DALIBO
L'expertise PostgreSQL

Supervision

Ce document est téléchargeable gratuitement dans la base de connaissance DALIBO.

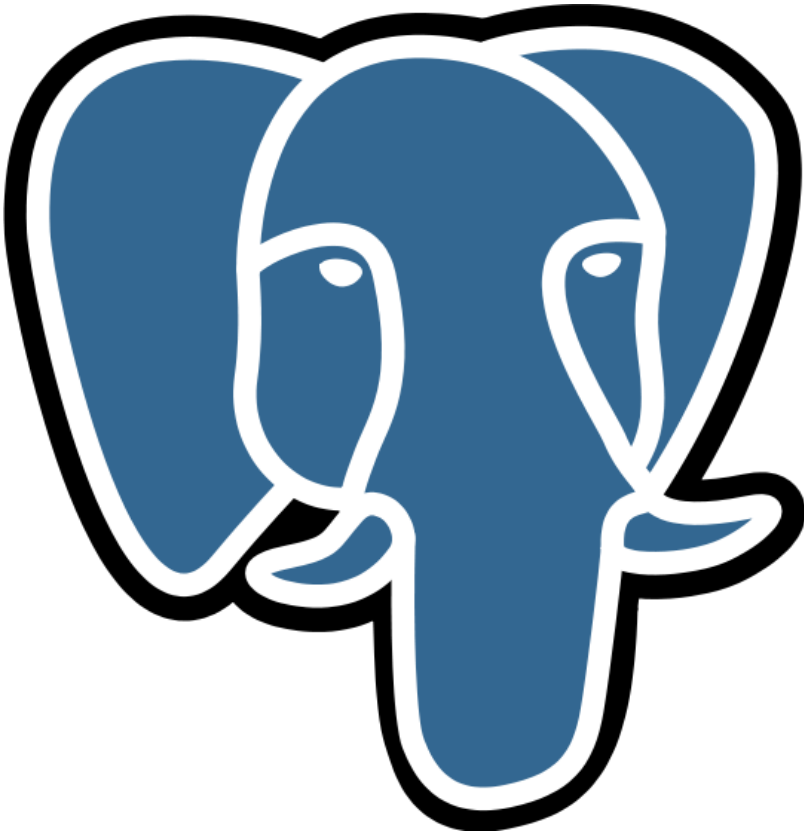
Pour toute information complémentaire, contactez :
formation@dalibo.com

Table des matières

Supervision....4
1 Introduction....5
1.1 Menu....5
1.2 Licence Creative Commons CC-BY-NC-SA....6
2 Politique de supervision....7
2.1 Objectifs de la supervision....7
2.2 Acteurs concernés....8
2.3 Exemples d'indicateurs - système d'exploitation....9
2.4 Exemples d'indicateurs - base de données....9
3 La supervision avec PostgreSQL....11
3.1 Informations internes....11
3.2 Outils externes....12
3.3 check_postgres....12
3.4 check_pgactivity....13
4 Traces....16
4.1 Configuration....16
4.2 Configuration : où tracer....17
4.3 Configuration : niveau des traces....18
4.4 Configuration : tracer les requêtes....19
4.5 Configuration : tracer certains comportements....20
4.6 Configuration : divers....22
4.7 Informations intéressantes à récupérer....22
4.8 Durée d'exécution des requêtes....23
4.9 Messages PANIC....24
4.10 Rechargement de la configuration....25
4.11 Fichiers temporaires....25
4.12 Outils....26
4.13 Utiliser pgBadger....27
4.14 Configurer PostgreSQL pour pgBadger....27
4.15 Options de pgBadger....28
4.16 pgBadger : exemple 1....31
4.17 pgBadger : exemple 2....32
4.18 pgBadger : exemple 3....33
4.19 pgBadger : exemple 4....33
4.20 Utiliser logwatch....34
4.21 Configurer logwatch....34
4.22 Options de logwatch....35
4.23 logwatch : exemple 1....35
4.24 logwatch : exemple 2....36
4.25 Utiliser tail_n_mail....38
4.26 Configurer tail_n_mail....38
4.27 tail_n_mail : exemple....39
5 Statistiques....40
5.1 Statistiques - configuration 1....40
5.2 Statistiques - configuration 2....42

5.3 Statistiques - configuration 3....43
5.4 Informations intéressantes à récupérer....44
5.5 Nombre de connexions par base....45
5.6 Taille des bases....46
5.7 Nombre de verrous....47
5.8 Et un grand nombre d'autres informations....47
5.9 Outils....48
5.10 Outils - munin....49
5.11 Outils - Nagios....49
5.12 Outils - Zabbix....50
5.13 Outils - OPM....50
5.14 Outils - pg_stat_statements....51
5.15 Outils - PoWA....51
6 Conclusion....53

Supervision



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 (permettant l'envoi de messages à une personne en astreinte).

1.1 Menu

- Politique de supervision
- Supervision occasionnelle
- Supervision automatique

1.2 Licence Creative Commons CC-BY-NC-SA

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

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

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

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

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

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

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

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

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

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

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

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

2 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, son objectif et du coup 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.

2.1 Objectifs de la supervision

- Améliorer 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 le problème de performances. 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.

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).

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

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 soucis au niveau du système disque : soit PostgreSQL écrit trop à cause d'une mauvaise configuration des journaux de transactions, soit les requêtes exécutées utilisent des fichiers temporaires pour trier les données, ou pour une toute autre raison.

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.

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

2.4 Exemples d'indicateurs - base de données

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

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), et quelques ratios (utilisation du

cache par exemple).

3 La supervision avec PostgreSQL

- Supervision occasionnelle : pour les cas où il est possible d'intervenir immédiatement
- Supervision automatique
  - permet de remonter des informations rapidement
  - permet de conserver les informations

La supervision occasionnelle est intéressante lorsqu'un utilisateur se plaint d'un problème survenant maintenant. Cependant, cela reste assez limité.

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 envoyer des alertes quand des seuils sont dépassés.

3.1 Informations internes

- PostgreSQL propose deux canaux d'informations
  - les statistiques
  - les traces
- Mais rien pour les conserver, les historiser

PostgreSQL propose deux canaux d'informations : les statistiques d'activité et les traces applicatives.

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.

3.2 Outils externes

- Nécessaire pour conserver les informations
- … et exécuter automatiquement des actions dessus
  - Génération de graphiques (Munin, Zabbix, OPM)
  - 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.

3.3 check_postgres

- Script de monitoring PostgreSQL pour Nagios ou MRTG
  - nombreuses sondes spécifiques
  - nombreuses données de performance remontées
- http://bucardo.org/wiki/Check_postgres

Le script de monitoring `check_postgres` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios.

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. Voici les sondes `check_postgres` à mettre en place sur ces différents aspects.

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.

Vacuum :

- `bloat` : vérifie le volume de données « mortes » et la fragmentation des tables et des 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` : vérifie le nombre de verrous
- `wal_files` : Compte le nombre de segments du journal de transaction présents dans le répertoire pg_xlog.
- `query_time` : identifie les requêtes en cours d'exécution depuis trop longtemps.

Configuration :

- `settings_checksum` : indique si la configuration a été modifiée depuis la dernière vérification.

Réplication :

- `archive_ready` : compte le nombre de segments du journal de transaction en attente d'archivage.
- `hot_standby_delay` : calcule le délai de réplication entre un serveur maître et un esclave.

3.4 check_pgactivity

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - reprend la plupart des sondes de `check_postgres`, en corrigeant certaines
  - davantage de données de performance remontées
- Peut être utilisé en remplacement de `check_postgres`
- Développé par Dalibo pour les besoins de l'outil OPM
  - mais peut être utilisé indépendamment
- https://github.com/OPMDG/check_pgactivity

Tout comme le script `check_postgres`, le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios.

