demandent une action de la part de l'administrateur, à savoir : penser à aller les regarder.

logwatch est une petite application permettant d'analyser les journaux de nombreux services et de produire un rapport synthétique. Le nombre de services connus est impressionnant et il est simple d'en ajouter de nouveaux. logwatch est le plus souvent intégré à votre distribution Linux. Depuis la version 7.4.2 il sait analyser les traces de PostgreSQL.

Après son installation, il se lancera tous les jours grâce au fichier `/etc/cron.daily/00logwatch`. Vous recevrez tous les jours par mail les rapports des événements importants détectés

dans les fichiers de traces. Vous pouvez aussi le lancer manuellement ainsi :

```
/usr/sbin/logwatch --detail Low --service postgresql --range All
```

`--range All` indique que l'on veut un rapport sur tous les fichiers de traces existants. La valeur par défaut est `Yesterday`. Pour n'avoir un rapport que sur la journée, choisir `Today`.

Avec le niveau de détail minimal (`--detail` à `Low`), seuls les messages de type `FATAL`, `PANIC` et `ERROR` seront remontés. Avec la valeur `Med`, apparaîtront aussi les messages de type `WARNING` et `HINTS`.

Exemple de résultat :

```
################### Logwatch 7.3.6 (05/19/07) ####################
       Processing Initiated: Tue Dec 13 12:28:46 2011
       Date Range Processed: all
       Detail Level of Output: 5
       Type of Output/Format: stdout / text
       Logfiles for Host: devel
 ################################################################
--------------------- PostgreSQL Begin ------------------------
Fatals:
-------
9 times:
[2011-12-04 04:28:46 +/-9 day(s)]  password authentication failed for
                                   user "postgres"
8 times:
[2011-11-21 10:15:01 +/-11 day(s)]  terminating connection due to
                                    administrator command
…
Errors:
-------
7 times:
[2011-11-14 12:20:44 +/-58 minute(s)]  relation "COUNTRIES" does not exist
5 times:
[2011-11-14 12:21:24 +/-59 minute(s)]  syntax error at or near
                                       "role_permission_view"
…
Warnings:
---------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  terminating connection because of
                                   crash of another server process
Hints:
------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  In a moment you should be able to
```

```
                                reconnect to the database and repeat
                                you command.
14 times:
[2011-12-08 09:47:16 +/-19 day(s)]  No function matches the given name and
                                     argument types. You might need to add
                                     explicit type casts.
---------------------- PostgreSQL End ------------------------
###################### Logwatch End #########################
```

logwatch dispose de nombreuses options, et la page de manuel est certainement la meilleure documentation à l'heure actuelle.

1.5.12 TAIL_N_MAIL

- Outil externe écrit en Perl
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs

tail_n_mail est un outil écrit par la société EndPointCorporation. Son but est d'analyser périodiquement le contenu des fichiers de traces et d'envoyer un mail en cas de la détection d'un motif d'intérêt. Par exemple, il peut envoyer un mail lorsqu'il rencontre un message de niveau `PANIC` ou `FATAL` dans les traces. Ainsi une personne d'astreinte sera prévenue rapidement et pourra agir en conséquence.

1.5.13 CONFIGURER TAIL_N_MAIL

```
EMAIL: astreinte@dalibo.com
MAILSUBJECT: HOST Postgres fatal errors (FILE)
FILE: /var/log/postgresql-%Y-%m-%d.log
INCLUDE: PANIC:
INCLUDE: FATAL:
EXCLUDE: database ".+" does not exist
INCLUDE: temporary file
INCLUDE: reloading configuration files
```

Les clés `INCLUDE` et `EXCLUDE` permettent d'indiquer les motifs à inclure et à exclure respectivement. Tout motif non inclus ne sera pas pris en compte. Cette configuration permet donc d'envoyer un mail à l'adresse `astreinte@dalibo.com` à chaque fois qu'un message contenant les mots `PANIC`, `FATAL`, `temporary file` ou `reloading configuration files` sont enregistrés dans les traces. Par contre, tous les messages contenant la phrase `database`

`... does not exist` ne sont pas pris en compte.

