−Table des matières

- Gestion mémoire avec PostgreSQL

Gestion mémoire avec PostgreSQL

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 107 (Juillet/Août 2008). Il est disponible maintenant sous licence Creative Commons.

Mieux gérer la mémoire est un excellent moyen pour gagner en performance et en réactivité. Ce n'est évidemment pas le seul, mais c'est celui qui permettra le gain le plus important. Malheureusement, peu de documentation existe sur ce thème. Les informations sont généralement diluées dans la documentation officielle et dans les sources. Cet article a pour but de faire un point complet sur ce thème.

PostgreSQL gère deux types d'allocations mémoire : une allocation globale au serveur et donc partagée par les différents processus exécutés par le serveur PostgreSQL, et une mémoire spécifique pour chaque processus dans le but de satisfaire les besoins particuliers de certaines actions.

Mémoire globale

Cache disque des fichiers de données

La lecture sur disque étant bien plus lente que la lecture en mémoire vive, PostgreSQL constitue un cache des fichiers de données en mémoire. Mais avant de détailler cette partie, commençons par nous intéresser à ce que PostgreSQL stocke sur disque. Nous allons restreindre la discussion uniquement à la partie que PostgreSQL pourra placer sur son cache.

Lors de l'initialisation du cluster avec la commande initdb, le répertoire de stockage est créé et rempli de répertoires et fichiers nécessaires au démarrage de PostgreSQL. Ce répertoire est précisé soit en argument de la commande initdb (avec l'option -D) soit par l'intermédiaire de la variable d'environnement $PGDATA. Dans ce répertoire se trouve par défaut un sous-répertoire base. Ce sous-répertoire va contenir toutes les bases de données ainsi que les objets physiques qui y seront stockés.

Supposons que nous voulons utiliser le répertoire /opt/mon_cluster comme répertoire de stockage. Nous commençons par initialiser le cluster :

```
guillaume@laptop:~$ initdb -D /opt/mon_cluster
Les fichiers de ce cluster appartiendront à l'utilisateur « guillaume ».
Le processus serveur doit également lui appartenir.
[...]
Succès. Vous pouvez maintenant lancer le serveur de bases de données par :

    postgres -D /opt/mon_cluster
ou
    pg_ctl -D /opt/mon_cluster -l journal_applicatif start
```

Le répertoire /opt/mon_cluster contient maintenant :

```
guillaume@laptop:~$ cd /opt/mon_cluster
guillaume@laptop:/opt/mon_cluster$ ls -lF
total 64
drwx------ 5 guillaume guillaume  4096 2008-05-24 15:20 base/
drwx------ 2 guillaume guillaume  4096 2008-05-24 15:20 global/
drwx------ 2 guillaume guillaume  4096 2008-05-24 15:19 pg_clog/
-rw------- 1 guillaume guillaume  3429 2008-05-24 15:19 pg_hba.conf
-rw------- 1 guillaume guillaume  1460 2008-05-24 15:19 pg_ident.conf
drwx------ 4 guillaume guillaume  4096 2008-05-24 15:19 pg_multixact/
drwx------ 2 guillaume guillaume  4096 2008-05-24 15:19 pg_subtrans/
drwx------ 2 guillaume guillaume  4096 2008-05-24 15:19 pg_tblspc/
drwx------ 2 guillaume guillaume  4096 2008-05-24 15:19 pg_twophase/
-rw------- 1 guillaume guillaume     4 2008-05-24 15:19 PG_VERSION
drwx------ 3 guillaume guillaume  4096 2008-05-24 15:19 pg_xlog/
-rw------- 1 guillaume guillaume 16591 2008-05-24 15:19 postgresql.conf
```

et le sous-répertoire base contient :

```
guillaume@laptop:/opt/mon_cluster$ cd base
guillaume@laptop:/opt/mon_cluster/base$ ls -lF
total 12
drwx------ 2 guillaume guillaume 4096 2008-05-24 15:20 1/
drwx------ 2 guillaume guillaume 4096 2008-05-24 15:20 11510/
drwx------ 2 guillaume guillaume 4096 2008-05-24 15:20 11511/
```

Chaque sous-répertoire du répertoire base correspond à une base de données. Ici, il en existe trois car la commande initdb prépare trois bases de données par défaut : template0, template1 et postgres. Les numéros constituant le nom des répertoires est l'OID des bases de données.

```
guillaume@laptop:/opt/mon_cluster/base$ pg_ctl -D /opt/mon_cluster start
serveur en cours de démarrage
guillaume@laptop:/opt/mon_cluster/base$ psql -q postgres
postgres=# SELECT oid, datname FROM pg_database ORDER BY oid;
  oid  |  datname
-------+-----------
     1 | template1
 11510 | template0
 11511 | postgres
(3 lignes)
```

Maintenant, créons une base de donnée :

```
postgres=# CREATE DATABASE glmf;
postgres=# SELECT oid, datname FROM pg_database ORDER BY oid;
  oid  |  datname
-------+-----------
     1 | template1
 11510 | template0
 11511 | postgres
 16384 | glmf
(4 lignes)

postgres=# \q
```

L'OID de la nouvelle base de données est 16384. Vérifions qu'un sous-répertoire de nom 16384 existe bien :

```
guillaume@laptop:/opt/mon_cluster/base$ ls -lFd 16384
drwx------ 2 guillaume guillaume 4096 2008-05-24 15:48 16384/
```

C'est bien le cas.