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 l'outil OPM (*Open PostgreSQL Monitoring*), apportant

notamment un nombre plus important de métriques de performances. Il utilise notamment un fichier local qui permet de retourner directement des ratios par rapport à la valeur précédente plutôt que des valeurs fixes. Le script peut être utilisé de façon autonome, en remplacement de `check_postgres`, sans nécessité d'installer toute l'infrastructure OPM.

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. Voici les sondes `check_pgactivity` à mettre en place sur ces différents aspects.

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*).

Vacuum :

- `bloat` : vérifie le volume de données « mortes » et la fragmentation des tables - requête entièrement réécrite par rapport à celle utilisée par `check_postgres`.
- `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_xlog.
- `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.
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL.

Configuration :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut.

Réplication :

- `ready_archives` : compte le nombre de segments du journal de transaction en attente d'archivage.
- `hot_standby_delta` : calcule le délai de réplication entre un serveur maître et un esclave.
- `is_master` : vérifie que l'instance est bien démarrée en lecture/écriture.
- `is_hot_standby` : vérifie que l'instance est en *recovery* et accepte les requêtes en lecture.

4 Traces

- Configuration
- Récupération
  - des problèmes importants
  - des requêtes lentes/fréquentes
- Outils externes de classement

La première information que fournit PostgreSQL quant à l'activité sur le serveur est 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 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 quelques informations intéressantes à récupérer. Enfin, nous verrons quelques outils permettant de traiter automatiquement les fichiers de trace.

4.1 Configuration

- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
- Tracer certains comportements

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas en même temps trop nombreuses pour ne pas être submergé par les informations et trop peu pour ne pas savoir ce qu'il se passe. Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

4.2 Configuration : où tracer

- `log_destination`
- `logging_collector`
  - `log_directory`, `log_filename`, `log_file_mode`
  - `log_rotation_age`, `log_rotation_size`, `log_truncate_on_rotation`
- `syslog_facility`, `syslog_ident`

PostgreSQL peut envoyer les traces sur plusieurs destinations :

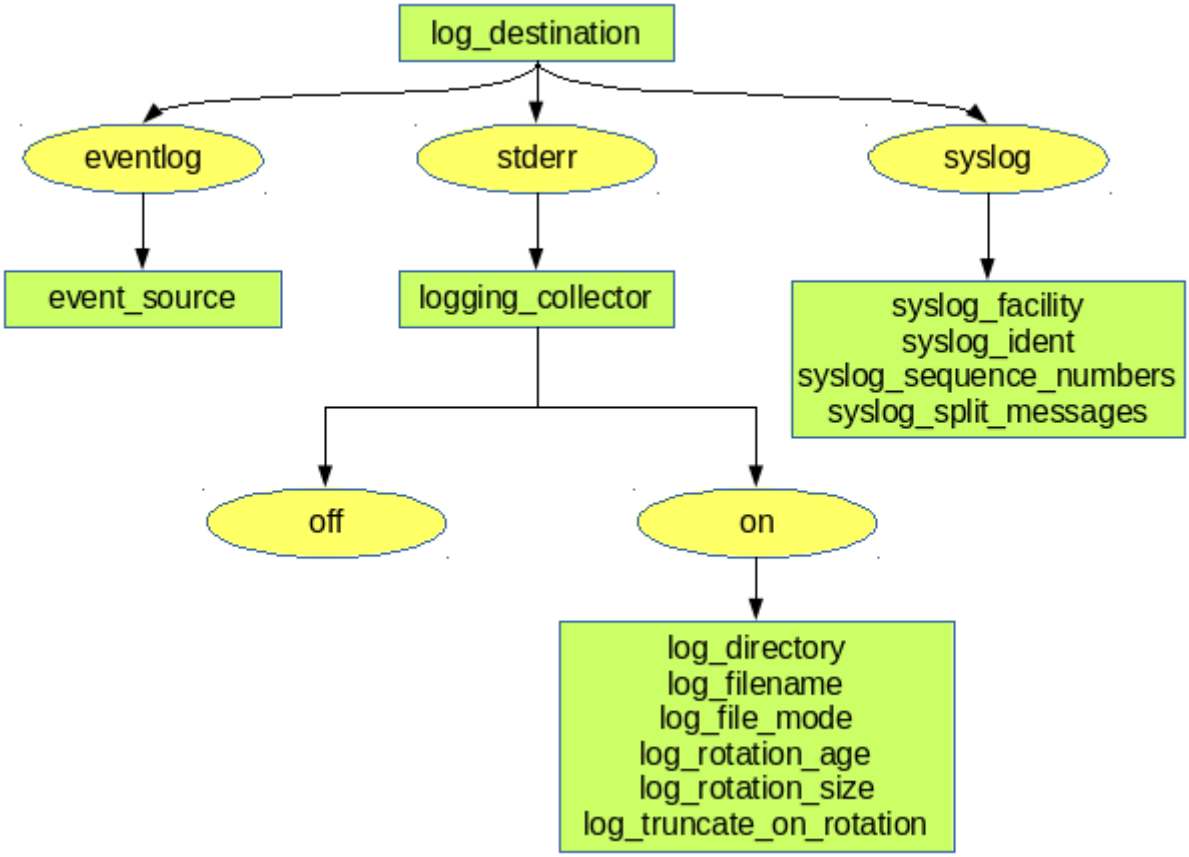
- `syslog`, fonctionne uniquement sur un serveur Unix, et est intéressant pour rassembler la configuration des traces ;
- `eventlog`, disponible unique sur Windows ;
- `stderr` et `csvlog`, disponible sur toutes les plateformes, correspond à la sortie des erreurs. La différence entre les deux réside dans le format des traces (texte simple ou CSV).

Pour `eventlog`, un dernier paramètre est disponible depuis la version 9.2. Ce paramètre, appelé `event_source`, correspond au nom du programme indiqué pour identifier les messages de PostgreSQL dans les traces. Par défaut, la valeur est PostgreSQL.

Si les traces sont envoyées à `syslog`, il reste à configurer 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. Les autres paramètres de cette page n'ont pas d'intérêt avec la destination `syslog`.

Les autres paramètres ne concernent que la destination `stderr` et `csvlog`. Il est possible d'indiquer l'emplacement des journaux applicatifs (paramètre `log_directory`), de spécifier le motif de leur nom (avec `log_filename`), ainsi que les droits des fichiers (avec `log_file_mode`). Une rotation est configurable suivant la taille (`log_rotation_size`) et la durée de vie (`log_rotation_age`).

Voici un graphe récapitulant les différentes possibilités :



4.3 Configuration : niveau des traces

- `log_min_messages`
- `log_min_error_statement`
- `log_error_verbosity`

`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 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.

4.4 Configuration : tracer les requêtes

- `log_min_duration_statement`
- `log_statement`
- `log_duration`

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

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 est préférable dans ce cas de configurer `log_statement` à la valeur `all`. Dans le cas contraire, il est impossible de dire la requête qui a pris ce temps d'exécution.

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

```
log_statements='all'
log_duration = on
```

Cela génère deux entrées dans les traces, de cette façon :

```
2012-08-10 17:06:18 CEST LOG:  statement: select * from pg_stat_activity ;
2012-08-10 17:06:18 CEST LOG:  duration: 103.767 ms
```

Parfois, on ne souhaite garder une trace que des requêtes très lentes, par exemple celles qui prennent plus de deux secondes à s'exécuter. Le paramètre `log_min_duration_statement` est là pour ça. Sa valeur correspond au nombre de millisecondes maximum avant qu'une requête ne soit tracée. Par exemple, avec cette configuration :

```
log_min_duration_statement = 2000
```

on pourrait obtenir cette trace :

```
2012-08-10 17:11:21 CEST LOG:  duration: 2906.270 ms  statement: insert into
t1 select i, i from generate_series(1, 200000) as i;
```

Remarquez que, cette fois-ci, la requête et la durée d'exécution sont indiquées dans le même message.