1.5.14 TAIL_N_MAIL : EXEMPLE

Exemple:

```
[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement: user=root,db=rojogrande
        FATAL: password authentication failed for user "root"
```

Voici un exemple de mail envoyé:

```
Matches from /var/log/postgresql/postgresql-10-main.log: 42
Date: Fri Jan  1 10:34:00 2010
Host: pollo

[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement:  user=root,db=rojogrande FATAL:  password authentication failed
                                            for user "root"

[2] Between lines 147999 and 148213, occurs 2 times.
First:  Jan  1 10:31:01 rojogrande postgres[3561]
Last:   Jan  1 10:31:10 rojogrande postgres[15312]
Statement: FATAL  main: write to worker pipe failed -(9) Bad file descriptor

[3] (from line 152341)
PANIC:  could not locate a valid checkpoint record
```

1.6 STATISTIQUES D'ACTIVITÉ

- Configuration
- Liste des vues statistiques
- Outils externes de classement

Les statistiques sont certainement les informations les plus simples à récupérer par un système de supervision. Il faut dans un premier temps s'assurer que la configuration est adéquate. Ceci fait, il est possible de lire les statistiques disponibles dans les vues proposées par défaut. Enfin, il existe quelques outils capables de récupérer des informations

provenant des tables statistiques de PostgreSQL. Leur mise en place permettra une supervision facilitée.

1.6.1 STATISTIQUES D'ACTIVITÉ - CONFIGURATION 1

- Tracer l'activité :
  - `track_activities` = `on`
- S'assurer que les requêtes ne sont pas tronquées :
  - `track_activity_query_size` = `10000` ou +
- Récupérer l'identifiant de requête
  - `compute_query_id` = on

Il est important d'avoir des informations sur les sessions en cours d'exécution sur le serveur. Cela se fait grâce au paramètre `track_activities`. Il est à `on` par défaut. Ainsi, dans la vue `pg_stat_activity` qui liste les sessions, les colonnes `xact_start`, `query_start`, `state_change`, `wait_event_type`, `wait_event`, `state` et `query` sont renseignées.

```
bench=# SELECT * FROM pg_stat_activity
        WHERE backend_type = 'client backend' \gx

-[ RECORD 1 ]----+--------------------------------------------------------------------------
datid            | 16425
datname          | bench
pid              | 3006
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | test
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2021-02-05 14:29:42.833102+00
xact_start       | 2021-02-05 17:28:12.157115+00
query_start      | 2021-02-05 17:28:12.157115+00
state_change     | 2021-02-05 17:28:12.157117+00
wait_event_type  |
wait_event       |
state            | active
backend_xid      |
backend_xmin     | 186938
query_id         |
query            | select * from pg_stat_activity where backend_type = 'client backend'
backend_type     | client backend
```

Il est à noter que la requête indiquée dans la colonne `query` est tronquée à 1024 caractères par défaut. En pratique, cette limite est vite atteinte par de longues requêtes. Il est conseillé de l'augmenter à quelques kilooctets (paramètre `track_activity_query_size`).

Depuis la version 14, il est possible d'avoir en plus l'identifiant de la requête. Pour cela, il faut activer le paramètre `compute_query_id`.

1.6.2 STATISTIQUES D'ACTIVITÉ - CONFIGURATION 2

- `track_counts` = `on`
- `track_io_timing` = `on`
- `track_functions` = `off` / `pl` / `all`

Par défaut, le paramètre `track_counts` est à `on`. Le collecteur de statistiques est alors capable de récupérer des informations sur des nombres de lignes lues, insérées, mises à jour, supprimées, vivantes, mortes, etc., et de blocs lus dans le cache de PostgreSQL (*hit*) ou en dehors (*read*).

