DALIBO

L'expertise PostgreSQL

Atelier supervision – PG Day France 2014 - Toulon

Table des matières

Supervision PostgreSQL

1 Licence des slides

- Creative Common BY-NC-SA
- Vous êtes libre
  - de partager
  - de modifier
- Sous les conditions suivantes
  - Attribution
  - Non commercial
  - Partage dans les mêmes conditions

2 Auteurs

- Julien Rouhaud
  - email: julien.rouhaud@dalibo.com
- Thomas Reiss
  - email: thomas.reiss@dalibo.com
- Consultants PostgreSQL chez Dalibo

3 Menu

- Politique de supervision
- Statistiques
- Traces
- Outils

4 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.

4.1 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.

4.2 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).

4.3 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.

4.4 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).

5 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.

5.1 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.

5.2 Outils externes

- Nécessaire pour conserver les informations
- … et exécuter automatiquement des actions dessus
  - Génération de 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.

5.3 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.

5.4 pgBadger

- Outils d'analyse des logs PostgreSQL
  - https://github.com/dalibo/pgbadger
  - génère un rapport HTML
- Différents aspects mesurés :
  - checkpoints
  - activité d'autovacuum
  - requêtes SQL
  - erreurs

5.5 pgCluu

- Outils de collectes de métriques de performances
  - https://github.com/darold/pgcluu
  - génère un rapport HTML complet
- Différents aspects mesurés :
  - informations sur le système
  - consommation des ressources CPU, RAM, I/O
  - utilisation de la base de données

6 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.

6.1 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.

6.2 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`
- `silent_mode`

PostgreSQL peut envoyer les traces sur plusieurs destinations :

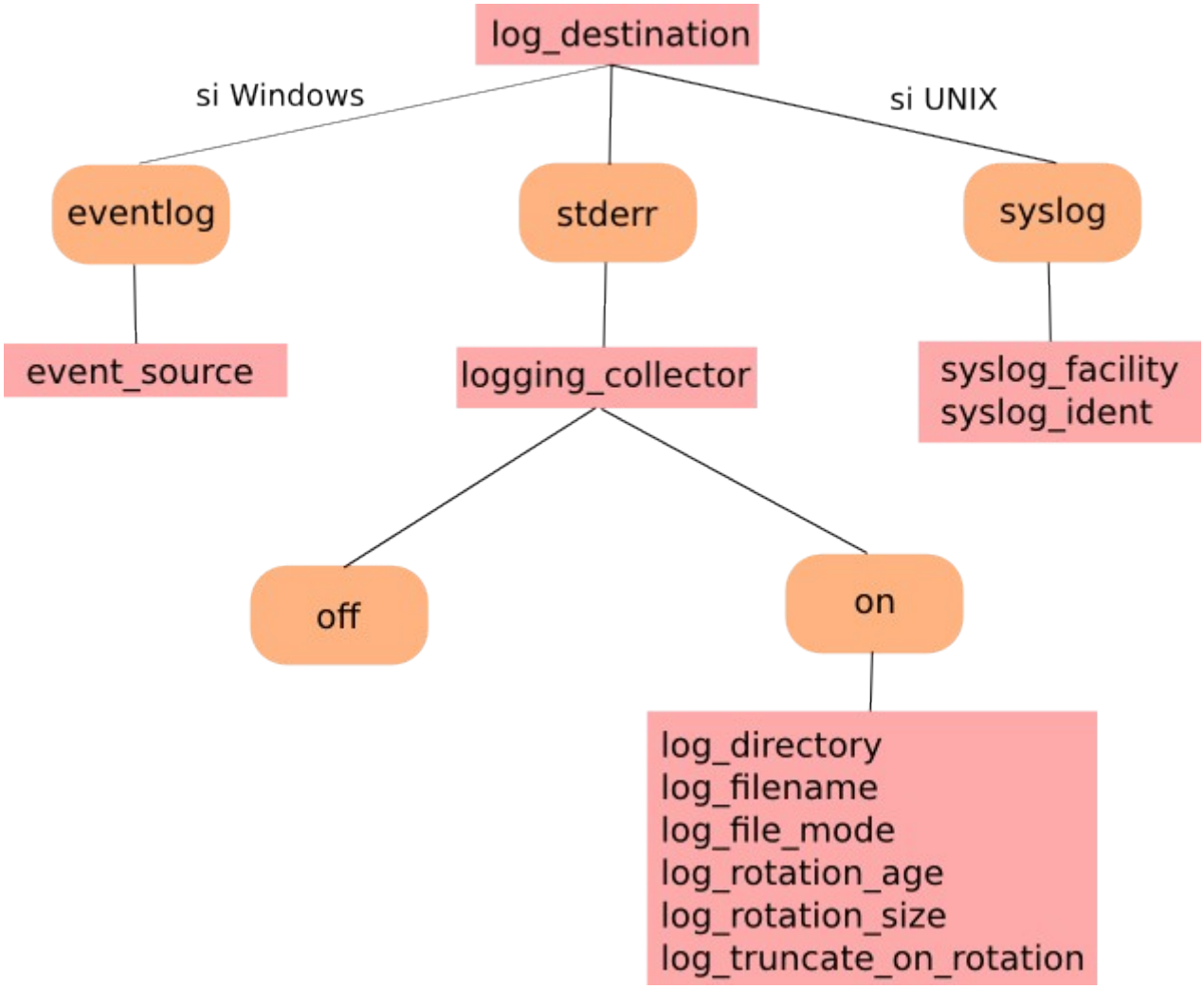
- `syslog`, fonctionne uniquement sur un serveur Unix, et est intéressant pour rassembler la configuration des traces ;
- `eventlog`, disponible unique sur Windows ;
- `stderr` et `csvlog`, disponible sur toutes les plateformes, correspond à la sortie des erreurs. La différence entre les deux réside 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 :



6.3 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.

6.4 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.

6.5 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.

6.6 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.

6.7 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.

6.8 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');
```

6.9 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.

6.10 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 L0G 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.

6.11 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.

6.12 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.

6.13 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/

6.14 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.

6.15 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

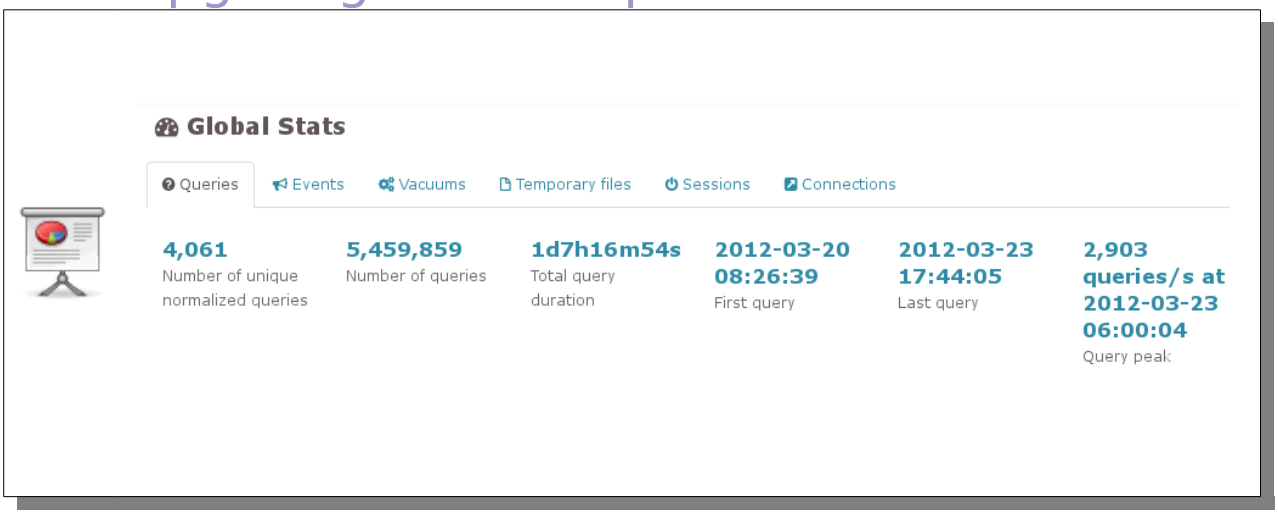
  - `--begin` : à partir de quand commencer l'analyse ;
  - `--end` : à partir de quand terminer l'analyse ;
  - `--dbclient` : ne prendre en compte que les traces qui concernent cet hôte client
  - `--dbname` : ne prendre en compte que les traces qui concernent cette base de données
  - `--dbuser` : ne 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.

6.16 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;
```

