Module M3

Mémoire & journalisation

22.09

Dalibo SCOP

https://dalibo.com/formations

Mémoire & journalisation

Module M3

TITRE : Mémoire & journalisation
SOUS-TITRE : Module M3

REVISION: 22.09
DATE: 02 septembre 2022

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

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

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com !

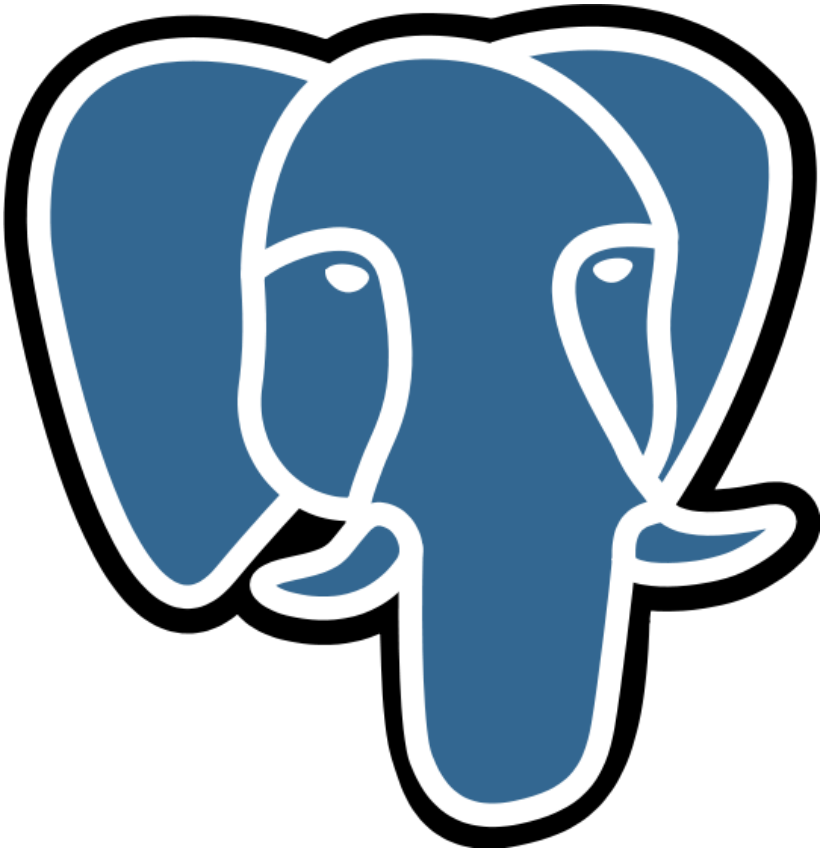
Table des Matières

Licence Creative Commons BY-NC-SA 2.0 FR **5**

1 Mémoire et journalisation dans PostgreSQL **10**

1.1 Au menu 10
1.2 Mémoire partagée 11
1.3 Mémoire par processus 12
1.4 Shared buffers 13
1.5 Journalisation 20
1.6 Au-delà de la journalisation 27
1.7 Conclusion 29
1.8 Quiz 29
1.9 Travaux pratiques 30
1.10 Travaux pratiques (solutions) 34

1 MÉMOIRE ET JOURNALISATION DANS POSTGRESQL



1.1 AU MENU

La mémoire & PostgreSQL :

- mémoire partagée
- mémoire des processus
- les *shared buffers* & la gestion du cache
- la journalisation

1.2 MÉMOIRE PARTAGÉE

- Implémentation
  - `shared_memory_type`
- Zone de mémoire partagée :
  - cache disque des fichiers de données (`shared_buffers`)
  - cache disque des journaux de transactions (`wal_buffers`)
  - données de session (`max_connections` et `track_activity_query_size`)
  - verrous (`max_connections` et `max_locks_per_transaction`)
    - etc

La zone de mémoire partagée est allouée statiquement au démarrage de l'instance. Depuis la version 12, le type de mémoire partagée est configuré avec le paramètre `shared_memory_type`. Sous Linux, il s'agit par défaut de `mmap`, sachant qu'une très petite partie utilise toujours `sysv` (System V). Il est possible de basculer uniquement en `sysv` mais ceci n'est pas recommandé et nécessite généralement un paramétrage du noyau Linux. Sous Windows, le type est `windows`. Avant la version 12, ce paramètre n'existe pas.

Elle est calculée en fonction du dimensionnement des différentes zones :

- `shared_buffers` : le cache des fichiers de données ;
- `wal_buffers` : le cache des journaux de transaction ;
- les données de sessions : les principaux paramètres liés sont `max_connections` (défaut : 100) et `track_activity_query_size` (défaut : 1024) ;
- les verrous : les paramètres sont `max_connections` et `max_locks_per_transaction` (défaut : 64).

Toute modification des paramètres régissant la mémoire partagée imposent un redémarrage de l'instance.

Nous verrons en détail l'utilité de certaines de ces zones dans les chapitres suivants.

1.3 MÉMOIRE PAR PROCESSUS

- `work_mem`
  - × `hash_mem_multiplier` (v 13)
- `maintenance_work_mem`
  - `autovacuum_work_mem`
- `temp_buffers`
- Pas de limite stricte à la consommation mémoire d'une session !
  - ni total
- Augmenter prudemment & superviser

Chaque processus, en plus de la mémoire partagée à laquelle il accède en permanence, peut allouer de la mémoire pour ses besoins propres. L'allocation de cette mémoire est temporaire : elle est libérée dès qu'elle n'est plus utile, en fin de requête ou de session. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Cette mémoire est utilisée dans plusieurs contextes :

Tris, hachages, jointures

De la mémoire de tri peut être consommée lors de l'exécution de requêtes avec une clause `ORDER BY` ou certaines jointures, comme une jointure par hachage. Ce tri sera effectué en mémoire, à hauteur de la valeur de `work_mem` (seulement 4 Mo par défaut), potentiellement pour **chaque** nœud concerné. S'il estime que cette mémoire ne suffira pas, PostgreSQL triera les données sur disque, avec écriture de fichiers temporaires, ce qui peut notablement ralentir la requête.