4.5 Configuration : tracer certains comportements

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits`
- `log_temp_files`

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements.

Quand on souhaite avoir une trace de qui se connecte, il est intéressant de pouvoir tracer les connexions et, parfois aussi, les déconnexions. En activant le paramètre `log_connections`, nous obtenons les traces suivantes pour une connexion réussie :

```
2012-08-10 17:14:22 CEST LOG:  connection received: host=[local]
2012-08-10 17:14:22 CEST LOG:  connection authorized: user=u1 database=b1
```

Le nom de l'utilisateur et la base de données sont tracés.

`log_disconnections` fait l'inverse :

```
2012-08-10 17:15:27 CEST LOG:  disconnection: session time: 0:01:04.634
user=u1 database=b1 host=[local]
```

Il ajoute une information importante : la durée de la session. Il est possible de récupérer cette information pour connaître la durée moyenne des sessions. Cette information est importante pour savoir si un outil de pooling de connexions a un intérêt.

`log_autovacuum_min_duration` correspond à `log_min_duration_statement`,

mais pour l'autovacuum. Son but est de tracer l'activité de l'autovacuum si son exécution demande plus d'un certain temps.

`log_checkpoints` permet de tracer l'activité des CHECKPOINT. Cela ajoute un message dans les traces pour indiquer qu'un CHECKPOINT commence et une autre quand il termine. Cette deuxième trace est l'occasion d'ajouter des statistiques sur le travail du CHECKPOINT :

```
2012-08-10 17:21:23 CEST LOG:  checkpoint starting: xlog
2012-08-10 17:21:25 CEST LOG:  checkpoint complete: wrote 2161 buffers
(70.3%); 0 transaction log file(s) added, 0 removed, 3 recycled; write=0.873
s, sync=0.701 s, total=1.649 s; sync files=3, longest=0.304 s, average=0.233
s
```

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 paramètre `log_lock_waits` permet de tracer une attente trop importante de verrous. En fait, quand un verrou est en attente, un chronomètre est déclenché. Lorsque l'attente dépasse la durée indiquée par le paramètre `deadlock_timeout`, un message est enregistré, comme dans cet exemple :

```
2012-08-10 17:38:40 CEST LOG:  process 15976 still waiting for
AccessExclusiveLock on relation 26160 of database 16384 after 1000.123 ms
2012-08-10 17:38:40 CEST STATEMENT:  drop table t1;
```

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur le serveur, ce qui peut diminuer fortement les performances.

Le paramètre `log_temp_files` permet de tracer toute création de fichiers temporaires, comme ici :

```
2012-08-10 17:41:11 CEST LOG:  temporary file: path
"base/pgsql_tmp/pgsql_tmp15617.1", size 59645952
```

Tout fichier temporaire demande des écritures disques. Ces écritures peuvent poser problème pour les performances globales du système. Il est donc important de savoir si des fichiers temporaires sont créés ainsi que leur taille.

4.6 Configuration : divers

- `log_line_prefix`
- `lc_messages`
- `log_timezone`

Lorsque la destination des traces est `syslog` ou `eventlog`, elles se voient automatiquement ajouter quelques informations dont un horodatage essentiel. Lorsque la destination est `stderr`, ce n'est pas le cas. Par défaut, l'utilisateur se retrouve avec des traces sans horodatage, autrement dit des traces inutilisables. PostgreSQL propose donc le paramètre `log_line_prefix` qui permet d'ajouter un préfixe à une trace. Ce préfixe peut contenir un grand nombre d'informations, comme un horodatage, le PID du processus serveur, le nom de l'application cliente, le nom de l'utilisateur, le nom de la base, etc.

Par défaut, les traces sont enregistrées dans la locale par défaut du serveur. Avoir des traces en français peut présenter certains intérêts pour les débutants mais cela présente plusieurs gros inconvénients. Chercher sur un moteur de recherche avec des traces en français donnera beaucoup moins de résultats qu'avec des traces en anglais. De même, les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il est vraiment préférable d'avoir les traces en anglais. Cela peut se faire ainsi :

```
lc_messages = 'C'
```

Quant à `log_timezone`, il permet de choisir le fuseau horaire pour l'horodatage des traces.

4.7 Informations intéressantes à récupérer

- Durée d'exécution
- Messages PANIC
- Rechargement de la configuration
- Fichiers temporaires

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 de niveau PANIC. Ces messages indiquent un état anormal du serveur qui s'est

soldé par un arrêt brutal. Ce genre de problème doit être surveillé fréquemment.

4.8 Durée d'exécution des requêtes

- `log_statement et log_duration`
- `log_min_duration_statement`
- Exemple:

```
LOG:  duration: 112.615 ms  statement: SELECT  * FROM t1 WHERE
c1=4;
```

- Outils: pgbadger, pgfouine, pgsi, etc

PostgreSQL peut tracer les requêtes exécutées ainsi que leur durée d'exécution. Il existe deux moyens pour cela :

- le couple `log_statement` et `log_duration` ;
- ou `log_min_duration_statement`.

Le paramètre `log_statement` permet de tracer les requêtes, suivant leur type. Par exemple, il est possible de tracer uniquement les requêtes de modification de schéma (requêtes `DDL`).

Le paramètre `log_duration` permet de tracer la durée d'exécution des requêtes. Il s'agit d'un booléen : soit la trace est activée, soit elle ne l'est pas.

Les deux paramètres ne sont pas liés. Par contre, il est possible de tracer toutes les requêtes avec leur durée avec cette configuration :

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

Une telle configuration donnera deux lignes dans les traces :

```
2012-09-14 17:32:39 CEST LOG:  statement: insert into t1 values
(2000000,'test');
2012-09-14 17:32:39 CEST LOG:  duration: 167.138 ms
```

Pour la recherche d'optimisation de requêtes, il est préférable de passer par un autre paramètre. Ce dernier, appelé `log_min_duration_statement`, trace toute requête dont la durée d'exécution dépasse la valeur du paramètre (l'unité est la milliseconde). Il trace aussi la durée d'exécution des requêtes tracées. Par exemple, avec une valeur de 500, toute requête dont la durée d'exécution dépasse 500 ms sera tracée. À 0, toutes les requêtes se voient tracées. Pour

désactiver la trace, il suffit de mettre la valeur -1 (qui est la valeur par défaut).