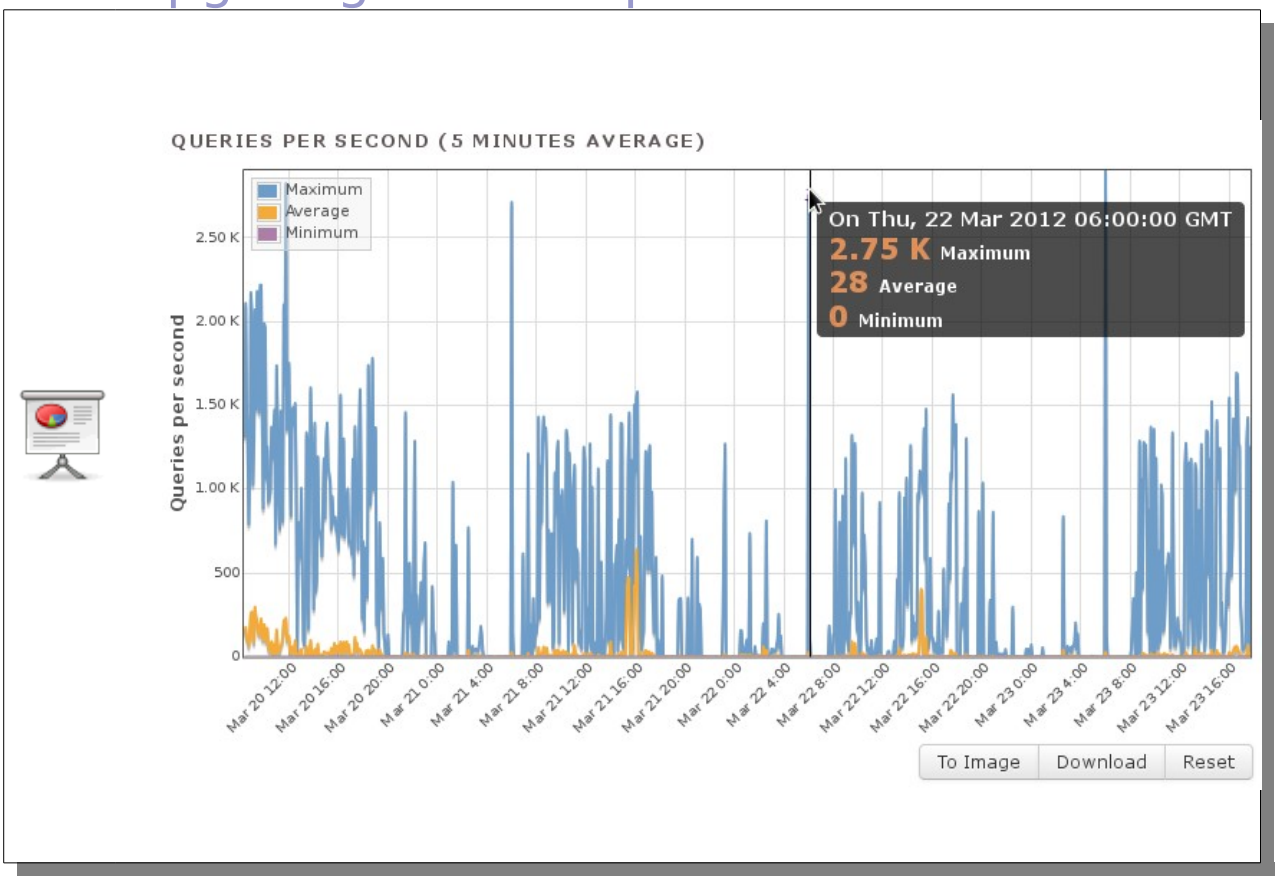
et

```
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.

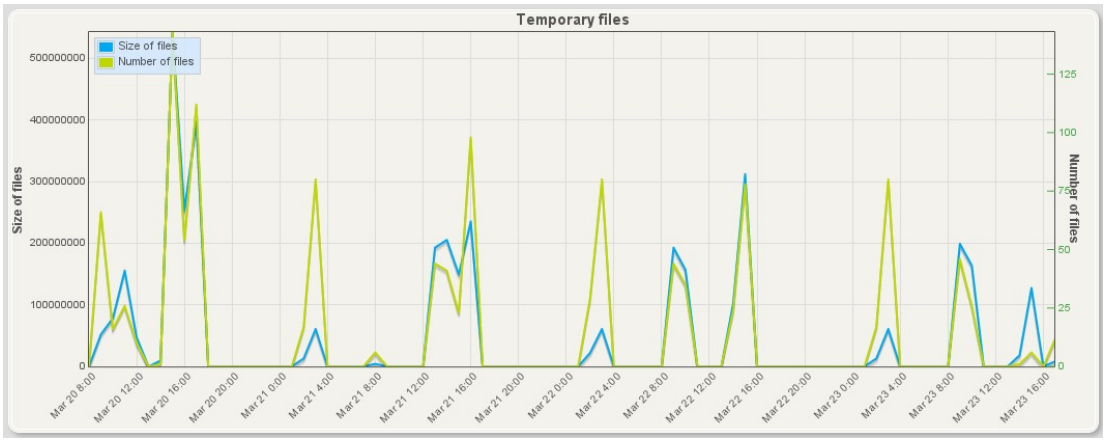
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 1 jour et 7 heures d'exécution de requêtes. Cependant, les traces vont du 20 au 23 mars, soit plus de trois jours. Cela indique que le serveur est assez peu sollicité. Il est plus fréquent que la durée d'exécution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

6.17 pgBadger : exemple 2



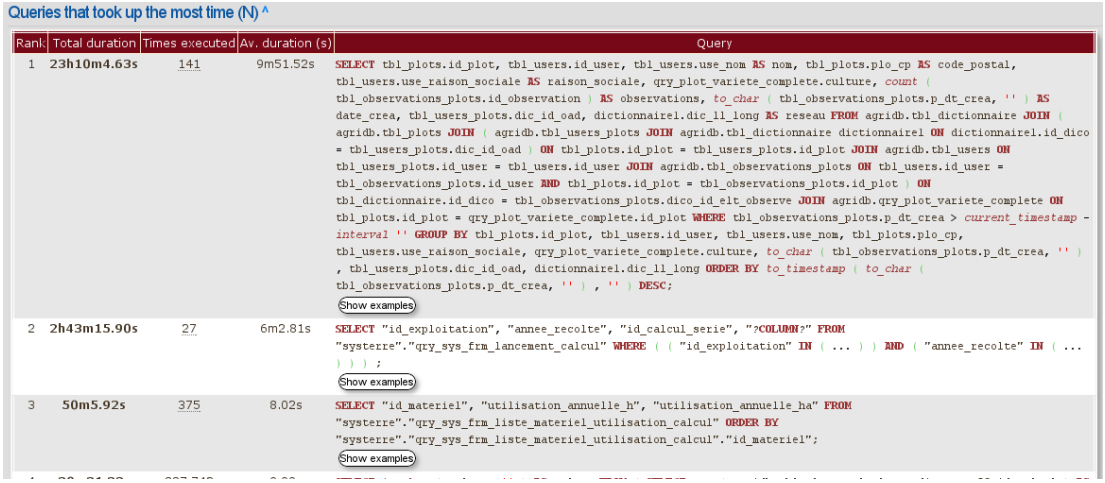
Ce graphe indique le nombre de requêtes par seconde. Le système ici est assez peu utilisé quand on considère le nombre moyen (en vert sur le graphe).

6.18 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.

6.19 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 indiquait que l'exécution sérielle des requêtes aurait pris 31 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.

7 Statistiques

- Configuration
- Liste des vues statistiques
- Informations intéressantes à récupérer
- Outils disponibles

La supervision occasionnelle est essentielle à maîtriser mais il est aussi important de configurer correctement une supervision automatique. Les outils Nagios et Zabbix permettent cette supervision automatique, en les couplant avec une sonde comme check_postgres. Mais cette sonde ne fait pas tout. Il est important de pouvoir la modifier et créer ses propres sondes.

Pour cela, nous allons reprendre la configuration sur les statistiques d'activité de PostgreSQL. Nous allons ensuite regarder la liste des vues disponibles et expliquer les différents champs, et faire le point sur les informations essentielles pour de la supervision. Nous verrons aussi les informations importantes à récupérer et à grapher. Enfin, nous ferons le point sur les outils déjà disponibles.

7.1 Rappel sur la configuration

- `track_activities`, `track_activity_query_size`
- `track_counts`, `track_io_timing`, `track_functions`
- `update_process_title`
- `stats_temp_directory`

PostgreSQL dispose de quelques paramètres pour configurer au mieux le collecteur des statistiques d'activité.

PostgreSQL est capable de tracer en mémoire l'activité des différentes connexions aux bases de l'instance. Cela permet notamment de récupérer l'état de la requête en cours d'exécution, ainsi que la requête elle-même. Il est aussi capable de conserver des informations sur les connexions de réplication. Toutes ces informations se trouvent

dans deux vues : pg_stat_activity et pg_stat_replication. Certaines des informations (notamment celles sur les requêtes) ne sont récupérées que si le paramètre track_activities est renseigné. De plus, la requête n'est pas enregistrée entièrement. Elle est tronquée à une certaine position (1024 par défaut). Il est cependant possible d'augmenter la taille de la requête. Cette information est conservée dans la mémoire partagée, donc nécessite un redémarrage de PostgreSQL pour que la configuration soit prise en compte.

En activant track_counts, PostgreSQL récupère des informations de décompte : nombre de lignes, nombre de blocs, etc. Cela concerne principalement les vues pg_stat_% et pg_statio_%. Le paramètre track_io_timing permet de chronométrer les opérations disques. En l'activant, les vues statistiques pg_stat_database et pg_stat_bgwriter disposeront de plus d'informations. Il est aussi possible de récupérer des informations sur les procédures stockées en activant track_functions. Contrairement au paramètre précédent, il ne s'agit pas d'un booléen ici, mais d'un enum. Ce paramètre peut prendre trois valeurs, none pour ignorer les traces sur les fonctions, pl pour ne tracer que les fonctions en langage PL, et all pour tracer toutes les fonctions (utilisateurs). Ce paramètre étant assez récent, il est désactivé par défaut. Les informations récupérées sont disponibles à partir de la vue pg_stat_user_functions.

PostgreSQL met à jour le titre des processus pour donner différentes informations, notamment le type de requête exécutées pour les processus gérant les connexions utilisateurs. Cette mise à jour est plus ou moins performante suivant les systèmes. Le paramètre update_process_title permet de le désactiver s'il s'avère que cela prend trop de temps.

L'enregistrement des traces dans le fichier correspondant peut fortement affecter les performances pour les systèmes très utilisés. Depuis la version 8.4, il est possible d'indiquer un répertoire de stockage temporaire pour le fichier des statistiques. Le but est de pouvoir placer ce fichier sur un disque rapide, voire sur un montage en RAM, pendant son utilisation. Ce paramètre s'appelle stats_temp_directory.

7.2 Liste des vues directes

- Liste des sessions utilisateurs
  - pg_stat_activity
- Liste des sessions de réplication
  - pg_stat_replication
- Statistiques du background writer
  - pg_stat_bgwriter

La vue statistique pg_stat_activity renvoie une ligne par session connectée au serveur. Cette ligne indique la base, l'utilisateur, et le PID du processus serveur. Elle indique aussi le nom de l'application (à partir de la 9.0 et si l'application fournit cette information), l'adresse IP et le numéro de port du client, son nom d'hôte (si log_hostname est activé). Ensuite il y a trois informations d'horodatage : date de