Chaque répertoire de base de données contient deux types d'objets : les tables et les index. Tous les autres objets n'ont pas d'existence propre. Par exemple, en ce qui concernent les vues, seules leurs définitions sont stockées dans un catalogue système, ce qui fait que PostgreSQL ne permet pas de créer actuellement de vues matérialisées (des vues qui ont une existence propre sur disque).

Note : le seul moyen d'avoir des vues matérialisées avec PostgreSQL est d'émuler soi-même ce comportement avec le système des règles.

Le répertoire de base de données contient déjà des objets systèmes :

```
guillaume@laptop:/opt/mon_cluster/base$ ls -lF 16384
total 4220
-rw------- 1 guillaume guillaume  40960 2008-05-24 15:48 11429
-rw------- 1 guillaume guillaume      0 2008-05-24 15:48 11431
-rw------- 1 guillaume guillaume   8192 2008-05-24 15:48 11433
-rw------- 1 guillaume guillaume   8192 2008-05-24 15:48 11434
-rw------- 1 guillaume guillaume      0 2008-05-24 15:48 11436
[...]
guillaume@laptop:/opt/mon_cluster/base$ find 16384 -type f | wc -l
129
```

Il existe donc 129 objets par défaut dans une base. Créons maintenant une nouvelle table :

```
guillaume@laptop:/opt/mon_cluster/base$ psql -q glmf
glmf=# CREATE TABLE magazine (id integer);
```

Nous venons de créer une table, nommée magazine, contenant une seule colonne de type entier. Cette fois, PostgreSQL n'utilise pas l'OID comme nom de fichier mais son relfilenode. La preuve :

```
glmf=# SELECT relname, relfilenode FROM pg_class WHERE relname='magazine';
 relname  | relfilenode
----------+-------------
 magazine |       16387
(1 ligne)

glmf=# \! ls -lF 16384/16387
-rw------- 1 guillaume guillaume 0 2008-05-24 16:08 16384/16387
```

La table ne contient aucune donnée, elle fait donc 0 octets. En effet, les informations concernant sa contruction sont stockées dans des catalogues systèmes et, sans données, la table n'est qu'un fichier vide. Ajoutons donc une ligne :

```
glmf=# SELECT * FROM magazine;
 id
----
(0 lignes)

glmf=# INSERT INTO magazine (id) VALUES (104);
glmf=# SELECT * FROM magazine;
 id
-----
 104
(1 ligne)

glmf=# \! ls -lF 16384/16387
-rw------- 1 guillaume guillaume 8192 2008-05-24 16:09 16384/16387
```

En ajoutant un simple entier, soit quatre octets, PostgreSQL a ajouté 8 Ko dans ce fichier. C'est tout à fait normal. PostgreSQL gère les fichiers de données de cette façon. Il travaille par bloc de 8 Ko. Tant qu'il a de la place dans le dernier bloc de 8 Ko, il utilise ce bloc. Dans le cas contraire, il ajoute un bloc de

8 Ko au fichier.

```
glmf=# INSERT INTO magazine (id) VALUES (105);
glmf=# \! ls -lF 16384/16387
-rw------- 1 guillaume guillaume 8192 2008-05-24 16:09 16384/16387
glmf=# INSERT INTO magazine (id) VALUES (106);
glmf=# \! ls -lF 16384/16387
-rw------- 1 guillaume guillaume 8192 2008-05-24 16:09 16384/16387
glmf=# INSERT INTO magazine (id) SELECT x FROM generate_series(1, 1000) AS x;
glmf=# \! ls -lF 16384/16387
-rw------- 1 guillaume guillaume 32768 2008-05-24 16:10 16384/16387
```

Nous pouvons en déduire qu'un fichier de données d'une base PostgreSQL, qu'il concerne une table ou un index, a toujours une taille qui est un multiple de 8 Ko. De plus, et c'est un point particulièrement important pour la gestion du cache disque, PostgreSQL lit et écrit des blocs de 8 Ko. Il ne lit jamais moins.

Revenons à la question du cache disque.

La taille du cache disque de PostgreSQL dépend du paramètre shared_buffers. Lors de l'exécution d'initdb, PostgreSQL tente de découvrir une bonne valeur pour ce paramètre en testant plusieurs valeurs. Cependant, la valeur maximum qu'initdb peut donner à ce paramètre est 32 Mo (en fait, 4096 blocs de 8 Ko). Cette valeur est très basse étant donné les quantités de mémoire habituellement disponible sur les serveurs d'aujourd'hui. Il est fréquent de voir des valeurs entre 512 Mo et quelques Go. Autrement dit au minimum vingt fois plus. La règle actuelle est de configurer ce paramètre avec un quart de la mémoire disponible sur un serveur dédié à PostgreSQL. À partir de là, seuls des tests empiriques pourront nous aider à trouver la valeur optimale pour notre serveur.

Le coup des blocs de 8 Ko explique pourquoi les fichiers de configuration des versions antérieures à la 8.2 demandaient pour shared_buffers, non pas une taille mémoire, mais un nombre de blocs de 8 Ko (ce qui revient au même en fin de compte, mais la taille mémoire est plus compréhensible que le nombre de blocs).