Suivant la charge que le système va subir à cause des traces, il est possible de configurer finement la durée à partir de laquelle une requête est tracée. Cependant, il faut bien comprendre que plus la durée est importante, plus la vision des performances est partielle. Il est parfois plus intéressant de mettre 0 ou une très petite valeur sur une petite durée, qu'une grosse valeur sur une grosse durée. 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 les plus longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

La trace fournie par `log_min_duration_statement` ressemble à ceci :

```
2012-09-14 17:34:03 CEST LOG:  duration: 136.811 ms  statement: insert into
t1 values (2000000,'test');
```

4.9 Messages PANIC

- Exemple:

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

- Envoi immédiat d'une alerte
- Outils: 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 indiqué ci-dessous :

```
PANIC:  could not write to file "pg_xlog/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.

4.10 Rechargement de la configuration

- Exemple :

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

- Envoi d'une alerte pour s'assurer que cette configuration est voulue
- Outils: tail_n_mail

Il est intéressant de savoir quand la configuration du serveur change, 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. Une telle alerte vous permet de vérifier de la bonne configuration du serveur.

4.11 Fichiers temporaires

- Exemple :

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

- Envoi d'une alerte sur un problème potentiel de performances

L'exécution d'une requête peut nécessiter la création de fichiers temporaires. 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. Toute écriture de ce type de fichiers impacte défavorablement l'exécution des requêtes. Être averti lors de la création de ce type de fichiers peut être intéressant. Cela étant dit, il est préférable de faire analyser après coup un fichier de traces pour savoir si un grand nombre de fichiers temporaires a été créé. Il est possible ainsi, avec un peu d'expérience, de voir que le nombre de fichiers augmente dans le mois, ce qui va demander une action de la part de l'administrateur de bases de données : vérifier les requêtes exécutées, vérifier la configuration, etc.

4.12 Outils

- Beaucoup d'outils existent
- Deux types
  - en temps réel
  - rétro-analyse
- Nous verrons les plus connus/intéressants
  - 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).

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, et des outils spécifiques pour PostgreSQL comme tail_n_mail et logsaw.

L'analyse après coup permet une analyse plus fine, se terminant généralement par un rapport, fréquemment en HTML, parfois avec des graphes. Cette analyse plus fine nécessite des outils spécialisés. Là-aussi, il en existe plusieurs :

- pgBagder ;
- pgFouine ;
- pgsi ;
- pqa ;
- epqa ;
- etc.

Dans cette partie, nous allons voir les trois outils les plus intéressants.

4.13 Utiliser pgBadger

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations sur les requêtes et leur durée d'exécution
- Génération d'un rapport HTML très détaillé

pgBadger est un script Perl écrit par Gilles Darold. Il 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, il peut contenir des graphes zoomables.

C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL.

Le site web de pgBadger se trouve sur http://dalibo.github.com/pgbadger/

4.14 Configurer PostgreSQL pour pgBadger

- log_destination = syslog ou stderr ou csvlog
- log_min_duration_statement = 0
- lc_messages = 'C'

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. Par contre, il faudra le préciser à pgBadger si la configuration est différente de celle qui suit :

```
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d '
```

La langue des traces doit être l'anglais. De toute façon, il s'agit de la meilleure configuration des traces. En effet, il est difficile de trouver de l'information sur des traces en français, alors que ce n'est pas le cas avec des traces en anglais. Pour cela, il suffit de configurer `lc_messages` à la valeur `C`.

Enfin, il faut demander à PostgreSQL de tracer les requêtes. Il est préférable de passer par `log_min_duration_statement` plutôt que `log_statement` et `log_duration` pour que pgBadger puisse faire l'association entre les requêtes

et leur durée :