lancement du processus serveur, date de lancement de la dernière transaction, date de lancement de la dernière requête. Et enfin, la dernière requête exécutée ou la requête en cours d'exécution se trouve dans le champ `query`, le champ `state` indique l'état du processus, et le champ `waiting` indique si la requête est bloquée en attente de verrou. Toutes ces informations sont très intéressantes dans une perspective de supervision occasionnelle, ainsi que dans le cas d'une supervision automatique. Par exemple, il est assez simple de trouver le nombre de connexions par base, par utilisateur ou par application. Il est aussi possible de récupérer rapidement le nombre de requêtes en cours d'exécution et le nombre de requêtes en attente d'un verrou, ce qui permet d'évaluer l'utilisation du serveur. Il est aussi possible de calculer la durée d'exécution la plus importante des requêtes.

La vue `pg_stat_replication` donne un grand nombre d'informations sur les sessions de réplication. Il y aura une ligne par connexion de réplication. Cela peut correspondre à un esclave ou à un outil comme `pg_basebackup`. Les informations importantes pour la supervision correspondent aux colonnes `sent_location`, `write_location`, `flush_location`, `replay_location` qui permettent de calculer le retard des esclaves par rapport au maître.

Enfin, la vue `pg_stat_bgwriter` donne toutes sortes d'informations sur les activités d'écriture sur les fichiers de données.

7.3 Liste des vues compteurs lignes

- `pg_stat_database`
- `pg_stat_all_tables`
- `pg_stat_all_indexes`

Trois vues donnent des informations au niveau des nombres de lignes (comme par exemple le nombre de lignes insérées, supprimées ou modifiées).

Commençons par `pg_stat_database`. Cette vue donne une ligne par base de données présente sur l'instance. Les premières colonnes précisent la base : `datid` est l'OID de la base (ce qui permet une jointure avec la colonne `oid` du catalogue système `pg_database`), `datname` est le nom de la base. Les autres colonnes sont les informations statistiques :

- `numbackends` indique le nombre de connexions en cours sur cette base ;
- `xact_commit` et `xact_rollback` correspondent respectivement au nombre de transactions validées et annulées sur cette base ;
- `blks_read` et `blks_hit` correspondent respectivement au nombre de blocs lus en dehors du cache de PostgreSQL et dans le cache de PostgreSQL ;
- `tup_returned` et `tup_fetched` correspondent au nombre de lignes renvoyées par les parcours de table et les parcours d'index ;
- `tup_inserted`, `tup_updated` et `tup_deleted` correspondent respectivement au nombre de lignes insérées, modifiées et supprimées ;

- `conflicts` indique le nombre de conflits avec la réplication (information disponible uniquement sur un esclave) ;
- `stats_reset` précise la date de la dernière réinitialisation des statistiques.

Les colonnes `xact_commit` et `xact_rollback` sont intéressantes pour connaître le nombre de transactions par unité de temps (minute ou seconde généralement). Surveiller `xact_rollback` est souvent intéressant pour s'assurer à ce qu'il n'y ait pas un accroissement brutal d'annulations de requêtes. Le ratio de lecture en cache (calculable avec `blks_read` et `blks_hit`) est aussi intéressant à calculer. Un faible ratio indiquer un problème dans la configuration de la taille du cache. Enfin, les statistiques sur le nombre de lignes permettent de générer quelques graphes intéressants comme le nombre de lignes lues par rapport au nombre de lignes insérées/modifiées/supprimées. Cela permet de mieux comprendre l'utilisation de la base.

Ensuite, il y a `pg_stat_all_tables`, avec ses versions dérivées `pg_stat_sys_tables` (uniquement les tables systèmes) et `pg_stat_user_tables` (uniquement les tables utilisateurs). Cette vue renvoie une ligne par table de la base. Elle comprend elle-aussi une première partie d'identification de l'objet : `relid` est l'OID de la table (ce qui permet une jointure avec la colonne `oid` du catalogue système `pg_class`), `schemaname` est le nom du schéma, `relname` est le nom de la table. Suivent les informations statistiques :

- `seq_scan` et `idx_scan` sont respectivement le nombre de parcours séquentiels et d'index ;
- `seq_tup_read` est le nombre de lignes lues suite à un parcours séquentiel ;
- `idx_tup_fetch` est le nombre de lignes lues suite à un parcours d'index ;
- `n_tup_ins` est le nombre de lignes insérées ;
- `n_tup_upd` est le nombre de lignes modifiées ;
- `n_tup_del` est le nombre de lignes supprimées ;
- `n_tup_hot_upd` est le nombre de lignes modifiées en utilisant la technique HOT ;
- `n_live_tup` est le nombre de lignes vivantes ;
- `n_dead_tup` est le nombre de lignes mortes ;
- `last_vacuum`, `last_autovacuum`, `last_analyze`, `last_autoanalyze` sont respectivement l'horodatage du dernier `VACUUM` manuel, du dernier `VACUUM` automatique, du dernier `ANALYZE` manuel et du dernier `ANALYZE` automatique ;
- `vacuum_count`, `autovacuum_count`, `analyze_count`, `autoanalyze_count` sont respectivement le nombre de `VACUUM` manuels, `VACUUM` automatiques, `ANALYZE` manuels et `ANALYZE` automatiques.

Cette vue statistique est moins fréquemment utilisée dans la supervision automatique.