On pourrait se demander pourquoi ne pas donner toute la mémoire à PostgreSQL ? Tout simplement parce que le noyau sait beaucoup mieux gérer la mémoire que PostgreSQL. Il dispose de meilleurs algorithmes, peut prendre en considération le type de matériel utilisé, etc.

Mais alors, dans ce cas, pourquoi PostgreSQL ne se fie pas seulement au cache disque du système d'exploitation ? Pour deux raisons. La première, c'est que PostgreSQL sait mieux gérer son cache dans certains cas très précis liés à son domaine d'expertise. L'opération de maintenance VACUUM ou un parcours séquentiel complet d'une table peut facilement invalider un cache disque (celui de PostgreSQL, comme celui du noyau). Le moteur de PostgreSQL n'invalide qu'un pourcentage donné de son cache dans ce type de cas. Cela ralentit un peu le processus en charge du VACUUM, mais n'impacte pas les autres processus. Le système d'exploitation n'a aucune idée de l'opération en cours et pourrait invalider une grosse partie, voire tout son cache, ce qui est contre-productif. Il est donc nécessaire que les deux caches soient présents, avec une taille plus importante pour celui du système d'exploitation, mais avec une taille importante quand même pour celui de PostgreSQL.

Pour mieux comprendre l'utilisation du cache par PostgreSQL, nous allons utiliser le module contrib pg_buffercache.

Note : un module contrib est un outil qui se trouve dans les sources des versions stables de PostgreSQL mais dont l'intérêt n'a pas été prouvé pour un grand nombre d'utilisateurs et qui n'a donc pas été inclus dans le moteur pour cette raison. Ils sont de tout type : outils pour les développeurs, commandes spéciales pour les administrateurs, nouveau type de données, nouvelles procédures stockées, …

Une fois les modules contrib installés pour votre version, il vous faut trouver le fichier pg_buffercache.sql et le faire exécuter par psql de cette façon :

```
guillaume@laptop:/opt/mon_cluster/base$ cd
guillaume@laptop:~$ psql -f /chemin/vers/contrib/pg_buffercache.sql glmf
SET
CREATE FUNCTION
CREATE VIEW
REVOKE
REVOKE
```

Aucune erreur, nous allons pouvoir utiliser la vue et la procédure stockée que ce module a ajouté. Commençons par arrêter puis redémarrer le serveur (ceci pour vider le cache PostgreSQL précédent) :

```
guillaume@laptop:~$ pg_ctl -D /opt/mon_cluster restart
en attente de l'arrêt du serveur.... effectué
serveur arrêté
serveur en cours de démarrage
guillaume@laptop:~$ psql -q glmf
glmf=# \x
glmf=# SELECT * FROM pg_buffercache LIMIT 3;
-[ RECORD 1 ]--+------
bufferid       | 1
relfilenode    | 1262
reltablespace  | 1664
reldatabase    | 0
relblocknumber | 0
isdirty        | f
usagecount     | 5
-[ RECORD 2 ]--+------
bufferid       | 2
relfilenode    | 1260
reltablespace  | 1664
reldatabase    | 0
relblocknumber | 0
isdirty        | f
usagecount     | 3
-[ RECORD 3 ]--+------
bufferid       | 3
relfilenode    | 1259
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 0
isdirty        | f
usagecount     | 5
```

La vue pg_buffercache renvoie les colonnes suivantes :

- bufferid est le numéro du tampon dans le cache disque ;
- relfilenode est l'identifiant de la relation (une relation étant une table ou un index… un objet physique en fait) ;
- reltablespace est l'identifiant du tablespace ;
- reldatabase est l'identifiant de la base de données (car le cache disque est global) ;
- relblocknumber est le numéro du bloc disque dans la table
- isdirty indique si les données du bloc ont été modifiées (en mémoire) depuis son chargement ;
- usagecount précise le nombre de processus qui utilisent ce bloc.

Il existe un tampon pour chaque bloc de 8 Ko contenu dans l'espace mémoire allouée pour le cache disque. La vue les renvoie tous, qu'ils soient réellement utilisés ou non.

```
glmf=# \x
glmf=# SELECT count(*) FROM pg_buffercache;
 count
-------
  3072
(1 ligne)

glmf=# SHOW shared_buffers;
 shared_buffers
----------------
 24MB
(1 ligne)

glmf=# SELECT 24*1024/8;
 ?column?
----------
     3072
(1 ligne)
```

Nous avons bien 3072 tampons de 8 Ko vus par pg_buffercache.

La colonne relfilenode n'est renseignée que si le tampon correspondant est utilisé par un bloc disque. Donc, si nous voulons connaître le taux d'occupation du cache disque, voici comment faire :

```
glmf=# SELECT count(*) * 100 / 3072 AS "% utilisé du cache"
glmf-# FROM pg_buffercache WHERE relfilenode IS NOT NULL;
 % utilisé du cache
--------------------
                  9
(1 ligne)
```