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

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

Avant d'activer `track_io_timing` sur une machine peu performante, vérifiez avec l'outil `pg_test_timing`[10] que la quasi-totalité des appels dure moins d'une nanoseconde.

PostgreSQL sait aussi récupérer des statistiques sur les routines stockées. Il faut activer le paramètre `track_functions` qui a trois valeurs : `off` pour ne rien récupérer, `pl` pour récupérer les statistiques sur les procédures stockées en `PL/*` et `all` pour récupérer les statistiques des fonctions quelque soit leur langage (notamment celles en C). Les résultats (nombre et durée d'exécution) peuvent se suivre dans la vue `pg_stat_user_functions`.

---

[10] https://docs.postgresql.fr/current/pgtesttiming.html

1.6.3 STATISTIQUES D'ACTIVITÉ - CONFIGURATION 3

- `stats_temp_directory`
  - répertoire contenant les fichiers temporaires des statistiques
  - copié vers `pg_stat` lors d'un arrêt propre
  - à monter sur du tmpfs

Le collecteur de statistiques fonctionne ainsi :

- il est lancé au démarrage de PostgreSQL (il est d'ailleurs impossible de le désactiver complètement) ;
- il collecte ses statistiques dans le répertoire ciblé par le paramètre `stats_temp_directory` ;
- il met à jour les fichiers dès que les autres processus lui fournissent des statistiques ;
- à l'arrêt de PostgreSQL, il recopie les fichiers du répertoire temporaire dans le répertoire `$PGDATA/pg_stat`.

L'intérêt de ce fonctionnement est de pouvoir copier le fichier de statistiques sur un disque très rapide (comme un disque SSD), voire dans de la mémoire montée en disque comme `tmpfs` (c'est d'ailleurs le défaut sur Debian).

1.6.4 INFORMATIONS INTÉRESSANTES À RÉCUPÉRER

Sur :

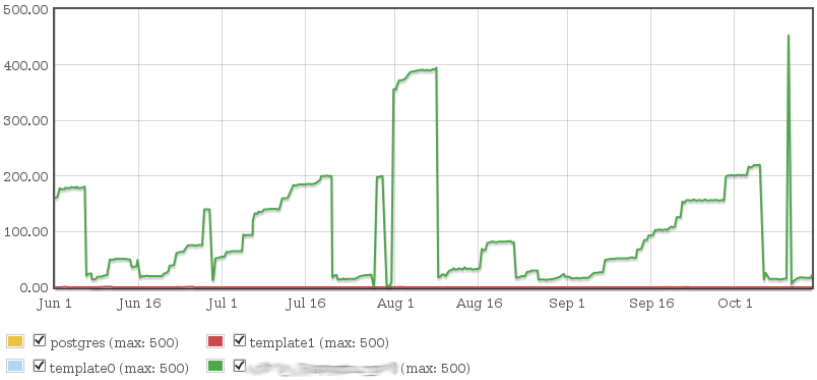
- l'activité
- l'instance
- les bases
- les tables
- les index
- les fonctions

Au niveau des statistiques, il existe un grand nombre d'informations intéressantes à récupérer. Cela va des informations sur l'activité en cours (nombre de connexions, noms des utilisateurs connectés, requêtes en cours d'exécution), à celles sur les bases de données (nombre d'écritures, nombre de lectures dans le cache), à celles sur les tables, index et fonctions.

Les connaître toutes présente peu d'intérêt car seulement certaines sont vraiment intéressantes à superviser. Nous allons voir ici quelques exemples.

1.6.5 NOMBRE DE CONNEXIONS PAR BASE

```
SELECT datname, numbackends FROM pg_stat_database;
SELECT datname, count(*) FROM pg_stat_activity
  WHERE datname IS NOT NULL
  GROUP BY datname;
```



Il est souvent intéressant de connaître le nombre de personnes connectées. Cela permet notamment de s'assurer que la configuration du paramètre `max_connections` est suffisamment élevée pour ne pas voir des demandes de connexion être refusées.