Augmenter la valeur du `work_mem` de manière globale peut parfois mener à une consommation excessive de mémoire. PostgreSQL 13 ajoute un paramètre `hash_mem_multiplier`, par défaut à 1, qui permet de multiplier d'autant la consommation mémoire uniquement pour les hachages (jointures ou certains agrégats).

Maintenance

Les ordres `CREATE INDEX`, `REINDEX` ou `VACUUM` nécessitent aussi de la mémoire de tri. Ces besoins assez ponctuels, mais gourmands, sont gérés par le paramètre `maintenance_work_mem`, habituellement configuré à des valeurs plus hautes que `work_mem`. Sur une machine bien dotée en RAM, 1 Go est une valeur courante.

Tables temporaires

Afin de minimiser les appels systèmes dans le cas d'accès à des tables et index temporaires (locales à chaque session, et qui disparaîtront avec elle), chaque session peut allouer un cache dédié à ces tables. Sa taille dépend du paramètre `temp_buffers`. La valeur par

défaut de 8 Mo peut être insuffisant dans certains cas, ce qui peut mener à la création de fichiers sur disque dans le répertoire de la base de données.

Il n'y a pas de limite globale à la mémoire pouvant être utilisée par ces paramètres !
Il est théoriquement possible que toutes les connexions (au nombre de `max_connections`) lancent simultanément des requêtes allouant plusieurs fois `work_mem` (si la requête en cours d'exécution nécessite plusieurs tris par exemple, ou si d'autres processus sont appelés à l'aide, notamment en cas de parallélisation de la requête). Certains plans d'exécution malheureux peuvent consommer ainsi beaucoup plus que prévu.
Il faut donc rester prudent sur les valeurs de ces paramètres, `work_mem` tout particulièrement, et superviser les conséquences d'une modification de celui-ci.

Une consommation mémoire excessive peut mener à une purge du cache de l'OS ou un *swap* excessif, tous deux désastreux pour les performances, voire un arrêt de l'instance si l'*Out-of-Memory killer* de Linux décide de tuer des processus. Au niveau système, ce phénomène peut être mitigé par le paramétrage de l'*overcommit*[2] .

1.4 SHARED BUFFERS

- *Shared buffers* ou blocs de mémoire partagée
  - partage les blocs entre les processus
  - cache en lecture ET écriture
  - double emploi partiel avec le cache du système (voir `effective_cache_size`)
  - importants pour les performances !
- Dimensionnement en première intention :
  - 1/4 RAM
  - max 8 Go

PostgreSQL dispose de son propre mécanisme de cache. Toute donnée lue l'est de ce cache. Si la donnée n'est pas dans le cache, le processus devant effectuer cette lecture l'y recopie avant d'y accéder dans le cache.

L'unité de travail du cache est le bloc (de 8 ko par défaut) de données. C'est-à-dire qu'un processus charge toujours un bloc dans son entier quand il veut lire un enregistrement. Chaque bloc du cache correspond donc exactement à un bloc d'un fichier d'un objet. Cette information est d'ailleurs, bien sûr, stockée en en-tête du bloc de cache.

Tous les processus accèdent à ce cache unique. C'est la zone la plus importante, par la taille, de la mémoire partagée. Toute modification de données est tracée dans le journal de transaction, **puis** modifiée dans ce cache. Elle n'est donc pas écrite sur le disque par

[2] https://dali.bo/j1_html#configuration-du-oom

le processus effectuant la modification, sauf en dernière extrémité (voir Synchronisation en arrière plan.

Tout accès à un bloc nécessite la prise de verrous. Un *pin lock*, qui est un simple compteur, indique qu'un processus se sert du buffer, et qu'il n'est donc pas réutilisable. C'est un verrou potentiellement de longue durée. Il existe de nombreux autres verrous, de plus courte durée, pour obtenir le droit de modifier le contenu d'un buffer, d'un enregistrement dans un buffer, le droit de recycler un buffer… mais tous ces verrous n'apparaissent pas dans la table `pg_locks`, car ils sont soit de très courte durée, soit partagés (comme le *spin lock*). Il est donc très rare qu'ils soient sources de contention, mais le diagnostic d'une contention à ce niveau est difficile.

Les lectures et écritures de PostgreSQL passent toutefois toujours par le cache du système. Les deux caches risquent donc de stocker les mêmes informations. Les algorithmes d'éviction sont différents entre le système et PostgreSQL, PostgreSQL disposant de davantage d'informations sur l'utilisation des données, et le type d'accès qui y est fait. La redondance est donc habituellement limitée.

Dimensionner correctement ce cache est important pour de nombreuses raisons.

Un cache trop petit :

- ralentit l'accès aux données, car des données importantes risquent de ne plus s'y trouver ;
- force l'écriture de données sur le disque, ralentissant les sessions qui auraient pu effectuer uniquement des opérations en mémoire ;
- limite le regroupement d'écritures, dans le cas où un bloc viendrait à être modifié plusieurs fois.

Un cache trop grand :

- limite l'efficacité du cache système en augmentant la redondance de données entre les deux caches ;
- peut ralentir PostgreSQL, car la gestion des `shared_buffers` a un coût de traitement ;
- réduit la mémoire disponible pour d'autres opérations (tris en mémoire notamment).

Ce paramétrage du cache est malgré tout moins critique que sur de nombreux autres SGBD : le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est donc préférable de sous-dimensionner `shared_buffers` que de le sur-dimensionner.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle[a] donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine de 32 Go de RAM, cela donne donc :

```
shared_buffers=8GB
```

Le défaut de 128 Mo n'est pas adapté à un serveur sur une machine récente.

À cause du coût de la gestion de cette mémoire, surtout avec de nombreux processeurs ou de nombreux clients, une règle conservatrice peut être de ne pas dépasser 8 ou 10 Go, surtout sur les versions les moins récentes de PostgreSQL. Jusqu'en 9.6, sous Windows, il était même conseillé de ne pas dépasser 512 Mo.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres comme `max_wal_size`, et de s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…).