Actuellement, 9 % du cache est utilisé. C'est tout à fait logique, nous avons démarré le serveur et effectué peu de requêtes. Il doit donc y avoir surtout des catalogues systèmes en cache. Voyons cela :

```
glmf=# SELECT rel.relname AS "Relation",
glmf-# count(*) AS "Nb de tampons dans le cache"
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode
glmf-# GROUP BY rel.relname
glmf-# ORDER BY count(*) DESC;
            Relation             | Nb de tampons dans le cache
---------------------------------+-----------------------------
 pg_attribute                    |                          32
 pg_proc                         |                          28
 pg_proc_oid_index               |                          13
 pg_operator                     |                          13
 pg_proc_proname_args_nsp_index  |                          13
 pg_class                        |                          12
 pg_operator_oprname_l_r_n_index |                          12
 pg_attribute_relid_attnum_index |                          12
 pg_rewrite                      |                          11
 pg_statistic                    |                           7
[...]
```

En effet, nous n'y voyons que des catalogues systèmes. Créons une nouvelle table :

```
glmf=# CREATE TABLE magazine2 (id integer);
glmf=# SELECT relname, relfilenode FROM pg_class WHERE relname='magazine2';
  relname  | relfilenode
-----------+-------------
 magazine2 |       16395
(1 ligne)

glmf=# \x
glmf=# SELECT buf.*
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode AND rel.relname='magazine2';
(Aucune ligne)
glmf=# \cd /opt/mon_cluster/base/16384
glmf=# \! ls -l 16395
-rw------- 1 guillaume guillaume 0 2008-05-24 16:16 16395
```

La table est créée, elle est vide, et le cache ne contient aucune donnée de cette table. Insérons maintenant une donnée :

```
glmf=# INSERT INTO magazine2 (id) VALUES (104);
glmf=# SELECT buf.*
```

```
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode AND rel.relname='magazine2';
-[ RECORD 1 ]--+------
bufferid       | 254
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 0
isdirty        | t
usagecount     | 1
```

Et là voilà ! Voici ce qu'il s'est passé. La requête INSERT a demandé l'ajout de la donnée dans la table. Cette table était vide, un bloc de 8 Ko est ajouté au fichier /opt/mon_cluster/base/16384/16394 :

```
glmf=# \! ls -l 16395
-rw------- 1 guillaume guillaume 8192 2008-05-24 16:20 16395
```

Ce bloc est ensuite chargé en mémoire dans le premier tampon libre du cache (ici, le 254 d'après la colonne bufferid). Ce tampon est modifié pour contenir la nouvelle valeur. Comme ce bloc est modifié en mémoire, la colonne isdirty vaut true. Seul notre processus a utilisé ce tampon d'où la valeur de 1 pour usagecount. Ce tampon contient le premier bloc du fichier (relblocknumber à 0… la numérotation commençant à zéro).

Au bout d'un certain temps (dépendant du paramètre checkpoint_timeout) ou après un événement (checkpoint_segments journaux de transaction écrits sans CHECKPOINT, ou exécution d'un CHECKPOINT manuel, ou enfin l'arrêt du serveur), le cache disque est parcouru par un processus appelé bgwriter. Tous les tampons modifiés en mémoire sont écrits sur disque (sauf dans certains cas, par exemple si bgwriter_lru_maxpages est dépassé). Bref, après un CHECKPOINT, le tampon de notre nouvelle table a cette tête :

```
glmf=# SELECT buf.*
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode AND rel.relname='magazine2';
-[ RECORD 1 ]--+------
bufferid       | 254
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 0
isdirty        | f
usagecount     | 1
```

Si nous insérons une nouvelle ligne, PostgreSQL n'a pas besoin de lire ce bloc sur disque. Il utilise son cache et le modifie directement :

```
glmf=# INSERT INTO magazine2 (id) VALUES (105);
glmf=# SELECT buf.*
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode AND rel.relname='magazine2';
-[ RECORD 1 ]--+------
bufferid       | 254
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 0
isdirty        | t
usagecount     | 2
```

De nouveau, isdirty est true. Usagecount est passé à deux. Insérons beaucoup de données :

```
glmf=# INSERT INTO magazine2 (id) select x from generate_series(1, 1000) as x;
glmf=# SELECT buf.*
glmf-# FROM pg_buffercache AS buf, pg_class AS rel
glmf-# WHERE buf.relfilenode=rel.relfilenode AND rel.relname='magazine2';
-[ RECORD 1 ]--+------
bufferid       | 254
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 0
isdirty        | t
usagecount     | 5
-[ RECORD 2 ]--+------
bufferid       | 257
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 1
isdirty        | t
usagecount     | 5
-[ RECORD 3 ]--+------
bufferid       | 258
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 2
isdirty        | t
usagecount     | 5
-[ RECORD 4 ]--+------
bufferid       | 259
relfilenode    | 16395
reltablespace  | 1663
reldatabase    | 16384
relblocknumber | 3
isdirty        | t
usagecount     | 5
```

Nous avons maintenant quatre tampons de 8 Ko. Le fichier doit donc faire 32 Ko :

```
glmf=# \! ls -l 16395
-rw------- 1 guillaume guillaume 32768 2008-05-24 16:23 16395
```

