Optimiser une instance PostgreSQL

Atelier Performances

1.2

Table des matières

Au menu

- Paramétrage PostgreSQL
- Surveiller l'activité de l'instance
- Surveiller l'activité SQL
- Les outils
- Tp

Gestion de la mémoire

- Zone de mémoire partagée
  - `shared_buffers`
- Zone de chaque processus
  - tris en mémoire / aggregat (`work_mem`)
  - création d'index, vacuum (`maintenance_work_mem`)

Autres paramètres à surveiller

- `max_wal_size` (réduire les FPI et donc les I/O)
- paramètrage autovacuum (contenir la fragmentation)
- `jit` (à désactiver globalement)
- `max_worker_processes`, `max_parallel_[maintenance]_workers`

Surveiller l'activité SQL

- Quelles sont les requêtes lentes ?
- Quelles sont les requêtes les plus fréquentes ?
- Quelles requêtes génèrent des fichiers temporaires ?
- Quelles sont les requêtes bloquées ?
  - et par qui ?
- Progression d'une requête de maintenance

Trace des requêtes exécutées

- `log_min_duration_statements` = <temps minimal d'exécution>
  - `0` permet de tracer toutes les requêtes
  - trace des paramètres
  - traces exploitables par des outils tiers
  - pas d'informations sur les accès, ni les plans d'exécution
- `log_min_duration_sample` = <temps minimal d'exécution>
  - `log_statement_sample_rate` et/ou `log_transaction_sample_rate`
  - trace d'un ratio des requêtes

Trace des fichiers temporaires

- `log_temp_files = \<taille minimale\>`
  - `0` trace tous les fichiers temporaires
  - associe les requêtes SQL qui les génèrent
  - traces exploitables par des outils tiers

Fragmentation des tables et index

- Dépôt github : https://github.com/ioguix/pgsql-bloat-estimation
- Scripts permettant d'estimer la fragmentation
  - des tables : `table_bloat.sql`
  - des index btree : `btree_bloat.sql`
- Permet d'adapter le comportement de l'`autovacuum` sur les tables
- Plus précis : pgstattuple

Paramétrage du planificateur

- 2 paramètres qui permettent au planificateur de limiter sa tache d'optimisation
  - `join_collapse_limit` (défaut = 8) : le planificateur réécrit les constructions des `JOIN` explicites en une liste d'éléments `FROM`
  - `from_collapse_limit` (défaut = 8) : le planificateur assemble les sous-requêtes dans des requêtes supérieures
  - `random_page_cost`
  - `effective_cache_size`
  - `max_parallel_workers_per_gather`

Outils d'analyse

- Différents outils existent autour de PostgreSQL
- Outils d'analyse occasionnel et temps réel :
  - pg_activity, …
- Outils d'analyse des traces :
  - pgBadger, …

Outils d'analyse

- Outils d'analyse des statistiques :
  - pgCluu, `pg_stat_statements`, PoWA
- Outils d'analyse de plan d'Exécution :
  - explain.dalibo.com, explain.depesz.com
- Outils de supervision
  - temBoard, Nagios, Zabbix, Prometheus, …

---

pg_activity

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

```
sudo -u postgres pg_activity -U postgres
```

---

pgBadger

- Script Perl, très simple d'utilisation
- En entrée : un ou plusieurs logs postgres
- En sortie : un rapport HTML très complet
- Site : https://pgbadger.darold.net/

---

pgBadger

- `--top` <n> : nombre de requêtes à afficher, par défaut 20
- `--extension` <format> : format de sortie (html, text, bin, json ou tsung)
- `--dbname` <database> : choix de la base à analyser
- `--dbuser` <user> : permet de spécifier un utilisateur à analyser
- `--exclude_user` <user> : permet d'exclure un utilisateur de l'analyse

---

pgBadger

- `--prefix` <log_line_prefix> : permet d'indiquer le format utilisé dans les logs
- `--begin` <date> `--` end <date> : permet d'indiquer la plage horaire du rapport
- `j ou --jobs` <n> : permet de paralléliser l'analyse des logs
- `--timezone` <+/- XX> : permet d'ajuster les fuseaux horaires dans les graphes

---

PostgreSQL Workload Analyzer (POWA)

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

Aucune historisation n'est en effet réalisée par `pg_stat_statements`. PoWA a été développé pour combler ce manque et ainsi fournir un outil équivalent à AWR d'Oracle, permettant de connaître l'activité du serveur sur une période donnée.

Sur l'instance de production de Dalibo, la base de données PoWA occupe moins de 300 Mo sur disque, avec les caractéristiques suivantes :

- 10 jours de rétention

- fréquence de capture : 1 min
- 17 bases de données
- 45263 requêtes normalisées
- dont ~28 000 COPY, ~11 000 LOCK
- dont 5048 requêtes applicatives

temBoard

- Dépôt github : https://github.com/dalibo/temboard
- Licence: PostgreSQL
- Notes: Serveur sur Linux, client web

temBoard - PostgreSQL Remote Control

- Multi-instances
- Surveillance OS / PostgreSQL
- Suivi de l'activité
- Configuration de chaque instance
- temBoard est un outil permettant à un DBA de mener à bien la plupart de ses tâches courantes.

Vue pg_stat_statements

- Ajoute la vue statistique `pg_stat_statements`
- Les requêtes sont normalisées
- Indique les requêtes exécutées, avec durée d'exécution, utilisation du cache, etc.

Contrairement à pgBadger, `pg_stat_statements` ne nécessite pas de tracer les requêtes exécutées. Il se branche (*hook*) à plusieurs endroits (plannification, exécution) pour récupérer directement les métriques utiles, celles-ci étant agrégées par requête normalisée, utilisateur et base.

Voici un exemple de requête sur la vue `pg_stat_statements` :

```
SELECT
    datname,
    usename,
    substring(query FROM 1 FOR 80),
    (total_exec_time / 1000 / 3600)::int AS exec_time_hours,
    max_exec_time,
    calls
FROM
    pg_stat_statements
    JOIN pg_database d ON (dbid = d.oid)
    JOIN pg_user u ON (userid = usesysid)
ORDER BY
    total_exec_time DESC
LIMIT 3;

-[ RECORD 1
 ↪  ]---+-----------------------------------------------------------------------------------
datname         | stack
usename         | postgres
substring       | SELECT * FROM interaction WHERE interac = $1
exec_time_hours | 269
max_exec_time   | 87766.553242
calls           | 34059
-[ RECORD 2
 ↪  ]---+-----------------------------------------------------------------------------------
datname         | stack
usename         | postgres
substring       | select * from posthistory where extract($1 from creationdate) = $2
exec_time_hours | 124
max_exec_time   | 49202.522936999994
calls           | 34061
-[ RECORD 3
 ↪  ]---+-----------------------------------------------------------------------------------
datname         | stack
usename         | postgres
substring      | SELECT COUNT(ph.posthistorytypeid) AS nb, pht.name AS name FROM posthistory ph J
exec_time_hours | 114
max_exec_time   | 40136.762156000004
calls           | 34061
```

Vue pg_stat_statements

Métriques les plus intéressantes :

- `total_exec_time` (requêtes consommatrices)
- `max_exec_time` (requêtes lentes unitairement)
- `temp_blks_written` (fichiers temporaires)
- `shared_blks_*` (utilisation du cache)
- `blk_*_time` (temps consacré aux I/O)