Un cache supplémentaire est disponible pour PostgreSQL : celui du système d'exploitation. Il est donc intéressant de préciser à PostgreSQL la taille approximative du cache, ou du moins de la part du cache qu'occupera PostgreSQL. Le paramètre `effective_cache_size` n'a pas besoin d'être très précis, mais il permet une meilleure estimation des coûts par le moteur. Il est paramétré habituellement aux alentours des 2/3 de la taille de la mémoire vive du système d'exploitation, pour un serveur dédié.

Par exemple pour une machine avec 32 Go de RAM, on peut paramétrer en première intention dans `postgresql.conf` :

```
shared_buffers = '8GB'
effective_cache_size = '21GB'
```

Cela sera à ajuster en fonction du comportement observé de l'application.

---

[a] https://www.postgresql.org/docs/current/runtime-config-resource.html

1.4.1 NOTIONS ESSENTIELLES DE GESTION DU CACHE

- Buffer pin
- Buffer dirty/clean
- Compteur d'utilisation
- Clocksweep

Les principales notions à connaître pour comprendre le mécanisme de gestion du cache de PostgreSQL sont :

Buffer pin

Chaque processus voulant accéder à un buffer (un bloc du cache) doit d'abord en forcer le maintien en cache (*to pin* signifie *épingler*). Chaque processus accédant à un buffer incrémente ce compteur, et le décrémente quand il a fini. Un buffer dont le pin est différent de 0 est donc utilisé et ne peut être recyclé.

Buffer dirty/clean

Un buffer est *dirty* (« sale ») si son contenu dans le cache ne correspond pas à son contenu sur disque : il a été modifié dans le cache, ce qui a généralement été journalisé, mais le fichier de données n'est plus à jour.

Au contraire, un buffer non modifié (*clean*) peut être supprimé du cache immédiatement pour faire de la place sans être réécrit sur le disque, ce qui est le moins coûteux.

Compteur d'utilisation

Cette technique vise à garder dans le cache les blocs les plus utilisés.

À chaque fois qu'un processus a fini de se servir d'un buffer (quand il enlève son pin), ce compteur est incrémenté (à hauteur de 5 dans l'implémentation actuelle). Il est décrémenté par le *clocksweep* évoqué plus bas.

Seul un buffer dont le compteur est à zéro peut voir son contenu remplacé par un nouveau bloc.

Clocksweep (ou algorithme de balayage)

Un processus ayant besoin de charger un bloc de données dans le cache doit trouver un buffer disponible. Soit il y a encore des buffers vides (cela arrive principalement au démarrage d'une instance), soit il faut libérer un buffer.

L'algorithme *clocksweep* parcourt la liste des buffers de façon cyclique à la recherche d'un buffer *unpinned* dont le compteur d'utilisation est à zéro. Tout buffer visité voit son compteur décrémenté de 1. Le système effectue autant de passes que nécessaire sur tous les

blocs jusqu'à trouver un buffer à 0. Ce *clocksweep* est effectué par chaque processus, au moment où ce dernier a besoin d'un nouveau buffer.

1.4.2 RING BUFFER

But : ne pas purger le cache à cause :

- des grandes tables
- de certaines opérations
  - *Seq Scan* (lecture)
  - `VACUUM` (écritures)
  - `COPY`, `CREATE TABLE AS SELECT…`
  - …

Une table peut être plus grosse que les *shared buffers*. Sa lecture intégrale (lors d'un parcours complet ou d'une opération de maintenance) ne doit pas mener à l'éviction de tous les blocs du cache.

PostgreSQL utilise donc plutôt un *ring buffer* quand la taille de la relation dépasse 1/4 de `shared_buffers`. Un *ring buffer* est une zone de mémoire gérée à l'écart des autres blocs du cache. Pour un parcours complet d'une table, cette zone est de 256 ko (taille choisie pour tenir dans un cache L2). Si un bloc y est modifié (`UPDATE`…), il est traité hors du *ring buffer* comme un bloc sale normal. Pour un `VACUUM`, la même technique est utilisée, mais les écritures se font dans le *ring buffer*. Pour les écritures en masse (notamment `COPY` ou `CREATE TABLE AS SELECT`), une technique similaire utilise un *ring buffer* de 16 Mo.

Le site The Internals of PostgreSQL[3] et un README[4] dans le code de PostgreSQL entrent plus en détail sur tous ces sujets tout en restant lisibles.

1.4.3 CONTENU DU CACHE

2 extensions en « contrib » :

- `pg_buffercache`
- `pg_prewarm`

Deux extensions sont livrées dans les *contribs* de PostgreSQL qui impactent le cache.

`pg_buffercache` permet de consulter le contenu du cache (à utiliser de manière très ponctuelle). La requête suivante indique les objets non système de la base en cours,

[3] https://www.interdb.jp/pg/pgsql08.html
[4] https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README

présents dans le cache et s'ils sont *dirty* ou pas :

```
pgbench=# CREATE EXTENSION pg_buffercache ;

pgbench=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

        relname        | isdirty | blocs | taille
-----------------------+---------+-------+---------
 pgbench_accounts      | f       |  8398 | 66 MB
 pgbench_accounts      | t       |  4622 | 36 MB
 pgbench_accounts_pkey | f       |  2744 | 21 MB
 pgbench_branches      | f       |    14 | 112 kB
 pgbench_branches      | t       |     2 | 16 kB
 pgbench_branches_pkey | f       |     2 | 16 kB
 pgbench_history       | f       |   267 | 2136 kB
 pgbench_history       | t       |   102 | 816 kB
 pgbench_tellers       | f       |    13 | 104 kB
 pgbench_tellers_pkey  | f       |     2 | 16 kB
```

L'extension `pg_prewarm` permet de précharger un objet dans le cache de PostgreSQL (s'il y tient, bien sûr) :

```
=# CREATE EXTENSION pg_prewarm ;
=# SELECT pg_prewarm ('nom_table_ou_index', 'buffer') ;
```

Il permet même de recharger dès le démarrage le contenu du cache lors d'un arrêt (voir la documentation[5]).