Le cache se remplit petit à petit au fur et à mesure de la lecture des pages disque. La dernière question est de savoir ce qu'il se passe une fois que ce cache est plein. C'est un problème très connu en informatique, et un grand nombre d'algorithmes est disponible, chacun tentant de trouver la meilleure approche pour résoudre ce problème. PostgreSQL a une solution assez particulière. Quand une page disque est chargé dans un tampon du cache, ce tampon voit son compteur d'utilisation (colonne usagecount) passé à 1. Chaque processus qui utilise ce tampon incrémente automatiquement le usagecount du tampon de 1, avec malgré tout une valeur maximum de 5 (valeur configurable uniquement à la compilation). Chaque fois qu'un processus, postgres ou bgwriter, cherche une place pour une page disque, si un tampon n'est pas sélectionnable, son usagecount est décrémenté. Arrivé à 0, le tampon est réutilisable. Autrement dit, une page disque très populaire résiste à cinq recherches avant que son tampon ne soit invalidé. Si le tampon sélectionné est modifié, il faut dans un premier temps l'écrire sur disque. Dans ce cas, ce n'est pas bgwriter qui se charge de l'écriture, mais le processus postgres lui-même. Cela signifie que, même dans le cas d'une requête SELECT, le processus postgres en charge de l'exécution de cette requête peut être amené à écrire des tampons du cache sur le disque. Évidemment, les performances de l'exécution de cette requête s'en trouvent fortement impactées (et pas dans le bon sens). Il est donc essentiel de savoir si bgwriter a bien le temps d'écrire les tampons modifiés du cache sur disque. Pour cela, vous disposez de la vue système pg_stat_bgwriter. La requête suivant nous donne les informations nécessaires :

```
glmf=# SELECT buffers_checkpoint, buffers_backend, buffers_alloc
glmf-# FROM pg_stat_bgwriter;
-[ RECORD 1 ]------+----
buffers_checkpoint | 44
buffers_backend    | 9
buffers_alloc      | 535
```

Sur 535 pages placées dans le cache, 53 (résultat de l'addition des deux premières colonnes) ont été modifiés. 44 ont été écrites par bgwriter, 9 par les processus postgres, ce qui est un ratio tout à fait correct. En effet, il est préférable que le pourcentage soit en faveur du bgwriter. Si c'est le contraire, vous pouvez influer sur le délai entre deux réveils de bgwriter grâce au paramètre bgwriter_delay (en millisecondes). bgwriter_lru_maxpages indique le nombre maximum de tampons écrits sur disque pour un seul tour. bgwriter_lru_multiplier (nouveau paramètre de la 8.3) permet d'affiner l'optimisation automatique du processus bgwriter. PostgreSQL tient le compte des tampons nécessaires pour quelques tours précédents de bgwriter. La moyenne de ces différents calculs, multiplié par bgwriter_lru_multiplier donne une indication sur le nombre de tampons nécessaires pour le prochain tour. bgwriter s'arrêtera donc une fois qu'il aura libéré ce nombre de tampons. Le côté très intelligent de cette façon de fonctionner, c'est que le processus n'écrit sur disque que le nombre de tampons qu'il estime nécessaire de libérer pour l'activité entre ses écritures et un prochain réveil de bgwriter. Le multiplicateur permet d'ajouter une marge de sécurité.

Il existe deux exceptions à la mise en cache des fichiers de données : les parcours séquentiels et les opérations VACUUM. Si vous réalisez un VACUUM ou si vous parcourez séquentiellement une table dont la taille est supérieure au quart de la taille du cache, le processus postgres en charge de l'exécution de la requête utilise un sous-ensemble strict du cache. Ce sous-ensemble est équivalent à la plus petite valeur entre deux : le huitième de la taille du cache ou 256 Ko. Dans la majorité (et le meilleur) des cas, ce sous-ensemble vaudra 256 Ko. Une fois ce sous-ensemble plein, si le parcours séquentiel a de nouveau besoin d'un tampon et que ce dernier n'a pas été utilisé par un autre processus, il est immédiatement invalidé et ré-utilisé.

Cache des journaux de transactions

En dehors des caches des objets servant à accélérer l'accès à leur données, il existe un autre cache disque qui sert uniquement aux journaux de transaction. Comme les journaux de transaction ne sont qu'écrits, le cache permet d'attendre la fin d'une transaction avant d'écrire l'ensemble des pages disques modifiées par la transaction terminée dans les journaux de transaction. Cela sous-entend que le cache est assez gros pour contenir la transaction la plus longue possible pour vos applications, et qu'il peut contenir les résultats intermédiaires de plusieurs transactions en même temps.

Dans le cas où le cache des journaux de transaction est rempli, le processus postgres (ou le processus « wal writer » en 8.3) écrit certains tampons du cache sur disque pour vider une partie du cache et y placer les tampons dont il a besoin.

Le dimensionnement de ce cache est assez complexe à quantifier. Néanmoins, la mémoire allouée par défaut (64 Ko) est faible. Une augmentation raisonnable ne consomme de toute façon pas énormément de mémoire, il ne faut donc pas se priver des avantages que cela amène. Pour les machines multi-processeurs (donc les machines qui peuvent réellement exécuter plusieurs instructions SQL en même temps), une valeur de 1 Mo, voire quelques Mo, est raisonnable. Cela donne une idée de l'étendue possible des modifications : il est inhabituel de voir une valeur supérieure à une dizaine de Mo (Tom Lane indique même dans un mail de 2007 que personne n'a apporté la preuve qu'une valeur supérieure à 1 Mo soit intéressante).

Il est aussi important de prendre en considération le fait que, plus ce cache est grand, plus les pertes de transaction sont importantes si vous désactivez synchronous_commit (ie synchronous_commit = off) et que le serveur plante brutalement.