Enfin, il y a `pg_stat_all_indexes`, avec ses versions dérivées `pg_stat_sys_indexes` (uniquement les index systèmes) et `pg_stat_user_indexes` (uniquement les index utilisateurs). Cette vue renvoie une ligne par index de la base. Elle comprend elle-aussi une première partie d'identification de l'objet : `relid` est l'OID de la table (ce qui permet une jointure avec la colonne `oid` du catalogue système `pg_class`), `indexrelid` est l'OID de l'index (ce qui permet une jointure avec la colonne `oid` du

catalogue système pg_class), schemaname est le nom du schéma, relname est le nom de la table, indexrelname est le nom de l'index. Suivent les informations statistiques :

- idx_scan correspond au nombre de parcours d'index ;
- idx_tup_read est le nombre d'entrées lues dans l'index ;
- idx_tup_fetch est le nombre de lignes lues dans la table.

Là-aussi, cette vue statistique est peu fréquemment utilisée dans la supervision automatique.

7.4 Liste des vues compteurs blocs

- pg_statio_all_tables
- pg_statio_all_indexes
- pg_statio_all_sequences

Ces trois vues donnent des informations au niveau des nombres de blocs. À chaque fois, le but est de savoir si les blocs sont lus dans le cache de PostgreSQL (mot clé hit) ou en dehors (mot clé read).

pg_statio_all_tables donne une liste de tables avec des informations sur les blocs :

- heap_blks_read, nombre de blocs lus dans la partie HEAP de la table, en dehors du cache de PostgreSQL ;
- heap_blks_hit, nombre de blocs lus dans la partie HEAP de la table, dans le cache de PostgreSQL ;
- idx_blks_read, nombre de blocs lus dans la partie HEAP de l'index, en dehors du cache de PostgreSQL ;
- idx_blks_hit, nombre de blocs lus dans la partie HEAP de l'index, dans le cache de PostgreSQL ;
- toast_blks_read, nombre de blocs lus dans la partie TOAST de la table, en dehors du cache de PostgreSQL ;
- toast_blks_hit, nombre de blocs lus dans la partie TOAST de la table, dans le cache de PostgreSQL ;
- tidx_blks_read, nombre de blocs lus dans la partie TOAST de l'index, en dehors du cache de PostgreSQL ;
- tidx_blks_hit, nombre de blocs lus dans la partie TOAST de l'index, dans le cache de PostgreSQL.

PostgreSQL est capable d'externaliser le stockage de données volumineuses dans un fichier nommé TOAST (acronyme de « The Oversized-Attribute Storage Technique »). Chaque table se voit attribuer un fichier standard (appelé aussi HEAP), et un fichier d'externalisation des grosses données (appelé TOAST).

`pg_statio_all_indexes` donne une liste d'index avec des informations sur les blocs. Cependant, au contraire de `pg_statio_all_tables`, cela reste un résumé. Voici les deux colonnes statistiques :

- `idx_blks_read`, nombre de blocs lus dans l'index, en dehors du cache de PostgreSQL ;
- `idx_blks_hit`, nombre de blocs lus dans l'index, dans le cache de PostgreSQL.

Enfin, `pg_statio_all_sequences` donne une liste des séquences avec des informations sur les blocs. Là-aussi, cela reste un résumé. Voici les deux colonnes statistiques :

- `idx_blks_read`, nombre de blocs lus dans la séquence, en dehors du cache de PostgreSQL ;
- `idx_blks_hit`, nombre de blocs lus dans la séquence, dans le cache de PostgreSQL.

7.5 Informations intéressantes à récupérer

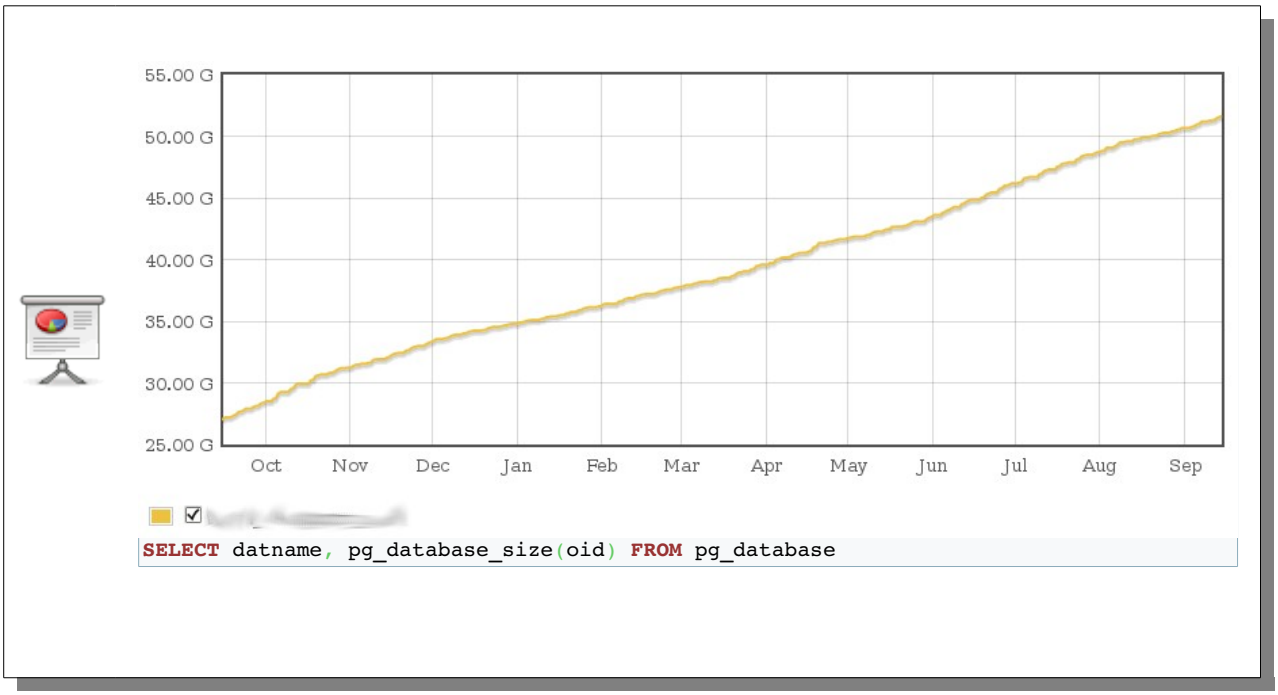
- sur l'activité
- sur l'instance
- sur les bases
- sur les tables
- sur les index
- sur les fonctions

Il existe un grand nombre d'informations intéressantes à récupérer pour de la supervision automatique. Nous n'en verrons que quelques exemples.

7.6 Nombre de connexions par base

Ces deux requêtes donnent la même information, à savoir le nombre de connexions en cours pour chaque base de données de l'instance. Le graphe ci-dessus affiche le résultat de la requête exécutée toutes les cinq minutes, sur une semaine. On voit bien ici qu'il n'y a pas de pics importants. Le nombre de connexions est simple et ne change pas fondamentalement. Dans ce type de graphe, il faut surtout faire attention aux gros pics ainsi qu'aux montées lentes mais qui ne retombent jamais. Dans les deux cas, cela peut nécessiter une meilleure configuration du paramètre `max_connections` ou l'installation d'un pooler de connexions.

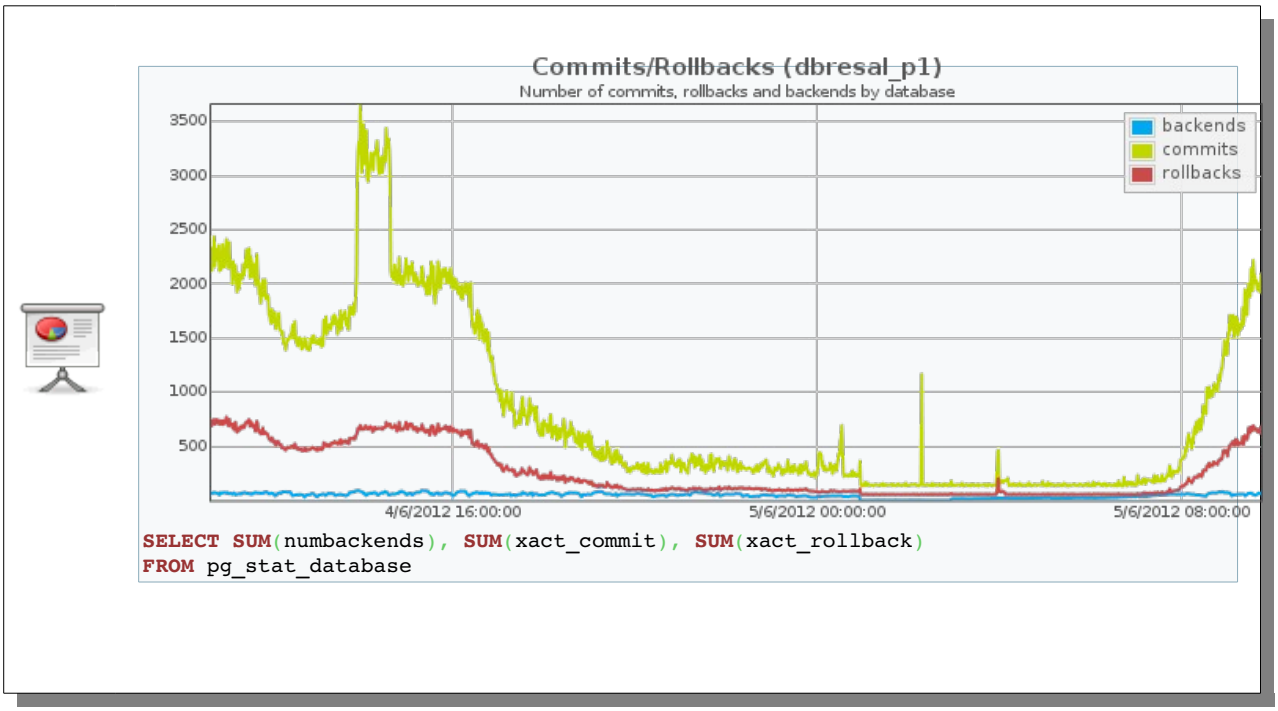
7.7 Taille des bases



Le graphe affiche ici la montée de la volumétrie d'une base sur une année complète. L'augmentation est peu importante jour par jour, mais sur une année, elle représente quand même 30 Go, ce qui correspond à un doublage de la taille de la base en une année.

Dans ce type de graphes, le point essentiel à regarder est les pics. Un pic vers le haut laisse supposer un ajout important de données. Si cela survient fréquemment, il faudra surveiller de près l'espace libre du disque. Un pic vers le bas laisse supposer une suppression massive de données. Cela peut être le résultat d'un `VACUUM FULL`, d'un `REINDEX`, voire d'un archivage des anciennes données. C'est une information importante à connaître, pour être sûr que des données ne sont pas en train de disparaître (par exemple à cause d'un script mal codé).

7.8 COMMIT et ROLLBACK



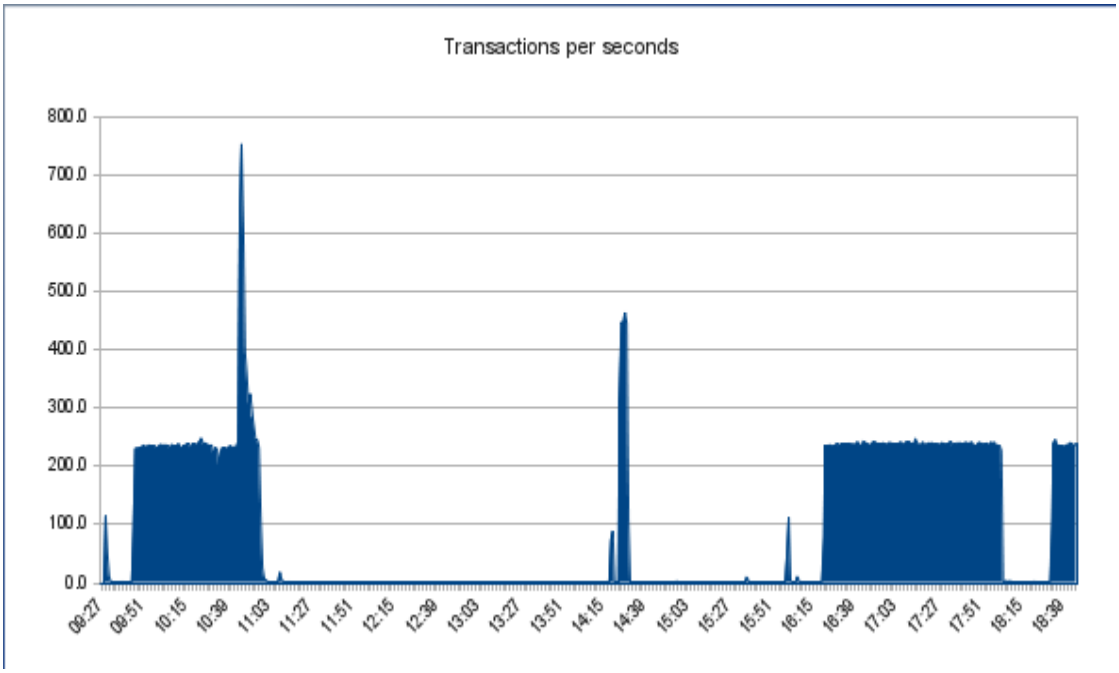
Cette statistique est intéressante, notamment après une mise à jour applicative. Elle permet de vérifier que les transactions sont bien commitées.

Dans le cas de ce graphe, on remarque que l'activité a lieu surtout la journée, même si le nombre de clients connectés n'augmente pas. On remarque aussi que le nombre de transactions annulées est assez haut, ce qui est préoccupant.

7.9 Nombre de transactions/seconde