---

[5] https://docs.postgresql.fr/current/pgprewarm.html

1.4.4 SYNCHRONISATION EN ARRIÈRE PLAN

- Le *Background Writer* synchronise les buffers
  - de façon anticipée
  - une portion des pages à synchroniser
  - paramètres : `bgwriter_delay`, `bgwriter_lru_maxpages`, `bgwriter_lru_multiplier` et `bgwriter_flush_after`
- Le *checkpointer* synchronise les buffers
  - lors des checkpoints
  - synchronise toutes les dirty pages
- Écriture directe par les *backends*
  - dernière extrémité

Afin de limiter les attentes des sessions interactives, PostgreSQL dispose de deux processus, le *Background Writer* et le *Checkpointer*, tous deux essayant d'effectuer de façon asynchrone les écritures des buffers sur le disque. Le but est que les temps de traitement ressentis par les utilisateurs soient les plus courts possibles, et que les écritures soient lissées sur de plus grandes plages de temps (pour ne pas saturer les disques).

Le *Background Writer* anticipe les besoins de buffers des sessions. À intervalle régulier, il se réveille et synchronise un nombre de buffers proportionnel à l'activité sur l'intervalle précédent, dans ceux qui seront examinés par les sessions pour les prochaines allocations. Quatre paramètres régissent son comportement :

- `bgwriter_delay` (défaut : 200 ms) : la fréquence à laquelle se réveille le *Background Writer* ;
- `bgwriter_lru_maxpages` (défaut : 100) : le nombre maximum de pages pouvant être écrites sur chaque tour d'activité. Ce paramètre permet d'éviter que le *Background Writer* ne veuille synchroniser trop de pages si l'activité des sessions est trop intense : dans ce cas, autant les laisser effectuer elles-mêmes les synchronisations, étant donné que la charge est forte ;
- `bgwriter_lru_multiplier` (defaut : 2) : le coefficient multiplicateur utilisé pour calculer le nombre de buffers à libérer par rapport aux demandes d'allocation sur la période précédente ;
- `bgwriter_flush_after` (défaut : 512 ko sous Linux, 0 ou désactivé ailleurs) : à partir de quelle quantité de données écrites une synchronisation sur disque est demandée.

Pour les paramètres `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier`, *lru* signifie *Least Recently Used* que l'on pourrait traduire par « moins récemment utilisé ». Ainsi, pour ce mécanisme, le *Background Writer* synchronisera les pages du cache qui ont été utilisées le moins récemment.

Le *checkpointer* est responsable d'un autre mécanisme : il synchronise tous les blocs modifiés lors des checkpoints. Son rôle est d'effectuer cette synchronisation, en évitant de saturer les disques en lissant la charge (voir plus loin).

Lors d'écritures intenses, il est possible que ces deux mécanismes soient débordés. Les processus *backend* peuvent alors écrire eux-mêmes dans les fichiers de données (après les journaux de transaction, bien sûr). Cette situation est évidemment à éviter, ce qui implique généralement de rendre le *bgwriter* plus agressif.

1.5 JOURNALISATION

- Garantir la durabilité des données
- Base encore cohérente après :
  - arrêt brutal des processus
  - crash machine
  - ...
- Écriture des modifications dans un journal **avant** les fichiers de données
- WAL : *Write Ahead Logging*

La journalisation, sous PostgreSQL, permet de garantir l'intégrité des fichiers, et la durabilité des opérations :

- L'intégrité : quoi qu'il arrive, exceptée la perte des disques de stockage bien sûr, la base reste cohérente. Un arrêt d'urgence ne corrompra pas la base.
- Toute donnée validée (`COMMIT`) est écrite. Un arrêt d'urgence ne va pas la faire disparaître.

Pour cela, le mécanisme est relativement simple : toute modification affectant un fichier sera d'abord écrite dans le journal. Les modifications affectant les vrais fichiers de données ne sont écrites qu'en mémoire, dans les *shared buffers*. Elles seront écrites de façon asynchrone, soit par un processus recherchant un buffer libre, soit par le *Background Writer*, soit par le *Checkpointer*.

Les écritures dans le journal, bien que synchrones, sont relativement performantes, car elles sont séquentielles (moins de déplacement de têtes pour les disques).

1.5.1 JOURNAUX DE TRANSACTION (RAPPELS)

Essentiellement :

- `pg_wal/` : journaux de transactions
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
- `pg_xact/` : état des transactions
- **Ces fichiers sont vitaux !**

Rappelons que les journaux de transaction sont des fichiers de 16 Mo par défaut, stockés dans `PGDATA/pg_wal` (`pg_xlog` avant la version 10), dont les noms comportent le numéro de *timeline*, un numéro de journal de 4 Go et un numéro de segment, en hexadécimal.

```
$ ls -l
total 2359320
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007C
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007D
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000023
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000024
drwx------ 2 postgres postgres    16384 Mar 26 16:28 archive_status
```

Le sous-répertoire `archive_status` est lié à l'archivage.

D'autres plus petits répertoires comme `pg_xact`, qui contient les statuts des transactions passées, ou `pg_commit_ts`, `pg_multixact`, `pg_serial`, `pg_snapshots`, `pg_subtrans` ou encore `pg_twophase` sont également impliqués.

Tous ces répertoires sont critiques, gérés par PostgreSQL, et ne doivent pas être modifiés !

1.5.2 CHECKPOINT

- Point de reprise
- À partir d'où rejouer les journaux ?
- Données écrites au moins au niveau du checkpoint
  - il peut durer
- Processus `checkpointer`

PostgreSQL trace les modifications de données dans les journaux WAL. Si le système ou l'instance sont arrêtés brutalement, il faut que PostgreSQL puisse appliquer le contenu

des journaux non traités sur les fichiers de données. Il a donc besoin de savoir à partir d'où rejouer ces données. Ce point est ce qu'on appelle un *checkpoint*, ou *point de reprise*.