Carte des espaces libres

Le cache disque n'est pas la seule structure globale en mémoire. Il existe aussi une structure assez peu connu mais toute aussi importante pour garantir un bon état des fichiers de données (donc tables et index). Il s'agit de la structure FSM, acronyme de « Free Space Map » (qu'on pourrait traduire en « Carte des Espaces Libres »).

Cette structure est remplie par le VACUUM. Ce dernier y stocke tous les espaces ré-utilisables des différentes tables et index du serveur. Ainsi, lorsqu'un processus postgres a besoin de stocker une certaine quantité de données pour une table, il regarde en premier lieu les espaces libres connues pour cette table dans la structure FSM. S'il trouve un espace libre suffisamment grand, il l'utilise. Dans le cas contraire, il enregistre ses données à la fin de la table, la faisant automatiquement grossir. Sans VACUUM et la structure FSM liée, les tables grossiraient éternellement et seraient de plus en plus fragmentées.

Mais que se passe-t'il si cette structure n'est pas assez grosse pour détailler tous les espace libres disponibles ? L'élément le moins intéressant est remplacé par le nouvel élément. Dis autrement, PostgreSQL va oublier des espaces libres, ce qui fait là-aussi automatiquement grossir les tables et index.

Il est donc essentiel de configurer la taille de la structure de façon adéquate. Pour cela, nous disposons de deux paramètres de configuration : max_fsm_pages et max_fsm_relations.

Commençons par le deuxième. Il précise le nombre de relations qui seront tracées. Le nombre de tables et d'index doit être inférieur à max_fsm_relations. Par défaut, ce dernier vaut 1000, ce qui doit convenir dans la majorité des cas.

max_fsm_pages représente le nombre de pages disques qui seront tracées. Il est nécessaire d'avoir au moins autant de pages tracées que de pages libres. Malheureusement, il est très complexe de savoir combien de pages sont réellement libres. De plus, ce nombre est en changement constant suivant

l'activité du serveur. Il existe donc plusieurs moyens permettant d'avoir une estimation de ce nombre de pages.

L'exécution d'un « VACUUM VERBOSE » donnera une information sur le nombre de pages disques nécessaires pour tracer les espaces libres pour la base de données sur laquelle a été exécuté le VACUUM VERBOSE. Il faut donc le lancer sur toutes les bases et additionner les résultats obtenus pour une valeur minimale.

Les deux autres moyens découlent d'une même idée. Plutôt que d'indiquer le nombre de pages vides, indiquons le nombre de pages total. Voici ces deux moyens :

- exécution de « SELECT sum(relpages) FROM pg_class WHERE relkind IN ('r', 't', 'i') ; » sur chaque base de données et addition des résultats obtenus ;
- du -sh du répertoire $PGDATA/base et division par 8 Ko pour obtenir le nombre de pages.

Ces deux dernières propositions semblent augmenter considérablement et inconsidérément le nombre de pages tracées. C'est exact d'une certaine façon. Cependant, chaque page disque surveillée n'occupe que six octets en mémoire, il serait donc dommage de viser un nombre très petit au risque, avec une base dont l'activité normale fait qu'elle grossit, de se retrouver avec des tables dont la taille explose.

Mémoire par processus

Suivant les objets utilisés et les opérations effectuées, un processus peut avoir besoin d'un surplus de mémoire. La configuration est d'autant plus délicate que chaque processus pourrait demander cette mémoire en même temps. Dis autrement, nous pouvons avoir un nombre maximum de processus (limite assurée par le paramètre max_connections) qui demandent cette mémoire. Dans certains cas, cela peut atteindre des valeurs énormes, dépassant la quantité de mémoire physique réellement disponible.

Cache des objets temporaires

Il existe un cache disque spécifique pour les tables et index temporaires. Ces derniers étant par nature locaux à la session et surtout rapidement obsolètes, ils disposent d'un cache propre dont la taille est bien inférieure à celles des objets permanents. Si votre application se repose essentiellement sur l'utilisation de tables temporaires, augmenter la valeur de ce paramètre peut vous apporter un gain en performance très appréciable. Néanmoins, comme le cache est local à la session, il est alloué par chaque processus qui en a besoin. Tout n'est pas alloué d'un coup, seul l'espace nécessaire est pris sur la mémoire jusqu'à arriver à la limite que représente temp_buffers.

Dans le cas où les objets temporaires remplissent complètement le cache de la session, des écritures sur disque auront lieu, ce qui impactera les performances.

Exécution d'une requête

Même si les cache disques de PostgreSQL constituent la grosse partie de la mémoire utilisée par PostgreSQL, ce ne sont pas les seuls moyens d'influer sur la mémoire utilisée. L'exécution d'une requête peut nécessiter un besoin supplémentaire de mémoire. Pour cela, le paramètre work_mem permet de contrôler la mémoire maximum utilisée avant de passer à l'utilisation de fichiers temporaires sur disque. Il s'agit principalement d'une mémoire utilisée pour des tris et du hachage de données. Cette mémoire est allouée par chaque processus qui en a besoin. Il faut donc éviter de mettre une valeur trop grosse car chaque processus peut l'utiliser. Si tous vos processus l'utilisent, le serveur peut se voir contraint à utiliser le swap et les performances s'en ressentiront. Mais si cette valeur est trop basse, les processus postgres doivent stocker les résultats intermédiaires sur disque dès que la mémoire de tri est remplie.