```
log_min_duration_statement = 0
log_statement = off
log_duration = off
```

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

- `log_checkpoints` pour des statistiques sur les `CHECKPOINT` ;
- `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.

4.15 Options de pgBadger

- `--outfile`
- `--begin`, `--end`
- `--dbname`, `--dbuser`, `--dbclient`, `--appname`

Il existe énormément d'options :

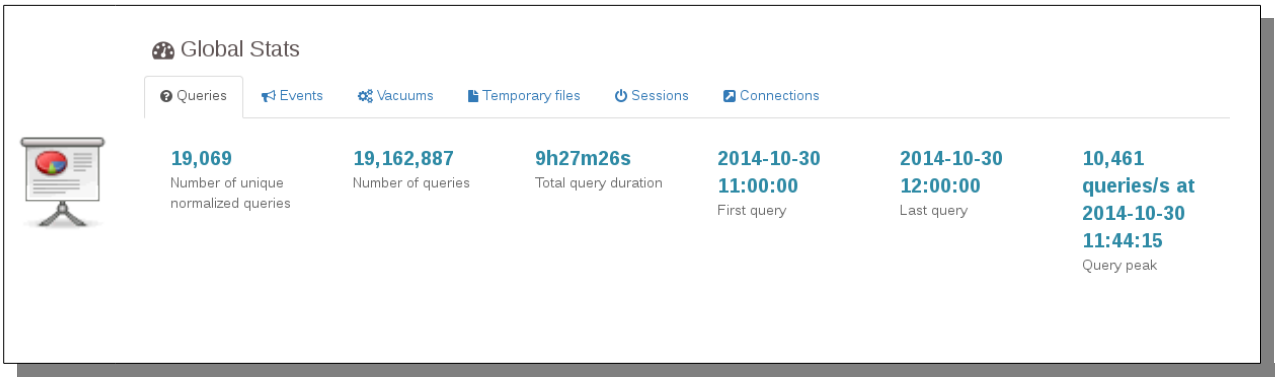
- fichier en entrée (traces de PostgreSQL)
  - `--format` (syslog, stderr, ou csv)
  - `--zcat`, pour préciser l'emplacement du programme de décompression si les fichiers en entrée sont compressés
  - `--ident`, nom d'identification du programme utilisé dans les traces syslog
  - `--prefix`, pour spécifier le préfixe des traces PostgreSQL s'il n'est pas conforme à la valeur recommandée dans la documentation de pgBadger (certaines informations restent indispensables)
  - `--csv-separator` (spécifique csv), pour préciser le séparateur de champs du format CSV
- filtres
  - `--begin` : à partir de quand commencer l'analyse ;
  - `--end` : à partir de quand terminer l'analyse ;
  - `--dbclient` : ne prendre en compte que les traces qui concernent cet hôte client
  - `--dbname` : ne prendre en compte que les traces qui concernent cette base

de données
  - `--dbuser` : ne prendre en compte que les traces qui concernent cet utilisateur
  - `--appname` : ne prendre en compte que les traces qui concernent cette application
  - `--exclude-query`, pour préciser l'expression rationnelle correspondant aux requêtes à exclure du rapport
  - `--exclude-file`, pour indiquer le fichier contenant une liste d'expressions rationnelles correspondant aux requêtes à exclure du rapport
  - `--select-only`, pour ne prendre en compte que les requêtes de type SELECT
  - `--exclude-user`, pour exclure les lignes concernant l'utilisateur spécifié
  - `--exclude-time`, pour spécifier une expression rationnelle correspondant à une date à exclure du rapport (par example : “2013-04-12 .*”)
  - `--exclude-appname`, pour exclure les lignes correspondant au nom d'application spécifié
  - `--include-query`, pour spécifier une expression rationnelle correspondant aux seules requêtes à garder dans le rapport
  - `--include-file`, pour indiquer le fichier contenant une liste d'expressions rationnelles correspondant aux seules requêtes à garder dans le rapport
  - `--watch-mode`, pour ne générer qu'un rapport texte sur les erreurs, façon logwatch
- fichier en sortie (rapport pgBadger)
  - `--outfile`, pour préciser le nom du fichier en sortie
  - `--extension`, pour préciser le format en sortie (texte ou HTML)
  - `--nograph`, pour ne pas générer les graphes
  - `--maxlength`, pour préciser la taille maximum d'une requête dans le fichier en sortie
  - `--nohighlight`, pour ne pas activer la coloration syntaxique du code SQL
  - `--no-prettify`, pour ne pas formater les requêtes
  - `--top`, pour préciser le nombre de requêtes par rapport
  - `--sample`, pour préciser le nombre de requêtes exemples à afficher
  - `--title`, pour modifier le titre de la page HTML
  - `--pie-limit`, pour modifier la valeur basse à partir de laquelle l'élément correspondant sera comptabilisé dans un élément “Others”
  - `--exclude-query`, pour préciser l'expression rationnelle correspondant aux requêtes à exclure du rapport

  - `--exclude-file`, pour indiquer le fichier contenant une liste d'expressions rationnelles correspondant aux requêtes à exclure du rapport
  - `--disable-error`, `--disable-hourly`, `--disable-type`, `--disable-query`, `--disable-session`, `--disable-connection`, `--disable-lock`, `--disable-temporary`, `--disable-checkpoint`, `--disable-autovacuum`, pour désactiver certains rapports
  - `--average`, pour préciser l'intervalle (en minutes) utilisé pour calculer les moyennes servant à construire les graphes
  - `--nocomment`, pour supprimer automatiquement les chaînes de commentaire du texte des requêtes
  - `--outdir`, pour préciser le répertoire dans lequel le rapport doit être créé
  - `--charset`, pour précisé le jeu de caractères utilisé par le rapport HTML
- exécution
  - `--incremental`, active le mode incrémentiel, qui va gérer des rapports pour chaque journée contenue dans les logs
  - `--last-parsed`, pour spécifier un fichier qui sera utilisé pour mémoriser la position d'arrêt (ligne, timestamp) de la dernière analyse des traces, et reprendre ultérieurement de cette même position
  - `--jobs`, pour préciser le nombre de processus à utiliser pour paralléliser la génération du rapport
- divers
  - `--help`, pour accéder à l'aide sur les options en ligne de commande
  - `--quiet`, pour ne rien afficher pendant l'analyse
  - `--verbose`, pour afficher des traces de l'exécution de pgBagder
  - `--version`, pour afficher la version de pgBadger

Il est possible que d'autres options soient déjà disponibles, ce programme étant fréquemment amélioré.

4.16 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, les requêtes :

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

et

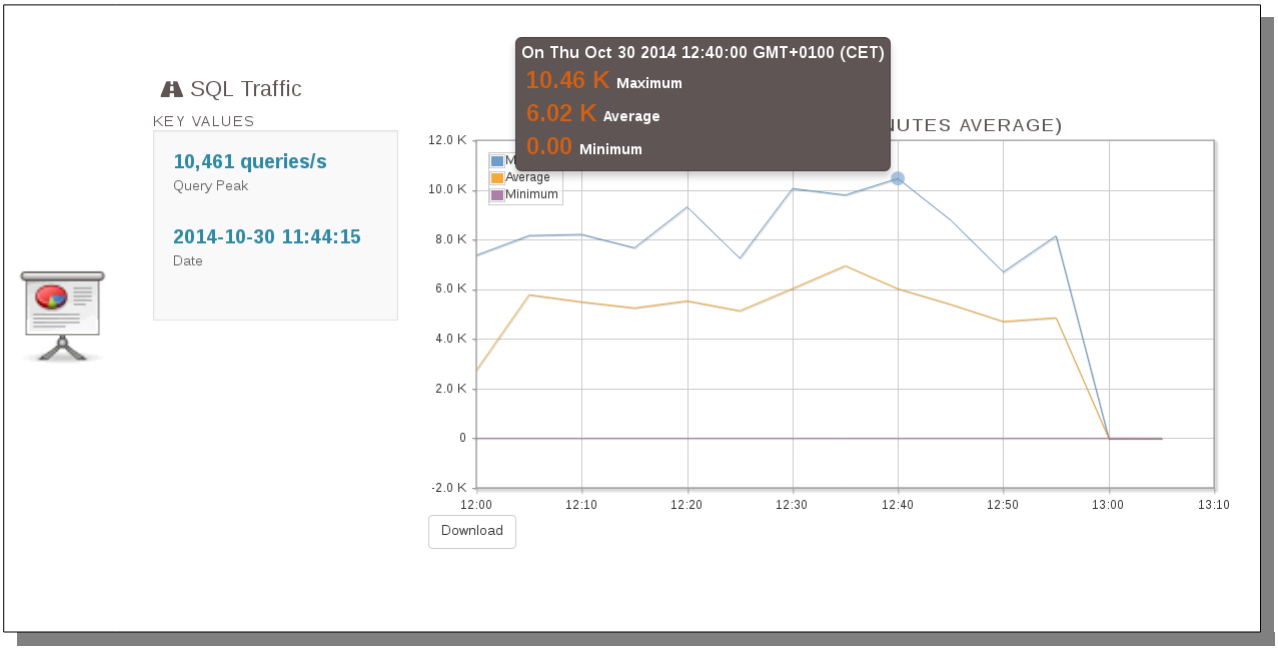
```
SELECT * FROM utilisateurs WHERE id = 2;
```

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 fiche 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 montré ci-dessus, pgBadger a comptabilisé environ 5,5 millions de requêtes, mais seulement 3941 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 4000.

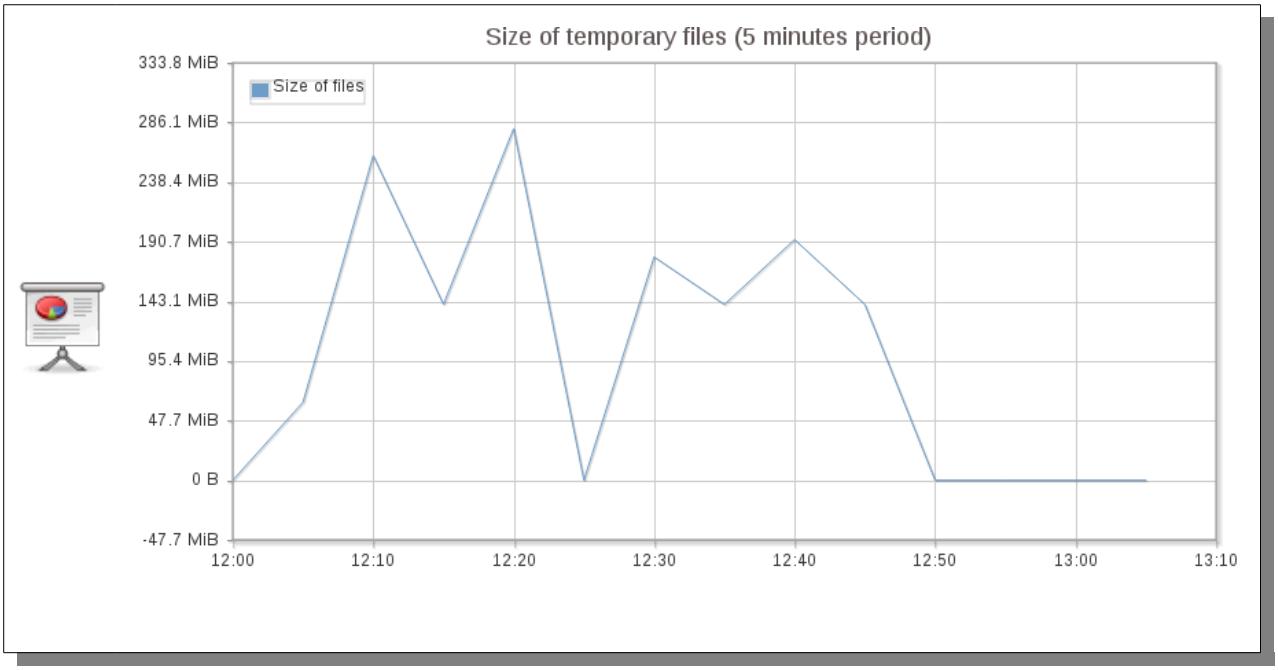
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 1 jour et 7 heures d'exécution de requêtes. Cependant, les traces vont du 20 au 23 mars, soit plus de trois jours. Cela indique que le serveur est assez peu sollicité. Il est plus fréquent que la durée d'exécution sérielle des requêtes soit 3 à 4 fois plus importants que la durée des traces.

4.17 pgBadger : exemple 2



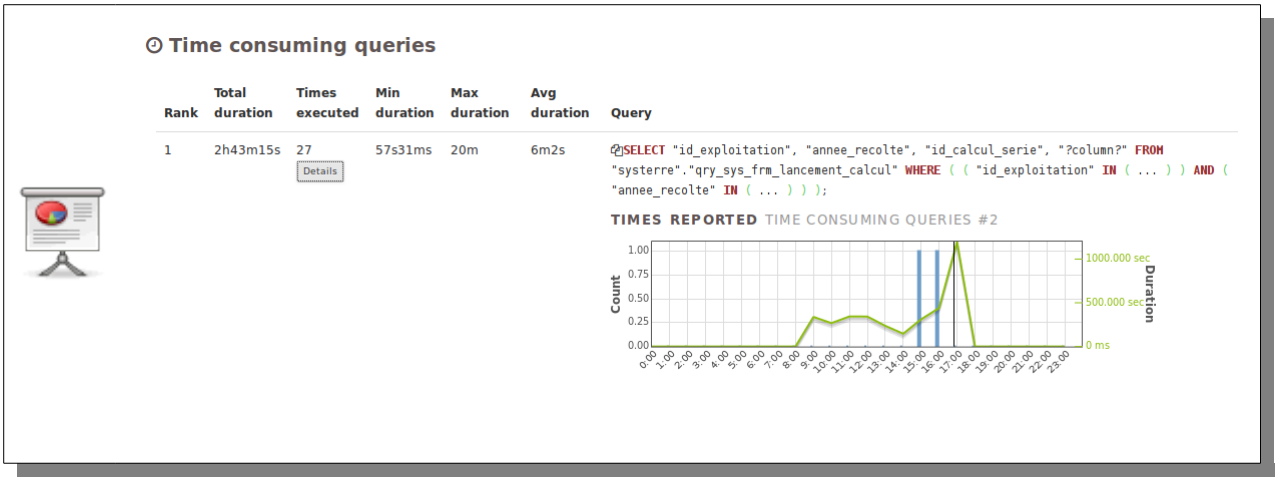
Ce graphe indique le nombre de requêtes par seconde. Le système ici est assez peu utilisé quand on considère le nombre moyen (en orange sur le graphe).

4.18 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. On remarque ainsi la création de fichiers pour un total de plus de 400 Mo par moment.

4.19 pgBadger : exemple 4



Le plus important est certainement ce tableau. Il affiche 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. On remarque d'ailleurs dans cet exemple qu'avec les trois premières requêtes, on arrive à un total de 27 heures. Le premier exemple nous indique que l'exécution sérielle des requêtes aurait pris 23 heures. En ne travaillant

que sur ces trois requêtes, nous travaillons en fait sur 87% du temps total d'exécution des requêtes comprises dans les traces. Pas besoin donc de travailler sur les 4000 requêtes normalisées.

4.20 Utiliser logwatch

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

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.

Après son installation, vous pouvez voir qu'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 comme dans l'exemple qui suit.

4.21 Configurer logwatch

- Pas de configuration disponible par défaut pour PostgreSQL
- Dépôt git Dalibo
  - https://github.com/dalibo/pgsql_logwatch/
  - à installer manuellement

La configuration et le script relatifs au traitement des journaux applicatifs de PostgreSQL par `logwatch` ne sont pas disponibles en standard dans le paquet logqwatch. Vous devrez l'installer manuellement à partir du dépôt `GIT`

disponible sur https://github.com/dalibo/pgsql_logwatch/ puis procéder aux actions suivantes :