Les principes sont les suivants :

Toute entrée dans les journaux est idempotente, c'est-à-dire qu'elle peut être appliquée plusieurs fois, sans que le résultat final ne soit changé. C'est nécessaire, au cas où la récupération serait interrompue, ou si un fichier sur lequel la reprise est effectuée était plus récent que l'entrée qu'on souhaite appliquer.

Tout fichier de journal antérieur à l'avant-dernier point de reprise valide (ou au dernier à partir de la version 11) **peut être supprimé** ou recyclé, car il n'est plus nécessaire à la récupération.

PostgreSQL a besoin des fichiers de données qui contiennent toutes les données jusqu'au point de reprise. Ils peuvent être plus récents et contenir des informations supplémentaires, ce n'est pas un problème.

Un checkpoint n'est pas un « instantané » cohérent de l'ensemble des fichiers. C'est simplement l'endroit à partir duquel les journaux doivent être rejoués. Il faut donc pouvoir garantir que tous les blocs modifiés dans le cache *au démarrage du checkpoint* auront été synchronisés sur le disque quand le checkpoint sera terminé, et donc marqué comme dernier checkpoint valide. Un checkpoint peut donc durer plusieurs minutes, sans que cela ne bloque l'activité.

C'est le processus `checkpointer` qui est responsable de l'écriture des buffers devant être synchronisés durant un checkpoint.

1.5.3 DÉCLENCHEMENT & COMPORTEMENT DES CHECKPOINTS

- Déclenchement périodique (dans l'idéal)
  - `checkpoint_timeout`
- Quantité de journaux
  - `max_wal_size` (pas un plafond !)
- Dilution des écritures
  - `checkpoint_completion_target` × durée moyenne entre deux checkpoints
- Surveillance :
  - `checkpoint_warning`, `log_checkpoints`

Plusieurs paramètres influencent le comportement des checkpoints.

Dans l'idéal les checkpoints sont périodiques. Le temps maximum entre deux checkpoints

est fixé par `checkpoint_timeout` (par défaut 300 secondes). C'est parfois un peu court pour les grosses instances.

Quand le checkpoint démarre, il vise à lisser le débit en écriture, et donc le calcule à partir d'une fraction de la durée d'exécution des précédents checkpoints. Cette fraction est fixée par `checkpoint_completion_target`, et vaut 0,5 par défaut jusqu'en version 13 incluse, et 0,9 depuis la version 14. PostgreSQL prévoit donc une durée de checkpoint de 150 secondes au départ, mais cette valeur pourra évoluer ensuite suivant la durée réelle des checkpoints précédents. La valeur préconisée pour `checkpoint_completion_target` est 0,9 car elle permet de lisser davantage les écritures dues aux checkpoints dans le temps.

Le checkpoint intervient aussi quand le volume des journaux dépasse le seuil défini par `max_wal_size`, par défaut 1 Go. Un checkpoint est alors déclenché.

Attention : le terme peut porter à confusion, le volume de l'ensemble des fichiers WAL peut bien dépasser la taille fixée par le paramètre `max_wal_size` en cas de forte activité, ce n'est **pas** une valeur plafond !
De plus, les journaux peuvent encore être retenus dans `pg_wal/` pour les besoins de l'archivage ou de la réplication.

Une fois le checkpoint terminé, les journaux inutiles sont recyclés, ou effacés pour redescendre en-dessous de la quantité définie par `max_wal_size`.

Il existe un paramètre `min_wal_size` (défaut : 80 Mo) qui fixe la quantité minimale de journaux, même en cas d'activité en écriture inexistante. Ils seront prêts à être remplis en cas d'écriture imprévue. Après un gros pic d'activité suivi d'une période calme, la quantité de journaux va très progressivement redescendre de `max_wal_size` à `min_wal_size`.

Avant PostgreSQL 9.5, le rôle de `max_wal_size` était tenu par le paramètre `checkpoint_segments`, exprimé en nombre de segments de journaux maximum entre deux checkpoints, par défaut à seulement 3 (soit 48 Mo), et qu'il était conseillé d'augmenter fortement.

Le dimensionnement de ces paramètres est très dépendant du contexte et de l'activité habituelle. Le but est d'éviter des gros pics d'écriture, et donc d'avoir des checkpoints essentiellement périodiques, même si des opérations ponctuelles peuvent y échapper (gros chargements, grosse maintenance…).

Des checkpoints plus rares ont également pour effet de réduire la quantité totale de journaux écrits. Par défaut, un bloc modifié est en effet intégralement écrit dans les journaux la première fois après un checkpoint.

Par contre, un écart plus grand entre checkpoints peut entraîner une restauration plus longue après un arrêt brutal, car il y aura plus de journaux à rejouer.

Si l'on monte `max_wal_size`, par cohérence, il faudra penser à augmenter aussi `checkpoint_timeout`, et vice-versa.

Il est possible de suivre le déroulé des checkpoints dans les traces si `log_checkpoints` est à `on`. De plus, si deux checkpoints sont rapprochés d'un intervalle de temps inférieur à `checkpoint_warning` (défaut : 30 secondes), un message d'avertissement sera tracé.

Le *sync* sur disque n'a lieu qu'en fin de checkpoint. Toujours pour éviter des à-coups d'écriture, PostgreSQL demande au système d'exploitation de forcer un vidage du cache quand `checkpoint_flush_after` a déjà été écrit (par défaut 256 ko).

Avant PostgreSQL 9.6, ceci se paramétrait au niveau de Linux en abaissant les valeurs des *sysctl* `vm.dirty_*`.

1.5.4 WAL BUFFERS : JOURNALISATION EN MÉMOIRE

- Mutualiser les écritures entre transactions
- Un processus d'arrière plan : `walwriter`
- Paramètres notables :
  - `wal_buffers`
  - `wal_writer_flush_after`
- Fiabilité :
  - `fsync` = `on`
  - `full_page_writes` = `on`
  - sinon **corruption** !