Il est aussi intéressant de pouvoir dénombrer le nombre de connexions par bases. Cela permet d'avoir une idée de la charge sur chaque base. Une base ayant de plus en plus d'utilisateurs et souffrant de performances devra peut-être être placée seule sur un serveur.

Il est aussi possible d'avoir le nombre de connexions par :

- utilisateur :

```
SELECT usename, count(*) FROM pg_stat_activity WHERE usename IS NOT NULL GROUP BY usename;
```

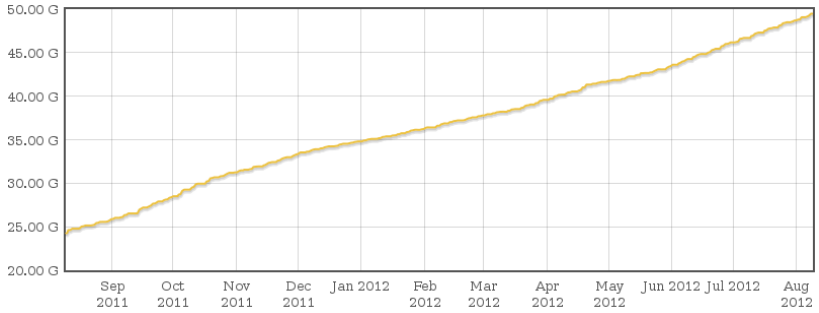
- application cliente :

```
SELECT application_name, count(*) FROM pg_stat_activity GROUP BY application_name;
```

Le graphe affiché ci-dessus montre l'utilisation des connexions sur différentes bases. Les bases systèmes ne sont pas du tout utilisées. Seule la base utilisateur reçoit un grand nombre de connexions. Il y a même eu deux pics, un à environ 400 connexions et un autre à 450 connexions.

1.6.6 TAILLE DES BASES

```
SELECT datname, pg_database_size(oid) FROM pg_database;
```

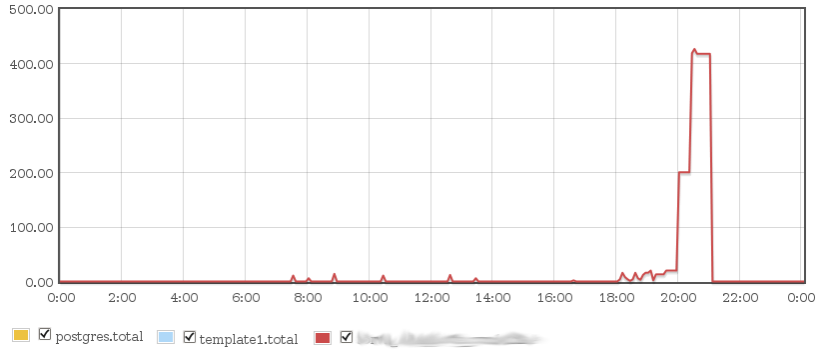


La volumétrie des bases est une autre information fréquemment demandée. L'exécution de cette requête toutes les cinq minutes permettra de suivre l'évolution de la taille des bases.

Le graphe ci-dessus montre une montée en volumétrie assez importante, la base ayant doublé en un an.

1.6.7 NOMBRE DE VERROUS

```
SELECT d.datname, count(*) FROM pg_locks l
JOIN pg_database d ON l.database=d.oid
GROUP BY d.datname ORDER BY d.datname;
```



Autre demande fréquente : pouvoir suivre le nombre de verrous. La requête ci-dessus récupère le nombre de verrous posés par base. Il est aussi possible de récupérer les types de verrous, ou de ne prendre en compte que certains types de verrous.

Le graphe montre une utilisation très limitée des verrous. En fait, on observe surtout une grosse utilisation des verrous entre 20h et 21h30. Si le graphe montrait plusieurs jours d'affilé, on pourrait s'apercevoir que le pic est présent tous les jours à ce moment-là. En fait, il s'agit de la sauvegarde. La sauvegarde pose un grand nombre de verrous, des verrous très légers qui vont bloquer très peu de monde, mais néanmoins, ils sont présents.