```
sudo cp logfiles_postgresql.conf /etc/logwatch/conf/logfiles/postgresql.conf
sudo cp services_postgresql.conf /etc/logwatch/conf/services/postgresql.conf
sudo cp scripts_postgresql /etc/logwatch/scripts/services/postgresql
sudo chmod 755 /etc/logwatch/scripts/services/postgresql
```

Ces fichiers sont maintenant intégrés à la version bêta 7.4.2 de Logwatch http://sourceforge.net/projects/logwatch/

4.22 Options de logwatch

- `--logfile`, les fichiers à traiter
- `--service`, filtre sur le service unique à traiter
- `--detail`, niveau de détails du rapport
- `--print`, pour afficher le rapport sur la sortie standard
- `--save`, pour sauvegarder le rapport dans un fichier
- `--mailto`, pour envoyer le rapport par mail

logwatch dispose d'autres options, la page man est certainement la meilleure documentation à l'heure actuelle.

4.23 logwatch : exemple 1

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

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

On peut changer aussi le niveau de détail, avec `--detail` à `Low`. Seuls les messages de type `FATAL`, `PANIC` et `ERROR` seront remontés. Avec la valeur `Med`, les messages de type `WARNING` et `HINTS` sont ajoutés aux précédents.

4.24 logwatch : exemple 2

```
/usr/sbin/logwatch --detail Med --service postgresql --range
Yesterday \
                    --output mail --mailto admin@mydom.com
--format html
```

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

 6 times:
 [2011-11-01 06:38:20 +/-23 hour(s)]  password authentication failed for
user "test"

 4 times:
 [2011-10-24 17:26:24 +/-14 seconds]  Ident authentication failed for user
"test"
 [2011-11-24 23:22:02 +/-1 day(s)]  database system is shut down

 3 times:
 [2011-12-10 02:49:02 +/-23 day(s)]  database "test" does not exist

 1 times:
 [2011-12-09 17:39:06]  role "test" is not permitted to log in

 Errors:
 -------

 7 times:
 [2011-11-14 12:20:44 +/-58 minute(s)]  relation "COUNTRIES" does not exist

 6 times:
```

```
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "dept_loc_fk" of relation
"departments" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "loc_c_id_fk" of relation
"locations" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "dept_mgr_fk" of relation
"departments" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "afpr_aft_fk" of relation
"flx_properties" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "jhist_job_fk" of
relation "job_history" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_manager_fk" of
relation "employees" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_dept_fk" of relation
"employees" does not exist
 [2011-11-14 12:19:42 +/-56 minute(s)]  syntax error at or near "dbms_lob"
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_job_fk" of relation
"employees" does not exist
 [2011-11-14 12:19:42 +/-56 minute(s)]  syntax error at or near "NAME"

 5 times:
 [2011-11-14 12:21:24 +/-59 minute(s)]  syntax error at or near
"role_permission_view"

 4 times:
 [2011-12-08 20:46:37 +/-16 hour(s)]  conflicting or redundant options
 [2011-11-01 19:10:33 +/-6 minute(s)]  relation "t_test" does not exist

 2 times:
 [2011-11-14 12:30:51 +/-1 minute(s)]  schema "hr" already exists
 [2011-11-14 14:32:15 +/-4 hour(s)]  syntax error at or near "hr"

 Warnings:
 ---------

 55 times:
 [2011-11-18 12:26:39 +/-6 day(s)]  terminating connection because of crash
of another server process

 1 times:
 [2011-11-29 09:05:47]  pgstat wait timeout
 [2011-11-14 16:34:03]  nonstandard use of \' in a string literal

 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 your 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.

 2 times:
 [2011-10-10 17:38:28 +/-3 minute(s)]  No operator matches the given name
and argument type(s). You might need to add explicit type casts.
 [2011-10-25 09:18:52 +/-2 seconds]  Consider increasing the configuration
parameter "checkpoint_segments".
```

```
 1 times:
 [2011-11-15 15:20:50]  Could not choose a best candidate function. You
might need to add explicit type casts.
 [2011-11-14 16:34:03]  Use '' to write quotes in strings, or use the escape
string syntax (E'...').


 ---------------------- PostgreSQL End -------------------------


 ###################### Logwatch End #########################
```

4.25 Utiliser 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 l'astreinteur sera prévenu rapidement et pourra agir en conséquence.