La journalisation s'effectue par écriture dans les journaux de transactions. Toutefois, afin de ne pas effectuer des écritures synchrones pour chaque opération dans les fichiers de journaux, les écritures sont préparées dans des tampons (*buffers*) en mémoire. Les processus écrivent donc leur travail de journalisation dans des *buffers*, ou *WAL buffers*. Ceux-ci sont vidés quand une session demande validation de son travail (`COMMIT`), qu'il n'y a plus de *buffer* disponible, ou que le *walwriter* se réveille (`wal_writer_delay`).

Écrire un ou plusieurs blocs séquentiels de façon synchrone sur un disque a le même coût à peu de chose près. Ce mécanisme permet donc de réduire fortement les demandes d'écriture synchrone sur le journal, et augmente donc les performances.

Afin d'éviter qu'un processus n'ait tous les buffers à écrire à l'appel de `COMMIT`, et que cette opération ne dure trop longtemps, un processus d'arrière-plan appelé *walwriter* écrit à intervalle régulier tous les buffers à synchroniser.

Ce mécanisme est géré par ces paramètres, rarement modifiés :

- `wal_buffers` : taille des *WAL buffers*, soit par défaut 1/32e de `shared_buffers` avec un maximum de 16 Mo (la taille d'un segment), des valeurs supérieures pouvant être intéressantes pour les très grosses charges ;
- `wal_writer_delay` (défaut : 200 ms) : intervalle auquel le *walwriter* se réveille pour écrire les buffers non synchronisés ;
- `wal_writer_flush_after` (défaut : 1 Mo) : au-delà de cette valeur, les journaux écrits sont synchronisés sur disque pour éviter l'accumulation dans le cache de l'OS.

Pour la fiabilité, on ne touchera pas à ceux-ci :

- `wal_sync_method` : appel système à utiliser pour demander l'écriture synchrone (sauf très rare exception, PostgreSQL détecte tout seul le bon appel système à utiliser) ;
- `full_page_writes` : doit-on réécrire une image complète d'une page suite à sa première modification après un checkpoint ? Sauf cas très particulier, comme un système de fichiers *Copy On Write* comme ZFS ou btrfs, ce paramètre doit rester à `on` pour éviter des corruptions de données ;
- `fsync` : doit-on réellement effectuer les écritures synchrones ? Le défaut est `on` et **il est très fortement conseillé de le laisser ainsi en production**. Avec `off`, les performances en écritures sont certes très accélérées, mais en cas d'arrêt d'urgence de l'instance, les données seront totalement corrompues ! Ce peut être intéressant pendant le chargement initial d'une nouvelle instance par exemple, sans oublier de revenir à `on` après ce chargement initial. (D'autres paramètres et techniques existent pour accélérer les écritures et sans corrompre votre instance, si vous êtes prêt à perdre certaines données non critiques : `synchronous_commit` à `off`, les tables *unlogged*…)

1.5.5 COMPRESSION DES JOURNAUX

- `wal_compression`
  - compression
  - un peu de CPU

`wal_compression` compresse les blocs complets enregistrés dans les journaux de transactions, réduisant le volume des WAL et la charge en écriture sur les disques.

Le rejeu des WAL est aussi plus rapide, ce qui accélère la réplication et la reprise après un crash. Le prix est une augmentation de la consommation en CPU.

1.5.6 LIMITER LE COÛT DE LA JOURNALISATION

- `synchronous_commit`
  - perte potentielle de données validées
- `commit_delay` / `commit_siblings`
- Par session

Le coût d'un `fsync` est parfois rédhibitoire. Avec certains sacrifices, il est parfois possible d'améliorer les performances sur ce point.

Le paramètre `synchronous_commit` (défaut : `on`) indique si la validation de la transaction en cours doit déclencher une écriture synchrone dans le journal. Le défaut permet de garantir la pérennité des données dès la fin du `COMMIT`.

Mais ce paramètre peut être modifié dans chaque session par une commande `SET`, et passé à `off` **s'il est possible d'accepter une petite perte de données** pourtant committées. La perte peut monter à `3 × wal_writer_delay` (600 ms) ou `wal_writer_flush_after` (1 Mo) octets écrits. On accélère ainsi notablement les flux des petites transactions. Les transactions où le paramètre reste à `on` continuent de profiter de la sécurité maximale. La base restera, quoi qu'il arrive, cohérente. (Ce paramètre permet aussi de régler le niveau des transactions synchrones avec des secondaires.)

Il existe aussi `commit_delay` (défaut : `0`) et `commit_siblings` (défaut : `5`) comme mécanisme de regroupement de transactions[6] . S'il y au moins `commit_siblings` transactions en cours, PostgreSQL attendra jusqu'à `commit_delay` (en microsecondes) avant de valider une transaction pour permettre à d'autres transactions de s'y rattacher. Ce mécanisme, désactivé par défaut, accroît la latence de certaines transactions afin que plusieurs soient écrites ensembles, et n'apporte un gain de performance global qu'avec de nombreuses petites transactions en parallèle, et des disques classiques un peu lents. (En cas d'arrêt brutal, il n'y a pas à proprement parler de perte de données puisque les transactions délibérément retardées n'ont pas été signalées comme validées.)

---

[6] https://docs.postgresql.fr/current/wal-configuration.html

1.6 AU-DELÀ DE LA JOURNALISATION

- Sauvegarde PITR
- Réplication physique
  - par *log shipping*
  - par *streaming*

Le système de journalisation de PostgreSQL étant très fiable, des fonctionnalités très intéressantes ont été bâties dessus.