Donc, le but est d'éviter l'utilisation de fichiers temporaires sur disque tout en évitant que le serveur utilise de la swap. Voici un exemple de session montrant le problème. Nous commençons par créer une nouvelle table :

```
guillaume@laptop:~$ psql -q glmf
glmf=# CREATE TABLE magazine3 (id integer);
CREATE TABLE
```

Puis nous insérons un million d'entiers :

```
glmf=# INSERT INTO magazine3 (id)
glmf-# SELECT x FROM generate_series(1, 1000000) AS x;
INSERT 0 1000000
```

Enfin, nous activons la trace sur l'utilisation de fichiers temporaires (la valeur est la taille minimum des fichiers pour qu'une trace soit enregistrée ; avec 0, nous traçons tous les fichiers, quelque soit leur taille) :

```
glmf=# SET log_temp_files TO 0;
SET
```

Note : cette trace n'est disponible qu'à partir de la version 8.3.

Commençons avec un work_mem à 1 Mo :

```
glmf=# SET work_mem TO '1MB';
SET
glmf=# SELECT * FROM magazine3 ORDER BY id LIMIT 2;
 id
----
  1
  2
(2 lignes)
```

Nous avons limité la sortie à deux éléments. Le tri est rapide, PostgreSQL n'a pas besoin de trop de mémoire, donc ça tient facilement sur 1 Mo de work_mem. Récupérons toutes les lignes de cette table :

```
glmf=# SELECT * FROM magazine3 ORDER BY id;
LOG:  fichier temporaire : chemin « base/pgsql_tmp/pgsql_tmp6306.6 », taille 16007168
INSTRUCTION :  SELECT * FROM magazine3 ORDER BY id ;
   id
---------
       1
```

```
       2
[...]
```

Cette fois, PostgreSQL a dû créer un fichier temporaire de 16 Mo pour y stocker un résultat temporaire. Augmentons work_mem et ré-exécutons la même requête :

```
glmf=# SET work_mem TO '40MB';
SET
glmf=# SELECT * FROM magazine3 ORDER BY id ;
   id
---------
       1
       2
[...]
```

Cette fois, aucun fichier temporaire n'a été créé.

En évitant la création de fichiers sur le disque, nous améliorons les performances. Il est donc essentiel d'avoir une valeur de work_mem adéquate. Cela dépend principalement des requêtes que vous exécutez, si vous utilisez beaucoup de tris ou si vous avez des requêtes très grosses. Sachez de toute façon qu'il est tout à fait possible de configurer une valeur faible globale au serveur pour éviter l'utilisation du swap par le système d'exploitation et de faire en sorte que les clients ayant besoin de réaliser un gros tri configurent work_mem localement à la session avec l'instruction SET.

Opérations de maintenance

Pour certaines opérations de maintenance comme le VACUUM ou pour des modifications de schéma (spécifiquement pour la création d'index et l'ajout de clés étrangères), PostgreSQL alloue dynamiquement une portion de mémoire. La quantité utilisée est paramétrable avec la variable maintenance_work_mem.

Voici un exemple sur la création d'un index :

```
glmf=# \timing
Timing is on.
glmf=# SET client_min_messages TO log;
SET
Time: 0,474 ms
glmf=# SET log_temp_files TO 0;
SET
Time: 0,319 ms
glmf=# CREATE TABLE magazine4 (id integer);
CREATE TABLE
Time: 86,779 ms
glmf=# INSERT INTO magazine4 SELECT x FROM generate_series(1, 5000000) AS x;
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp8586.0", size 100000000
INSERT 0 5000000
Time: 29400,030 ms
glmf=# SET maintenance_work_mem TO 1024;
SET
Time: 0,327 ms
glmf=# CREATE INDEX i1 ON magazine4(id);
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp8586.1", size 80044032
CREATE INDEX
Time: 22006,237 ms
glmf=# DROP INDEX i1;
DROP INDEX
Time: 85,927 ms
glmf=# SET maintenance_work_mem TO 1000000;
SET
Time: 0,323 ms
glmf=# CREATE INDEX i1 ON magazine4(id);
CREATE INDEX
Time: 9919,892 ms
glmf=# DROP INDEX i1;
DROP INDEX
Time: 50,755 ms
```

Non seulement PostgreSQL n'écrit plus sur disque ce fichier de 80 Mo, mais en plus il est deux fois plus rapide (dans cet exemple). Il ne faut pas hésiter à allouer une grosse quantité de mémoire et ce, pour deux raisons. La plus importante est que cela améliore considérablement les performances de ces trois opérations. La seconde est que le risque d'un manque d'espace mémoire est faible car peu de personnes vont, en même temps, modifier le schéma, exécuter une opération de VACUUM et/ou créer un index. Il est en plus possible de diminuer ce risque en configurant par défaut une valeur raisonnable (par exemple 64 à 128 Mo), et créer un utilisateur qui dispose par défaut d'une configuration bien plus importante (par exemple 512 Mo). Voici un exemple pour gérer ce dernier cas.