4.26 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
```

La clé `MAILSUBJECT` précise le sujet du mail envoyé. La clé `EMAIL` permet de préciser à quelle adresse mail envoyer les rapports.

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.

Enfin, la clé `FILE` permet de préciser le fichier à analyser.

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 » et « 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.

4.27 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/pg_log/postgres-2010-01-01.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
```

5 Statistiques

- 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.

5.1 Statistiques - configuration 1

- Tracer l'activité
  - `track_activities`
- S'assurer que les requêtes ne sont pas tronquées
  - `track_activity_query_size`

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`. Si ce dernier est désactivé, la liste des sessions dans la vue `pg_stat_activity` est toujours présente et actualisée, mais les colonnes `xact_start`, `query_start`, `waiting` et `current_query` ne sont plus mises à jour. Une fois le paramètre activé, les colonnes sont renseignées.

```
b1=# SHOW track_activities;
 track_activities
------------------
 off
(1 row)

b1=# \x
Expanded display is on.
b1=# SELECT * FROM pg_stat_activity ;
-[ RECORD 1 ]----+-------------------------------
datid            | 16384
datname          | b1
```

```
procpid           | 6633
usesysid          | 10
usename           | guillaume
application_name  | psql
client_addr       |
client_hostname   |
client_port       | -1
backend_start     | 2012-08-09 11:59:23.089244+02
xact_start        |
query_start       |
waiting           | f
current_query     | <command string not enabled>
```

Il est à noter que la requête indiquée dans la colonne `current_query` est tronquée à 1024 caractères par défaut. Il est possible de configurer cette limite grâce au paramètre `track_activity_query_size`.

Voici un exemple montrant la requête tronquée après 100 caractères (valeur du paramètre `track_activity_query_size`) :

```
b1=# \x
Expanded display is on.
b1=# SELECT current_setting('track_activities'),
current_setting('track_activity_query_size');
-[ RECORD 1 ]---+----
current_setting | on
current_setting | 100

b1=# SELECT '0123456789', current_query FROM pg_stat_activity;
-[ RECORD 1 ]-+-------------------------------------------------------------
?column?      | 0123456789
current_query | select '0123456789', current_query from pg_stat_activity;

b1=# SELECT
'0123456789012345678901234567890123456789012345678901234567890123456789012345
678901234567890123456789', current_query FROM pg_stat_activity;
-[ RECORD 1 ]-
+-------------------------------------------------------------------------
--------------------------
?column?      |
0123456789012345678901234567890123456789012345678901234567890123456789012345
678901234567890123456789
current_query | select
'0123456789012345678901234567890123456789012345678901234567890123456789012345
678901234567890
```

5.2 Statistiques - configuration 2

- `track_counts`
- `track_io_timing`
- `track_functions`

Le collecteur de statistiques est capable de récupérer des informations sur des nombres de lignes (lues, insérées, mises à jour, supprimées, vivantes, mortes, etc) et sur des nombres de blocs (lus dans le cache de PostgreSQL et en dehors de ce cache). Pour avoir ce type de statistiques, il faut activer le paramètre `track_counts`.

En version 9.2, PostgreSQL peut chronométrer les opérations sur les disques : lectures, écritures, synchronisations. Pour cela, il faut activer le paramètre `track_io_timing`.

Depuis la version 9.0, PostgreSQL sait aussi récupérer des statistiques sur les procédures stockées. Il faut modifier la valeur du paramètre `track_functions` qui, contrairement aux autres paramètres, se trouve être un enum à 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.

5.3 Statistiques - configuration 3

- `stats_temp_directory`
  - fichier écrit toutes les 500ms avant la 8.4
  - seulement quand nécessaire à partir de la 8.4

Le collecteur de statistiques fonctionne ainsi :

- il est lancé au démarrage de PostgreSQL (il est d'ailleurs impossible de le désactiver complètement) ;
- il copie le fichier `global/pgstat.stat` dans le répertoire ciblé par le paramètre `stats_temp_directory` ;
- il modifie ce fichier dès que les autres processus lui fournissent des statistiques ;
- à l'arrêt de PostgreSQL, il copie le fichier `pgstat.stat` du répertoire ciblé par le paramètre `stats_temp_directory` dans le répertoire `global`.

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.

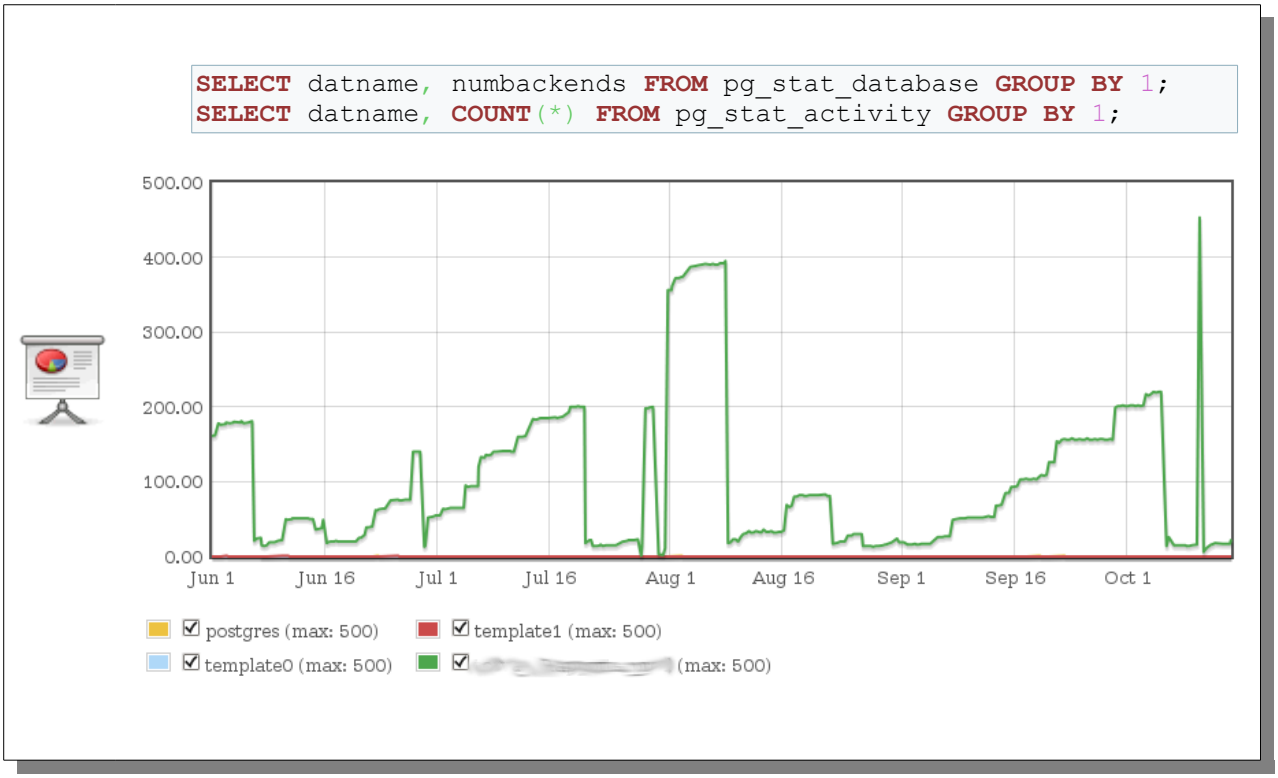
5.4 Informations intéressantes à récupérer

- sur l'activité
- sur l'instance
- sur les bases
- sur les tables
- sur les index
- sur 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 a peu d'intérêt car seulement certaines sont vraiment intéressantes à superviser. Nous allons voir ici quelques exemples.