1.6.1 L'ARCHIVAGE DES JOURNAUX

- Repartir à partir :
  - d'une vieille sauvegarde
  - les journaux archivés
- Sauvegarde à chaud
- Sauvegarde en continu
- Paramètres : `wal_level`, `archive_mode`, `archive_command`

Les journaux permettent de rejouer, suite à un arrêt brutal de la base, toutes les modifications depuis le dernier checkpoint. Les journaux devenus obsolète depuis le dernier *checkpoint* (l'avant-dernier avant la version 11) sont à terme recyclés ou supprimés, car ils ne sont plus nécessaires à la réparation de la base.

Le but de l'archivage est de stocker ces journaux, afin de pouvoir rejouer leur contenu, non plus depuis le dernier checkpoint, mais **depuis une sauvegarde**. Le mécanisme d'archivage permet de repartir d'une sauvegarde binaire de la base (c'est-à-dire des fichiers, pas un `pg_dump`), et de réappliquer le contenu des journaux archivés.

Il suffit de rejouer tous les journaux depuis le checkpoint précédent la sauvegarde jusqu'à la fin de la sauvegarde, ou même à un point précis dans le temps. L'application de ces journaux permet de rendre à nouveau cohérents les fichiers de données, même si ils ont été sauvegardés en cours de modification.

Ce mécanisme permet aussi de fournir une sauvegarde continue de la base, alors même que celle-ci travaille.

Tout ceci est vu dans le module *Point In Time Recovery*[7] .

Même si l'archivage n'est pas en place, il faut connaître les principaux paramètres impliqués :

[7] https://dali.bo/i2_html

wal_level :

Il vaut `replica` par défaut depuis la version 10. Les journaux contiennent les informations nécessaires pour une sauvegarde PITR ou une réplication vers une instance secondaire.

Si l'on descend à `minimal` (défaut jusqu'en version 9.6 incluse), les journaux ne contiennent plus que ce qui est nécessaire à une reprise après arrêt brutal sur le serveur en cours. Ce peut être intéressant pour réduire, parfois énormément, le volume des journaux générés, si l'on a bien une sauvegarde non PITR par ailleurs.

Le niveau `logical` est destiné à la réplication logique[8] .

(Avant la version 9.6 existaient les niveaux intermédiaires `archive` et `hot_standby`, respectivement pour l'archivage et pour un serveur secondaire en lecture seule. Ils sont toujours acceptés, et assimilés à `replica`.)

archive_mode & **archive_command** :

Il faut qu'`archive_command` soit à `on` pour activer l'archivage. Les journaux sont alors copiés grâce à une commande shell à fournir dans `archive_command`. En général elle est fournie par des outils de sauvegarde dédiés (par exemple pgBackRest, pitrery ou barman).

1.6.2 RÉPLICATION

- *Log shipping* : fichier par fichier
- *Streaming* : entrée par entrée (en flux continu)
- Serveurs secondaires très proches de la production, en lecture

La restauration d'une sauvegarde peut se faire en continu sur un autre serveur, qui peut même être actif (bien que forcément en lecture seule). Les journaux peuvent être :

- envoyés régulièrement vers le secondaire, qui les rejouera : c'est le principe de la réplication par *log shipping* ;
- envoyés par fragments vers cet autre serveur : c'est la réplication par *streaming*.

Ces thèmes ne seront pas développés ici. Signalons juste que la réplication par *log shipping* implique un archivage actif sur le primaire, et l'utilisation de `restore_command` (et d'autres pour affiner) sur le secondaire. Le *streaming* permet de se passer d'archivage, même si coupler *streaming* et sauvegarde PITR est une bonne idée. Sur un PostgreSQL récent, le primaire a par défaut le nécessaire activé pour se voir doté d'un secondaire : `wal_level` est à `replica` ; `max_wal_senders` permet d'ouvrir des processus dédiés à la réplication ; et

[8] https://dali.bo/w5_html

l'on peut garder des journaux en paramétrant `wal_keep_size` (ou `wal_keep_segments` avant la version 13) pour limiter les risques de décrochage du secondaire.

Une configuration supplémentaire doit se faire sur le serveur secondaire, indiquant comment récupérer les fichiers de l'archive, et comment se connecter au primaire pour récupérer des journaux. Elle a lieu dans les fichiers `recovery.conf` (jusqu'à la version 11 comprise), ou (à partir de la version 12) `postgresql.conf` dans les sections évoquées plus haut, ou `postgresql.auto.conf`.

1.7 CONCLUSION

Mémoire et journalisation :

- complexe
- critique
- mais fiable
- et le socle de nombreuses fonctionnalités évoluées

1.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/m3_quiz

1.9 TRAVAUX PRATIQUES

1.9.1 MÉMOIRE PARTAGÉE

But : constater l'effet du cache sur les accès.

> Se connecter à la base de données `b0` et créer une table `t2` avec une colonne `id` de type integer.

> Insérer 500 lignes dans la table `t2` avec `generate_series`.

> Réinitialiser les statistiques pour `t2` avec la fonction `pg_stat_reset_single_table_counters` (l'OID qui lui sert de paramètre est dans la table des relations `pg_class`).

> Afin de vider le cache, redémarrer l'instance PostgreSQL.

> Se connecter à la base de données `b0`.
> Lire les données de la table `t2`.

> Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

1.9.2 MÉMOIRE DE TRI

Activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Si ce n'est pas déjà fait, laisser défiler dans une fenêtre le fichier de traces.

Activer le chronométrage dans la session (`\timing on`).
Lire les données de la table `t2` en triant par la colonne `id`
Qu'observe-t-on ?

Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

1.9.3 CACHE DISQUE DE POSTGRESQL

Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

Créer une table `t2` avec une colonne `id` de type integer.

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Nous allons vider les caches.
D'abord, redémarrer l'instance PostgreSQL.

Vider le cache du système d'exploitation avec :

```
# sync && echo 3 > /proc/sys/vm/drop_caches
```

Se connecter à la base de données `b1`. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ?
(Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

Activer l'affichage de la durée des requêtes.
Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

Redémarrer l'instance PostgreSQL.

Se connecter à la base de données `b1` et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

Modifier le contenu de la table `t2` (par exemple avec `UPDATE t2 SET id = 0 WHERE id < 1000;`). Que contient le cache de PostgreSQL ?