1.6.8 ET UN GRAND NOMBRE D'AUTRES INFORMATIONS

- Ratio de lecture du cache (souvent appelé *hit ratio*)
- Retard de réplication
- Nombre de transactions par seconde

Les trois exemples proposés ci-dessus ne sont que les exemples les plus marquants. Beaucoup d'autres informations sont récupérables. On peut citer par exemple :

- le ratio de lecture dans le cache de PostgreSQL ;
- le retard de la réplication interne de PostgreSQL (envoi, écriture, application) ;
- le nombre de transactions par seconde ;
- l'activité d'une table (en nombre de lectures, insertions, suppressions, modifications) ;
- le nombre de parcours séquentiels et de parcours d'index ;
- etc.

1.6.9 OUTILS

- Beaucoup d'outils existent pour exploiter les statistiques :
  - Munin, Nagios, Zabbix + sondes PG
- Dédiés à PostgreSQL :
  - pg_stat_statements
  - PoWA

Il existe de nombreux outils utilisables avec les statistiques. Les plus connus sont Munin (sondes déjà intégrées), Nagios et Zabbix. Pour ces deux derniers, il est nécessaire d'ajouter des sondes comme `check_postgres` et `check_pgactivity` évoquées plus haut.

`pg_stat_statements` est un module contrib de PostgreSQL, donc non installé par défaut. Il collecte des statistiques sur toutes les requêtes exécutées. Son contenu est souvent extrêmement instructif.

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire historisant les informations collectées par `pg_stat_statements`, et fournissant une interface graphique permettant d'observer en temps réel les requêtes normalisées les plus consommatrices d'une instance selon plusieurs critères.

1.6.10 MUNIN

- Scripts Perl
- Sondes PostgreSQL incluses
- Récupère les statistiques toutes les 5 min
- Crée des pages HTML statiques et des fichiers PNG
  - donc des graphes

Munin est à la base un outil de métrologie utilisé par les administrateurs systèmes. Il comprend un certain nombre de sondes capables de récupérer des informations telles que la charge système, l'utilisation de la mémoire et des interfaces réseau, l'espace libre d'un disque, etc. Des sondes lui ont été ajoutées pour pouvoir surveiller certains services comme Sendmail, Postfix, Apache, et même PostgreSQL.

Munin est composé de deux parties : les sondes et le générateur de rapports. Les sondes sont exécutées toutes les cinq minutes. Elles sont généralement écrites en Perl. Elles récupèrent les informations et les stockent dans des bases BerkeleyDB. Ensuite, le générateur de rapports lit les bases en question pour générer des graphes au format PNG. Des pages HTML sont créées pour faciliter l'accès aux graphes.

Munin est inclus dans les distributions habituelles.

1.6.11 NAGIOS

- Outil GPL, sur https://www.nagios.org/
- Nombreux concurrents et équivalents
- Sondes dédiées à PostgreSQL : `check_postgres` et `check_pgactivity`

Nagios est un outil très connu de supervision. Il dispose par défaut de quelques sondes, principalement système. Il existe aussi des outils concurrents (Naemon, Icinga 2…) ou des surcouches compatibles au niveau des sondes.

Pour PostgreSQL, il est possible de le coupler à des sondes dédiées, comme `check_postgres` ou `check_pgactivity`, déjà évoquées, pour qu'il puisse récupérer un certain nombre d'informations statistiques de PostgreSQL. Ne pas oublier de compléter par des sondes plus classiques pour les I/O, le CPU, la RAM, etc.

1.6.12 OUTILS - ZABBIX

- Outil GPL, sur https://www.zabbix.com/
- Sonde `check_postgres.pl`
- Template pg-monz
  - https://pg-monz.github.io/pg_monz/index-en.html

Zabbix est certainement le deuxième outil opensource le plus utilisé pour la supervision. Son avantage par rapport à Nagios est qu'il est capable de faire des graphes directement et qu'il dispose d'une interface plus simple d'approche.