Par défaut, la configuration est de 64 Mo dans postgresql.conf.

```
guillaume@laptop:/opt/mon_cluster$ grep maintenance_work_mem postgresql.conf
maintenance_work_mem = 64MB             # min 1MB
guillaume@laptop:/opt/mon_cluster$ psql -q postgres
postgres=# SHOW maintenance_work_mem;
 maintenance_work_mem
----------------------
 64MB
(1 ligne)
```

Créons maintenant un utilisateur administrateur et ajoutons-lui une configuration spécifique sur le paramètre maintenance_work_mem :

```
postgres=# CREATE USER administrateur;
postgres=# ALTER USER administrateur SET maintenance_work_mem TO '512MB';
```

Connectons-nous en tant qu'utilisateur guillaume et vérifions la quantité de mémoire allouée aux opérations de maintenance :

```
guillaume@laptop:/opt/mon_cluster$ psql -q -U guillaume postgres
postgres=# SELECT current_user;
 current_user
--------------
 guillaume
(1 ligne)

postgres=# SHOW maintenance_work_mem;
 maintenance_work_mem
----------------------
 64MB
(1 ligne)
```

L'utilisateur guillaume dispose bien de 64 Mo pour maintenance_work_mem. Connectons-nous maintenant en tant qu'utilisateur administrateur :

```
guillaume@laptop:/opt/mon_cluster$ psql -q -U administrateur postgres
postgres=> SELECT current_user;
  current_user
----------------
 administrateur
(1 ligne)

postgres=> SHOW maintenance_work_mem;
 maintenance_work_mem
----------------------
 512MB
(1 ligne)
```

Attention toutefois à ne pas ajouter trop de mémoire. Comme la mémoire est allouée au lancement de la requête, cette dernière pourrait échouer si la mémoire est insuffisante sous certains systèmes (les BSD notamment). Du coup, même les VACUUM du processus autovacuum pourraient échouer sans que vous puissiez vous en rendre compte (en dehors d'une lecture attentive des journaux applicatifs).

Taille de la pile

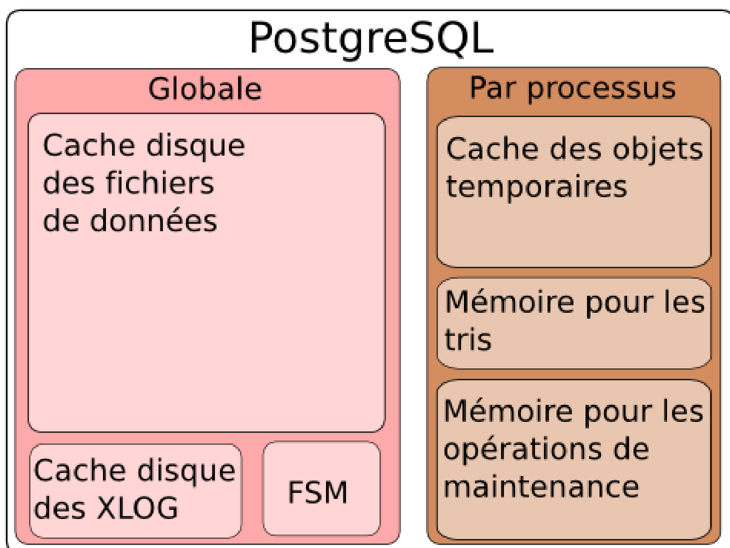
max_stack_depth permet de configurer la taille de la pile d'exécution du serveur. Une configuration logique revient à indiquer une valeur légèrement inférieure à celle du noyau :

```
guillaume@laptop:~$ ulimit -s
8192
guillaume@laptop:/opt/mon_cluster$ grep max_stack_depth postgresql.conf
#max_stack_depth = 2MB                  # min 100kB
```

Nous sommes sur la valeur de 2 Mo (valeur par défaut) et nous pourrions l'augmenter jusqu'à 6 à 7 Mo. L'augmenter à une valeur supérieure à celle du noyau nous exposerait au risque qu'un processus serveur plante suite à l'exécution d'une fonction récursive par exemple. L'intérêt d'augmenter cette valeur est donc de permettre l'exécution de fonctions complexes.

Résumé graphique

Voici un graphique résumant les différentes allocations mémoires, qu'elle soit globale au serveur ou local au processus.



La taille de chaque bloc donne une idée (très) approximative de la répartition de la mémoire.

Conclusion

Nous avons vu qu'il est possible de configurer un grand nombre de paramètres jouant sur la mémoire gérée par PostgreSQL. Pour cette configuration, le point de contrôle le plus important à prendre en compte concerne le cache des fichiers de données, gérée par le paramètre shared_buffers. Cependant, dans des cas d'utilisation très précis, certains autres paramètres sont décisifs :

- vos applications utilisent beaucoup d'objets temporaires ? Augmenter la valeur du paramètre temp_buffers ;
- vos transactions sont particulièrement longues ou vous avez de nombreux clients connectés en même temps ? Pensez à accroître wal_buffers ;

- vous faites de nombreux tris ? Augmentez work_mem avec finesse (la finesse dépendant principalement du nombre de clients connectés en même temps) ;
- vos créations d'index ou vos opérations VACUUM sont particulièrement longues ? Testez des valeurs plus importantes de maintenance_work_mem.

Liens/Références

- Configuration du serveur, http://docs.postgresqlfr.org/8.3/runtime-config.html
- Buffer Cache, Checkpoints, and the BGW, http://developer.postgresql.org/index.php/Buffer_Cache%2C_Checkpoints%2C_and_the_BGW

© Copyright 2005-2019 Dalibo