```
SELECT xact_commit+xact_rollback
FROM pg_stat_database
```

- Y soustraire la même information d'il y a une minute
- Diviser le tout par 60



Trouver le nombre de transactions est assez simple. Il suffit d'additionner les colonnes `xact_commit` et `xact_rollback` et de soustraire le résultat au résultat de la même opération une minute avant. Cela donne le nombre de transactions réalisées en une minute.

Dans le cas du graphe ci-dessus, on remarque bien les périodes de tests (de 9h50 à 11h00, puis de 16h20 à 18h10).

7.10 Ratio de lecture en cache

```
SELECT datname, 100.*blks_hit/(blks_read+blks_hit)
FROM pg_stat_database
WHERE blks_read+blks_hit>0
```

Cette requête calcule le ratio des lectures en cache. Généralement, avec un serveur PostgreSQL bien configuré, cela doit se situer entre 95 et 100%. Si le ratio est plus petit, il est important de se demander s'il ne serait pas nécessaire de grossir le cache de PostgreSQL, voire d'ajouter de la mémoire au serveur.

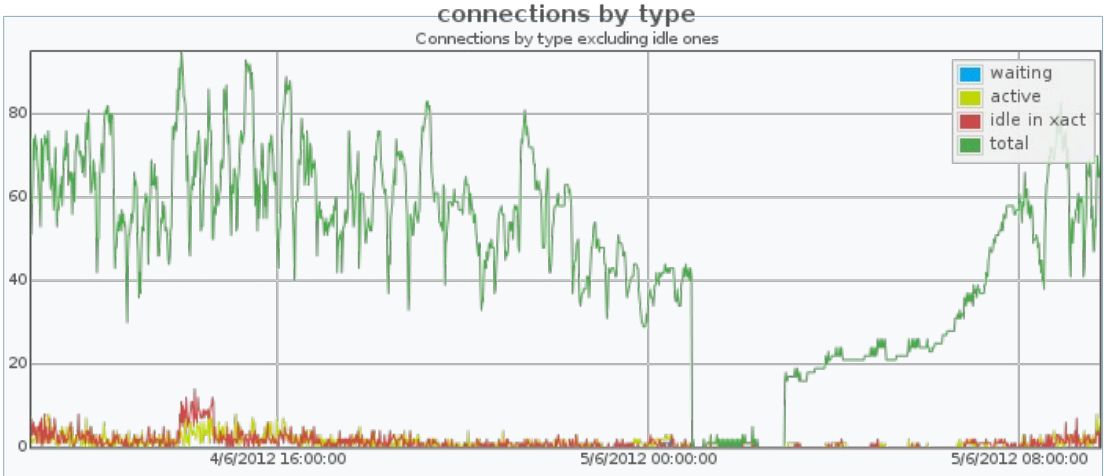
Typiquement, cette sonde a plutôt pour but de générer une alerte en cas de franchissement d'un seuil, plutôt que de créer un graphe.

7.11 Durée maximale d'une requête