5.5 Nombre de connexions par base



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, numbackends FROM pg_stat_database GROUP BY 1;
```

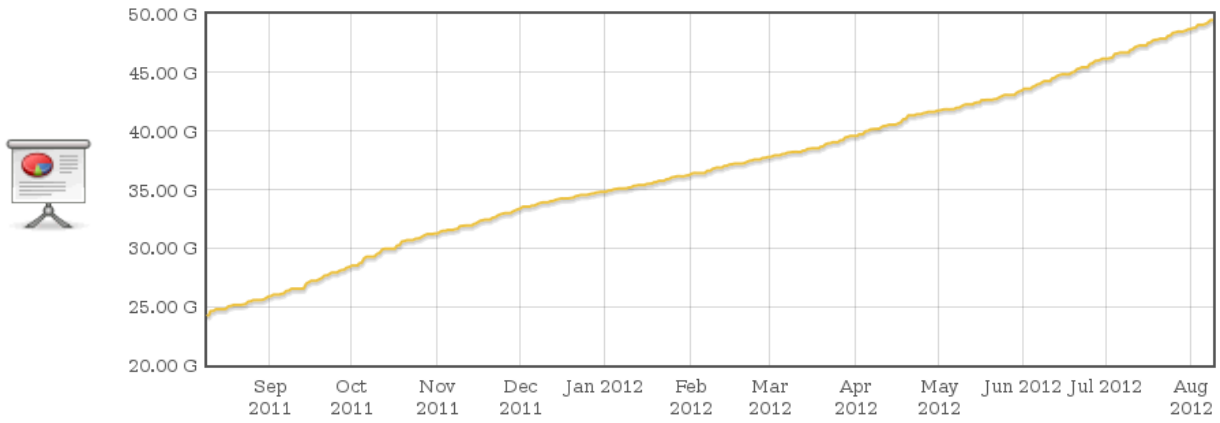
- application cliente :

```
SELECT application_name, numbackends FROM pg_stat_database GROUP BY 1;
```

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.

5.6 Taille des bases

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

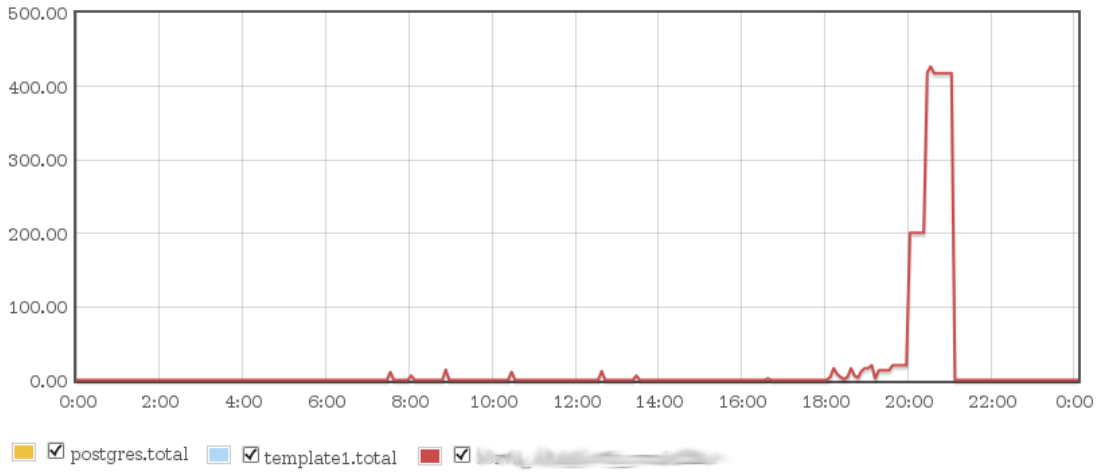


La volumétrie des bases est une autre information fréquemment demandée. L'exécution de cette requête toutes les cinq minutes permettra de suivre l'évolution de la taille des bases.

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

5.7 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.

5.8 Et un grand nombre d'autres informations

- Ratio de lecture du cache (souvent appelé `hitratio`)
- 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.

5.9 Outils

- Beaucoup d'outils existent pour exploiter les statistiques
- Les plus connus/intéressants sont :
  - Munin
  - Nagios
  - Zabbix
  - OPM
  - 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, généralement contenues dans un script de supervision. Le plus utilisé actuellement est `check_postgres` développé par End Point Corporation, auteurs entre autres de Bucardo. Le script `check_pgactivity`, développé par Dalibo, vise à le remplacer.

OPM (*Open PostgreSQL Monitoring*) est un outil lancé par Dalibo, qui repose sur la mise en place d'un système de collecte, stocke la plupart des statistiques de PostgreSQL et du système d'exploitation sous-jacent dans une base de données PostgreSQL, et permet de les exploiter sous forme de graphiques interactifs et personnalisables .

pg_stat_statements est un module contrib de PostgreSQL, il s'agit donc d'une fonctionnalité qui n'est pas installée par défaut. Il ajoute la collecte de statistiques sur toutes les requêtes exécutées.

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, 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.

5.10 Outils - munin

- Scripts Perl
- Sondes PostgreSQL incluses depuis la 1.4
- Récupère les statistiques toutes les cinq minutes
- 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.

Dalibo avait créé un projet de sondes PostgreSQL pour la version 1.2 de munin. Ces sondes ont été réécrites par les développeurs de munin et directement intégrées dans la version 1.4 officielle.

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 créés pour faciliter l'accès aux graphes.

5.11 Outils - Nagios

- Outil GPL, sur http://www.nagios.org/
- 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 est possible de le coupler à des sondes dédiées, comme la sonde Bucardo `check_postgres` ou la plus récente sonde Dalibo `check_pgactivity`, pour qu'il puisse récupérer un certain nombre d'informations statistiques de PostgreSQL.

5.12 Outils - Zabbix

- Outil GPL, sur http://www.zabbix.com/
- 2.0
- Sonde check_postgres.pl
- Template pg-monz
  - http://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.

5.13 Outils - OPM

- *Open PostgreSQL Monitoring*
- Suite de supervision lancée par Dalibo en septembre 2014
- Projet indépendant mené par OPMDG (*OPM Development Group*)
- Licence PostgreSQL
- http://opm.io/

Open PostgreSQL Monitoring est une suite de supervision lancée par Dalibo. L'objectif est d'avoir un outil permettant de surveiller un grand nombre d'instances PostgreSQL, et à terme de les administrer, de façon similaire à ce que permettent les outils *Oracle Enterprise Manager* ou *SQL Server Management Studio*.

Le projet est publié sous licence PostgreSQL, et un comité a été créé (OPMDG, *OPM Development Group*), à la manière du PGDG, de façon à assurer l'indépendance du projet.

Le cœur de OPM est une base de données PostgreSQL stockant de nombreuses statistiques concernant les instances et le système d'exploitation des serveurs les hébergeant. Ces données sont ensuite utilisées par l'interface graphique pour les afficher sous forme de graphiques interactifs et personnalisables.

À ce jour la collecte des statistiques nécessite la configuration de Nagios avec le script `check_pgactivity`, mais d'autres systèmes de collecte pourront être

ajoutés à l'avenir.

5.14 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 un module contrib de PostgreSQL, il s'agit donc d'une fonctionnalité qui n'est pas 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 :
http://www.postgresql.org/docs/9.3/static/pgstatstatements.html

5.15 Outils - PoWA

- *PostgreSQL Workload Analyzer*
- Effectue des captures des statistiques collectées par pg_stat_statements
- Fournit une interface graphique pour observer en temps réel l'activité des requêtes
- Licence PostgreSQL
- http://dalibo.github.io/powa/

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, sous licence PostgreSQL. Tout comme pour 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. 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.

PoWA est compatible avec les versions 9.3 et supérieures de PostgreSQL.

6 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.