Là-aussi, la sonde `check_postgres.pl` lui permet de récupérer des informations sur un serveur PostgreSQL.

1.6.13 OUTILS - PG_STAT_STATEMENTS

- Module contrib de PostgreSQL
- Récupère et stocke des statistiques d'exécution des requêtes
- Les requêtes sont normalisées
- Pas d'historisation

`pg_stat_statements` est une extension issue des « contrib » de PostgreSQL, donc livrée avec lui, mais non installée par défaut. Sa mise en place nécessite le préchargement de bibliothèques dans la mémoire partagée (paramètre `shared_preload_libraries`), et donc le redémarrage de l'instance.

Une fois installé et configuré, des mesures (nombre de blocs lus dans le cache, hors cache, ...) sont collectées sur toutes les requêtes exécutées, et elles sont stockées avec les requêtes normalisées. Ces données sont ensuite exploitables en interrogeant la vue `pg_stat_statements`. À noter que ces statistiques sont cumulées sans être historisées, il est donc souvent difficile d'identifier quelle requête est la plus consommatrice à un instant donné, à moins de réinitialiser les statistiques.

Voir aussi la documentation officielle : https://docs.postgresql.fr/current/pgstatstatements.html

---

1.6.14 OUTILS - POWA

- Site officiel : https://github.com/powa-team
- Licence : PostgreSQL
- Surveillance de l'activité SQL
- Captures des statistiques collectées par `pg_stat_statements`
  - et d'autres extensions
- Interface graphique : activité des requêtes en temps réel
- Dépôt GitHub
  - archiveur : https://github.com/powa-team/powa-archivist
  - UI web : https://github.com/powa-team/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire, sous licence PostgreSQL.

Tout comme pour l'extension standard `pg_stat_statements`, sa mise en place nécessite la modification du paramètre `shared_preload_libraries`, et donc le redémarrage de l'instance. Il faut également créer une nouvelle base de données dans l'instance. Par ailleurs, PoWA repose sur les statistiques collectées par `pg_stat_statements`, celui-ci doit donc être également installé.

Une fois installé et configuré, l'outil va récupérer à intervalle régulier les statistiques collectées par `pg_stat_statements`, les stocker et les historiser.

Il tire aussi partie d'autres extensions comme HypoPG, pg_qualstats, pg_stat_kcache…

L'outil fournit également une interface graphique permettant d'exploiter ces données, et donc d'observer en temps réel l'activité de l'instance. Cette activité est présentée sous forme de graphiques interactifs et de tableaux permettant de trier selon divers critères (nombre d'exécution, blocs lus hors cache…) les différentes requêtes normalisées sur l'intervalle de temps sélectionné.

1.7 CONCLUSION

- Un système est pérenne s'il est bien supervisé
- Supervision automatique
  - configuration des traces
  - configuration des statistiques
  - mise en place d'outils d'historisation

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, l'installation d'un outil d'historisation, de création de graphes et de génération d'alertes, est obligatoire pour pouvoir tirer profit des informations fournies par PostgreSQL.

1.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/h1_quiz

1.9 TRAVAUX PRATIQUES

Analyse de traces avec pgBadger

But : Analyser un journal de traces avec pgBadger

> S'assurer que la configuration suivante est en place, au moins le temps de l'audit :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

> Si cela n'a pas déjà été fait aujourd'hui, lancer une session `pgbench` de 10 minutes dans la base `pgbench`.

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

> Repérer où sont vos fichiers de traces, et notamment ceux d'aujourd'hui.

> Générer dans `/tmp` un rapport pgBadger sur toute la journée en cours. La documentation est entre autres sur le site de l'auteur : https://pgbadger.darold.net/documentation.html. Combien de requêtes a-t-il détecté ?

> Ouvrir le rapport dans un navigateur. Dans l'onglet *Overview*, à quoi correspondent les pics de trafic en nombre de requêtes ?