```sql
SELECT MAX(now()-query_start) FROM pg_stat_activity
```

Cette requête permet de récupérer la durée d'exécution de la plus grosse requête. En conservant cette information dans le temps, il est possible de savoir si le système est plus ou moins chargé à un instant t.

7.12 Nombre de sessions par type

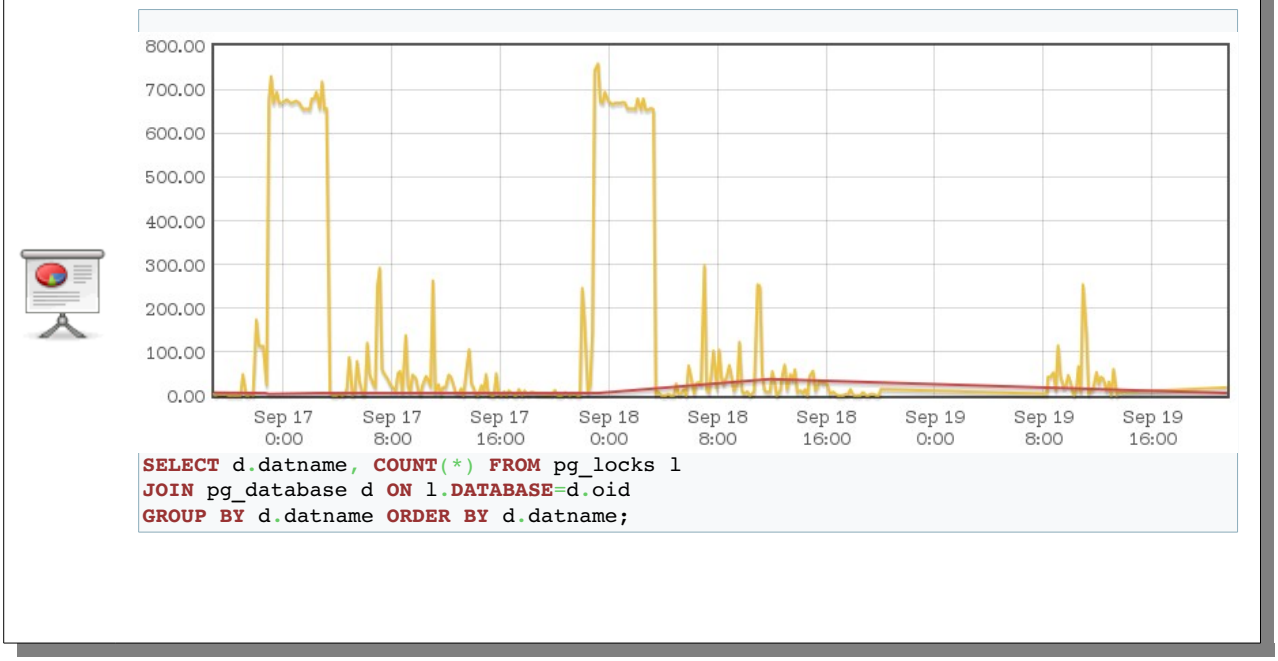



```sql
SELECT date_trunc('seconds', CURRENT_TIMESTAMP) AS datetime,
  COUNT(*) AS total,
  COALESCE(SUM((state='active')::INTEGER), 0) AS active,
  COALESCE(SUM(waiting::INTEGER), 0) AS waiting,
  COALESCE(SUM((state='idle in transaction')::INTEGER), 0) AS idle_in_xact
FROM pg_stat_activity WHERE pid <> pg_backend_pid();
```

Cette requête permet de savoir à un instant le nombre de connexions totales, le nombre de connexions en attente d'un verrou (`waiting`), le nombre de connexions en transactions ne faisant rien (`idle in xact`) et le nombre de connexions actives.

Dans le graphe, le nombre de connexions totales est très supérieur au nombre de connexions actives. Par contre, on remarque aucune connexion en attente, ce qui est un excellent point.

7.13 Nombre de verrous par base

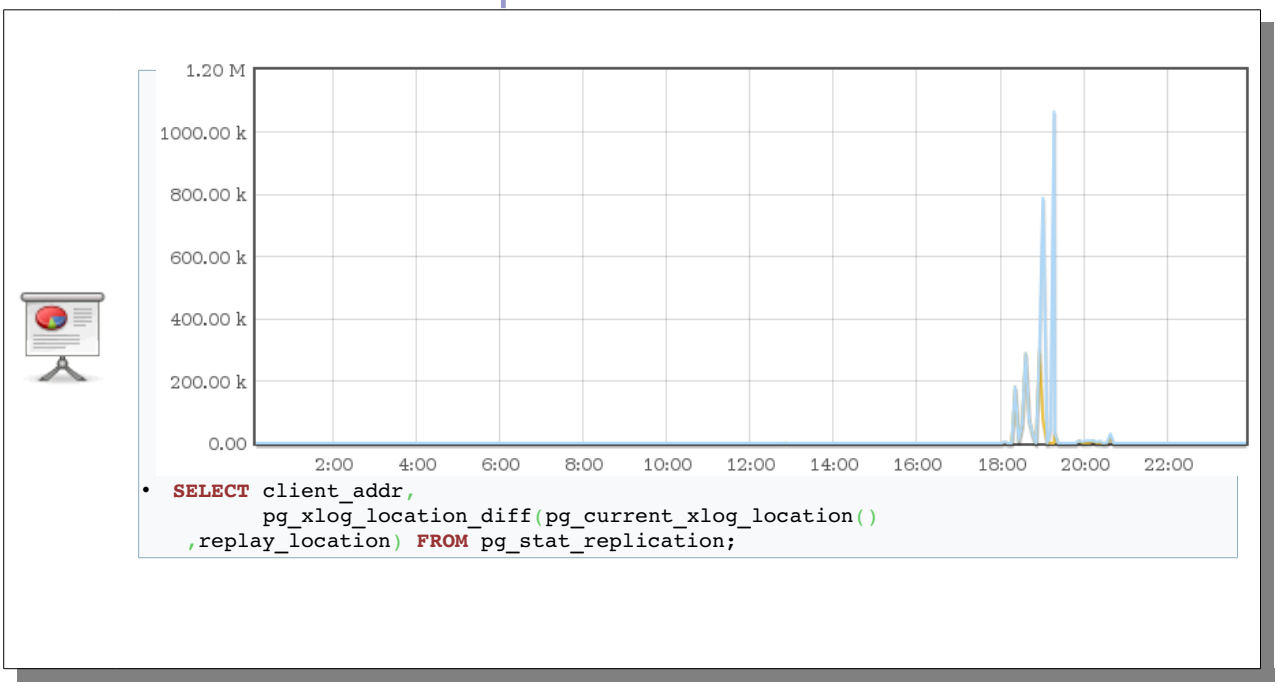


La requête ci-dessus récupère le nombre de verrous par base. Les gros pics visibles sont dus aux opérations de sauvegarde.

Il est parfois plus intéressant de décompter les verrous par type de verrou. Une telle requête pourrait s'écrire ainsi :