> Dans l'onglet *Top*, chercher l'histogramme de la répartition des durées des requêtes.

> Quelles sont les requêtes les plus lentes, prises une à une ?

> Quelles sont les requêtes qui ont consommé le plus de temps au total ?

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

Analyse de traces avec pgBadger

> S'assurer que la configuration suivante est en place, au moins le temps de l'audit :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

On vérifie que la configuration est bien active avec :

```
postgres@pgbench=# show log_min_duration_statement ;

 log_min_duration_statement
----------------------------
 0
(1 ligne)
```

Au besoin, le paramétrage peut soit se faire dans `postgresql.conf` (ne pas oublier de recharger la configuration), soit au niveau de la base de données (`ALTER DATABASE pgbench SET log_min_duration_statement = 0`).

> Si cela n'a pas déjà été fait aujourd'hui, lancer une session `pgbench` de 10 minutes dans la base `pgbench`.

Pour créer la base si elle n'existe pas déjà :

```
$ /usr/pgsql-14/bin/pgbench -i --foreign-keys -d pgbench -U testperf
```

Pour lancer un traitement de 10 minutes avec 20 clients répartis sur 2 processeurs :

```
$ /usr/pgsql-14/bin/pgbench -U testperf pgbench \
    --client=20 --jobs=2 -T 600 --no-vacuum -P 10
```

> 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 de la distribution :

```
# yum install pgbadger
```

Pour la version la plus à jour, 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.

```
$ curl -LO https://github.com/darold/pgbadger/archive/refs/tags/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`.

> Repérer où sont vos fichiers de traces, et notamment ceux d'aujourd'hui.

Par défaut, les traces sur Red Hat/CentOS/Rocky Linux sont dans `/var/lib/pgsql/14/data/log`.

Elles ont pu être déplacées dans `/var/lib/pgsql/traces` au fil de ces TP.

> Générer dans `/tmp` un rapport pgBadger sur toute la journée en cours. La documentation est entre autres sur le site de l'auteur : https://pgbadger.darold.net/documentation.html. Combien de requêtes a-t-il détecté ?

Dans sa plus simple expression, la commande est la suivante, à exécuter avec un utilisateur ayant accès aux traces :

```
$ cd /var/lib/pgsql/14/data/log
$ pgbadger -o /tmp/pgabdger-aujourdhui.html postgresql-2022-05-02*.log
```

Il peut arriver que la détection du format échoue, auquel cas il faut préciser la valeur du paramètre `log_line_prefix` (`%m [%p]` par défaut sur RHEL, mais l'auteur préconise `'%t [%p]: db=%d,user=%u,app=%a,client=%h '` que l'on a configuré initialement) avec l'option `-p`.

Le paramètre `-j` permet de paralléliser le travail sur plusieurs processeurs.

On peut vouloir des statistiques plus fines que la moyenne par défaut de 5 minutes (`-a`), ou cibler sur une page de dates précises (`-b` et `-e`). Enfin, préciser le fuseau horaire du serveur (`-z`) est souvent utile pour éviter des décalages dans les heures.