```
SELECT mode, COUNT(*) FROM pg_locks
GROUP BY mode
ORDER BY mode;
```

7.14 Retard de réplication sur les esclaves



Cette requête permet de récupérer l'adresse IP de l'esclave, ainsi que la position du maître et de l'esclave dans les journaux de transactions.

Le graphe est un peu différent vu qu'il n'affiche que la différence en octets entre la position du maître et celle d'un esclave. On remarque ainsi que le retard est insignifiant la majorité du temps, sauf entre 18h et 20h où un batch d'insertion de données impose un retard plus conséquent sur l'esclave.

Attention, avant 9.2, la fonction pg_xlog_location_diff() n'existait pas. Il fallait donc récupérer les informations avec la requête suivante :

```
SELECT client_addr,
       pg_current_xlog_location(), replay_location
  FROM pg_stat_replication;
```

puis faire le calcul soi-même. En considérant les adresses “W/SSSSSSSS” pour le maitre et “w/ssssssss” pour l'esclave, le calcul effectué pour retrouver le retard en terme de volumétrie est le suivant :

`(0xFF000000 * W + SSSSSSSS) - (0xFF000000 * w + ssssssss)`
Le résultat est alors exprimé en **hexadécimal**.

8 pgCluu

- script Perl
- récupère des information à intervalle régulier
  - PostgreSQL
  - système (nécessite sysstat)
- Génération d'un rapport HTML très détaillé

8.1 pgCluu - récupération des métriques

- script `pgcluu_collectd`
  - -i intervalle
  - -D : mode daemon
  - -h, -p ...
  - dernier argument: répertoire de stockage
    - attention à l'espace disque !

Le binaire `pgcluu_collectd` permet de lancer la collecte de nombreuses métriques. Il peut être lancé en intéractif (par défaut) ou en tâche de fond (option `-D`). Les principaux arguments sont :

- `-i` : pour spécifier l'intervalle de récupération des métrique (en secondes). Par défaut à 60 secondes.
- `-D` : lancer en tant que daemon
- `-k` : arrêter un daemon pgCluu
- `-h`, `-p`, `-U` : options de connexion classiques à la base

Enfin, le dernier argument passé à `pgcluu_collectd` spécifie le répertoire qui contiendra tous les fichiers de métrique. Ce répertoire doit exister et doit être vide avant de lancer `pgcluu_collected`. Selon la fréquence choisie ainsi que le matériel présent, ces données peuvent prendre beaucoup de place sur le disque. Il est donc nécessaire de s'assurer que l'espace disponible sur le système de fichier est suffisant.

8.2 pgCluu - génération d'un rapport

- script `pgcluu`
- -o répertoire de destination
- dernier argument: répertoire des métriques

Une fois les métriques disponibles, il est nécessaire de faire appel au binaire `pgcluu` afin de générer un rapport html à partir de celles-ci. Les principales options sont :

- `-o` : permet de définir le répertoire de sortie des fichiers html. Ce répertoire doit exister et être vide.
- dernier argument : répertoire contenant les métriques générées par `pgcluu_collected`

9 Supervision avec Nagios

- principe
- sondes
- éléments de configuration

9.1 Nagios - généralités

- Outil largement utilisé
- Grand parc de serveurs
- Utilisation de sondes
- Gestion d'alertes
- Configuration ardue
- Très bien documenté

Nagios est un outil libre très largement utilisé en entreprise. Il est particulièrement utile lorsqu'on a de nombreux serveurs à superviser.

Nagios est modulaire, et se base sur l'appel à des sondes externes afin de pouvoir superviser de nombreuses chose. Il est fourni avec un certains nombre de sondes généralistes (nagios-plugins), par exemple une sonde pour vérifier qu'une connexion ssh est possible ou vérifier la charge (`load`) d'un serveur. De nombreuses sondes plus

spécialisées et open-source existent afin d'étendre facilement sa portée. Il est également très simple de créer des nouvelles sondes afin de couvrir un besoin particulier.

Le but principal de Nagios est de gérer l'envoi d'alertes automatiques en fonction des sondes configurées. Il est réputé pour être assez compliqué à configurer. L'outil est cependant très bien documenté, et divers projets existent afin de faciliter sa configuration. Un exemple d'outil est le projet centreon : http://www.centreon.com/. Cet outil contient à la fois Nagios ainsi qu'une interface de configuration spécialement développée afin de pouvoir effectuer toute la configuration au travers d'une interface web.

9.2 Nagios - sondes

- Une fréquence
- Des seuils
- Des états

Chaque sonde configurée dans Nagios peut avoir sa propre fréquence de vérification. Par exemple, il est intéressant de superviser la charge d'un serveur très régulièrement, alors que la vérification de la présence de mises à jour ne nécessite pas une granularité de quelques minutes.

Il est ensuite nécessaire de spécifier des seuils pour chacunes des sondes. Chaque sonde gère deux seuils : *WARNING* et *CRITICAL*. Lors de l'exécution de la sonde, l'indicateur supervisé sera comparé à ces seuils, ce qui donnera l'état de la sonde au moment de son exécution. Une sonde peut avoir 4 états :

- 0 - OK
- 1 - WARNING
- 2 - CRITICAL
- 3 - UNKNOWN

Cet état est récupéré au niveau du code de sortie de la sonde.

Nagios peut alors envoyer des alertes en cas de changement d'état. Celles-ci sont généralement envoyées par mail, mais il est possible en fonction du matériel disponible de les envoyer par sms par exemple.

9.3 Nagios - sondes: perfdata

- détails sur l'indicateur
- spécifique à chaque sonde
- possibilité de graph
- limité à 4096 caractères

Nagios spécifie un format, appelé perfdata, permettant de renvoyer le détail des différents indicateurs supervisés. Ces données sont facultatives, mais un grand nombre de sondes le gère. Le format de sortie texte d'une sonde est le suivant :

```
MESSAGE_HUMAIN | indicateur1=valeur1;SEUIL_WARNING;SEUIL_CRITICAL
indicateur2=valeur2;SEUIL_WARNING;SEUIL_CRITICAL ...
```

Le caractère **|** délimite le message humain des données de performances. Chaque indicateur est séparé par un espace. Le rappel des différents seuils est facultatif.

Par exemple, dans le cas de la supervision du LOAD, les valeurs pour le load1, load5 et load15 seront renvoyées. Il est possible de configurer Nagios afin de générer un fichier de performance sur disque contenant ces différentes valeurs pour chaque appel de sonde. Un système externe peut alors récupérer ces différents fichiers, les analyser et insérer ces données en bas afin de pouvoir générer des graphes à partir de ces données.

Pour plus d'information sur ce paramétrage, il est nécessaire de se reporter à la documentation Nagios, plus particulièrement http://nagios.sourceforge.net/docs/3_0/perfdata.html et http://nagios.sourceforge.net/docs/3_0/pluginapi.html.

9.4 Nagios - principe de configuration

- /etc/nagios/ (RHEL/Centos)
- /etc/nagios3 et /etc/nagios-plugins/ (debian/ubuntu)
- définitions par blocs
- commandes et services
- de nombreuses façons de configurer

Les fichiers de configuration se trouvent dans `/etc/nagios` ou `/etc/nagios3`. Sur les distributions Debian/Ubuntu, une partie de la configuration se trouve également dans `/etc/nagios-plugins`.

La configuration du moteur nagios s'effectue dans le fichier `nagios.cfg`. Toute le reste de la configuration s'effectue dans les différents fichiers présents dans les deux répertoires spécifiés, grâce à une directive d'inclusion présente dans le fichier `nagios.cfg`. Par exemple sur une distribution debian :

```
# Debian also defaults to using the check commands defined by the debian
# nagios-plugins package
cfg_dir=/etc/nagios-plugins/config

# Debian uses by default a configuration directory where nagios3-common,
# other packages and the local admin can dump or link configuration
# files into.
cfg_dir=/etc/nagios3/conf.d
```

Les répertoires étant inclus, tout le reste de la configuration peut se faire dans le ou les fichiers de son choix.

La définition d'un bloc se fait ainsi :

```
define type_bloc {
    #définition
}
```

La plus grande partie de la configuration consiste à définir les **commandes** (appels aux sondes) qui seront effectuées, et définir les différents **services** appelant ces commandes. Toutes les commandes des sondes disponibles par défaut sont automatiquement configurées, par exemple pour la vérification du `load` :

```
# 'check_load' command definition
define command {
    command_name    check_load
    command_line    /usr/lib/nagios/plugins/check_load --warning='$ARG1$,$ARG2$,$ARG3$'
--critical='$ARG4$,$ARG5$,$ARG6$'
}
```

On voit ici que la sonde check_load est appelée avec une utilisation de 6 paramètres ($ARG1$ à $ARG6$), permettant de spécifier les 3 valeurs (`load1`, `load5` et `load15`) de chaque seuil (`warning` et `critical`).

9.5 Debug

- Débugger la configuration de Nagios
  - `nagios -v /etc/nagios/nagios.cfg`

```
[root@localhost ~]# nagios -v /etc/nagios/nagios.cfg

Nagios Core 3.5.1
Copyright (c) 2009-2011 Nagios Core Development Team and Community Contributors
Copyright (c) 1999-2009 Ethan Galstad
Last Modified: 08-30-2013
License: GPL

Website: http://www.nagios.org
Reading configuration data...
   Read main config file okay...
Processing object config file '/etc/nagios/objects/commands.cfg'...
Processing object config file '/etc/nagios/objects/contacts.cfg'...
```

```
Processing object config file '/etc/nagios/objects/timeperiods.cfg'...
Processing object config file '/etc/nagios/objects/templates.cfg'...
Processing object config file '/etc/nagios/objects/localhost.cfg'...
Processing object config directory '/etc/nagios/conf.d'...
Processing object config file '/etc/nagios/conf.d/services_template.cfg'...
Warning: Duplicate definition found for service 'generic-service' (config file
'/etc/nagios/conf.d/services_template.cfg', starting on line 2)
Error: Could not add object property in file '/etc/nagios/conf.d/services_template.cfg' on line 3.
   Error processing object config files!


***> One or more problems was encountered while processing the config files...

     Check your configuration file(s) to ensure that they contain valid
     directives and data defintions.  If you are upgrading from a previous
     version of Nagios, you should be aware that some variables/definitions
     may have been removed or modified in this version.  Make sure to read
     the HTML documentation regarding the config files, as well as the
     'Whats New' section to find out what has changed.
```