```
$ pgbadger -o /tmp/pgabdger-aujourdhui.html -p '%m [%p] ' -j 2  -a 1  \
           -b '2022-05-02 09:00:00' -e '2022-05-02 18:00:00' -Z '+02' \
           postgresql-2022-05-02*.log
...
[========================>] Parsed 170877924 bytes of 170877924 (100.00%),
                                        queries: 503580, events: 51
LOG: Ok, generating html report...
```

En l'occurence, pgBadger a détecté ici un demi-million de requêtes.

> Ouvrir le rapport dans un navigateur. Dans l'onglet *Overview*, à quoi correspondent les pics de trafic en nombre de requêtes ?

Il s'agit probablement des tests avec pgbench.

> Dans l'onglet *Top*, chercher l'histogramme de la répartition des durées des requêtes.

Le graphique indique que l'essentiel des requêtes doit figurer dans la tranche « 0ms-1ms ». La version tableau de ce graphique liste une poignée de requête autour de la seconde ou plus, que l'on voit également dans les *slowest queries* en-dessous.

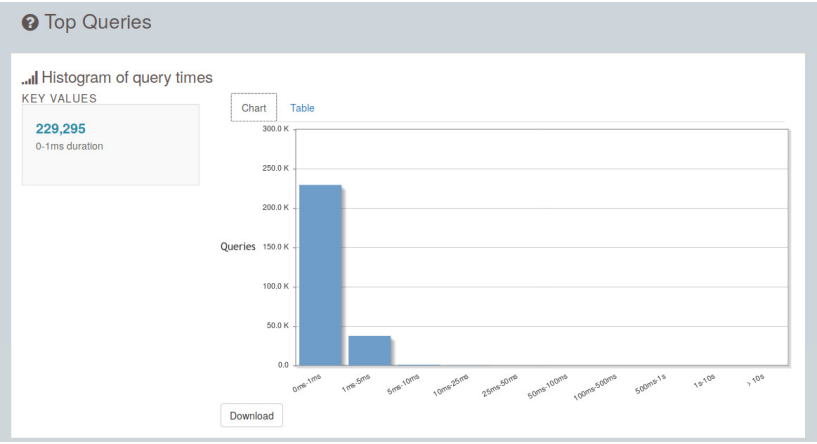


Figure 2: Nombre de requêtes par durée

> Quelles sont les requêtes les plus lentes, prises une à une ?

On les trouve dans *Top/Slowest individual queries*. Ce sont probablement les ordres `COPY` des différents essais avec `pg_restore`, et les ordres `CREATE DATABASE`.

> Quelles sont les requêtes qui ont consommé le plus de temps au total ?

On les trouve dans *Top/Time consuming queries*. Cet onglet est très intéressant pour repérer les requêtes relativement rapides, mais qui représentent la véritable charge de l'instance au quotidien.

Il s'agit très probablement des requêtes de `pgbench` :

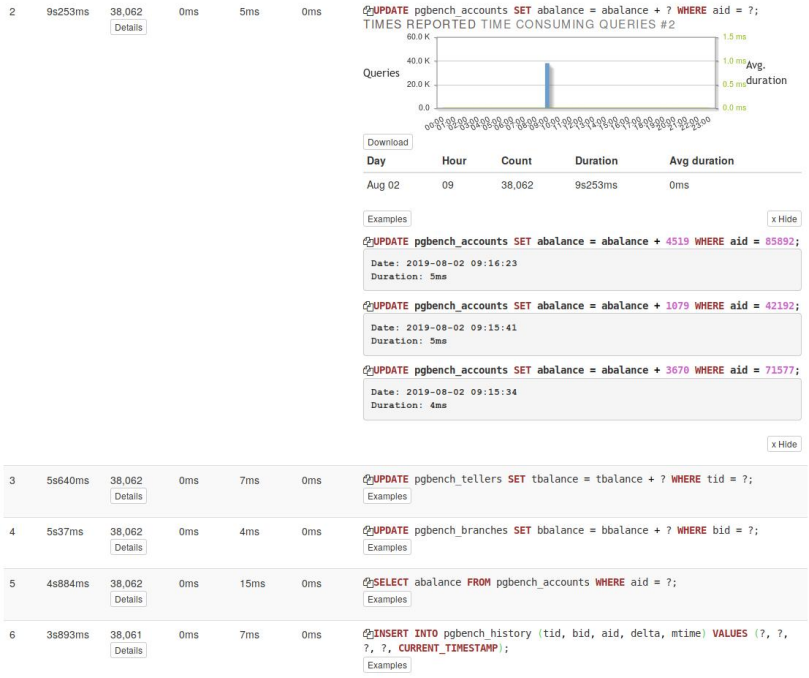


Figure 3: Requêtes les plus consommatrices en temps

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **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

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.