Exécuter un `CHECKPOINT`. Que contient le cache de PostgreSQL ?

1.9.4 JOURNAUX

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

> Exécuter un `CHECKPOINT`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

1.10.1 MÉMOIRE PARTAGÉE

But : constater l'effet du cache sur les accès.

> Se connecter à la base de données `b0` et créer une table `t2` avec une colonne `id` de type integer.

```
$ psql b0
```

```
b0=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

> Insérer 500 lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 500);
INSERT 0 500
```

> Réinitialiser les statistiques pour `t2` avec la fonction `pg_stat_reset_single_table_counters` (l'OID qui lui sert de paramètre est dans la table des relations `pg_class`).

Cette fonction attend un OID comme paramètre :

```
b0=# \df pg_stat_reset_single_table_counters
```

```
List of functions
-[ RECORD 1 ]-------+---------------------
Schema              | pg_catalog
Name                | pg_relation_filepath
Result data type    | text
Argument data types | regclass
Type                | func
```

L'OID est une colonne présente dans la table `pg_class` :

```
b0=# SELECT relname, pg_stat_reset_single_table_counters(oid)
     FROM pg_class WHERE relname = 't2';
```

```
 relname | pg_stat_reset_single_table_counters
---------+-------------------------------------
 t2      |
```

> Afin de vider le cache, redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-14
```

> Se connecter à la base de données `b0`.
> Lire les données de la table `t2`.

```
b0=# SELECT * FROM t2;
[...]
```

> Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

```
b0=# \x
Expanded display is on.

b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2' ;

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 0
idx_blks_read   |
idx_blks_hit    |
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

3 blocs ont été lus en dehors du cache de PostgreSQL (colonne `heap_blks_read`).

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 3
…
```

Les 3 blocs sont maintenant lus à partir du cache de PostgreSQL (colonne `heap_blks_hit`).

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 6
…
```

Quelle que soit la session, le cache étant partagé, tout le monde profite des données en cache.

1.10.2 MÉMOIRE DE TRI

> Activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Dans la session :

```
postgres=# SET client_min_messages TO log;
SET
postgres=# SET log_temp_files TO 0;
SET
```

Les paramètres `log_temp_files` et `client_min_messages` peuvent aussi être mis en place une fois pour toutes dans `postgresql.conf` (recharger la configuration). En fait, c'est généralement conseillé.

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 1000000);

INSERT 0 1000000
```

> Si ce n'est pas déjà fait, laisser défiler dans une fenêtre le fichier de traces.

Le nom du fichier dépend de l'installation et du moment. Pour suivre tout ce qui se passe dans le fichier de traces, utiliser `tail -f` :

```
$ tail -f /var/lib/pgsql/14/data/log/postgresql-Tue.log
```

> Activer le chronométrage dans la session (`\timing on`).
> Lire les données de la table `t2` en triant par la colonne `id`
> Qu'observe-t-on ?

```
b0=# \timing on
b0=# SELECT * FROM t2 ORDER BY id;

LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1197.0", size 14032896
   id
---------
       1
       1
       2
       2
       3
[...]
Time: 436.308 ms
```

Le message `LOG` apparaît aussi dans la trace, et en général il se trouvera là.

PostgreSQL a dû créer un fichier temporaire pour stocker le résultat temporaire du tri. Ce fichier s'appelle `base/pgsql_tmp/pgsql_tmp1197.0`. Il est spécifique à la session et sera détruit dès qu'il ne sera plus utile. Il fait 14 Mo.

Écrire un fichier de tri sur disque prend évidemment un certain temps, c'est généralement à éviter si le tri peut se faire en mémoire.

> Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

```
b0=# SET work_mem TO '100MB';
SET
```

> Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2 ORDER BY id;

   id
---------
```

```
       1
       1
       2
       2
[...]
Time: 240.565 ms
```

Il n'y a plus de fichier temporaire généré. La durée d'exécution est bien moindre.

1.10.3 CACHE DISQUE DE POSTGRESQL

> Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

```
b1=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

> Créer une table `t2` avec une colonne `id` de type integer.

```
b1=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

> Nous allons vider les caches.
> D'abord, redémarrer l'instance PostgreSQL.

```
 # systemctl restart postgresql-14
```

> Vider le cache du système d'exploitation avec :
> `# sync && echo 3 > /proc/sys/vm/drop_caches`

> Se connecter à la base de données `b1`. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ?
> (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

```
b1=# SELECT relfilenode, count(*)
  FROM pg_buffercache
```

```
  GROUP BY 1
  ORDER BY 2 DESC
  LIMIT 10;

 relfilenode | count
-------------+-------
             | 16181
        1249 |    57
        1259 |    26
        2659 |    15
[...]
```

Les valeurs exactes peuvent varier. La colonne `relfilenode` correspond à l'identifiant système de la table. La deuxième colonne indique le nombre de blocs. Il y a ici 16 181 blocs non utilisés pour l'instant dans le cache (126 Mo), ce qui est logique vu que PostgreSQL vient de redémarrer. Il y a quelques blocs utilisés par des tables systèmes, mais aucune table utilisateur (on les repère par leur OID supérieur à 16384).

> Activer l'affichage de la durée des requêtes.
> Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
Timing is on.

b1=# SELECT * FROM t2;

   id
---------
       1
       2
       3
       4
       5
[...]
Time: 277.800 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+-------
             | 16220
       16410 |    32
        1249 |    29
        1259 |     9
        2659 |     8
```

```
[...]
Time: 30.694 ms
```

32 blocs ont été alloués pour la lecture de la table `t2` (*filenode* 16410). Cela représente 256 ko alors que la table fait 35 Mo :

```
b1=# SELECT pg_size_pretty(pg_table_size('t2'));

 pg_size_pretty
----------------
 35 MB
(1 row)

Time: 1.913 ms
```

Un simple `SELECT *` ne suffit donc pas à maintenir la table dans le cache. Par contre, ce deuxième accès était déjà beaucoup rapide, ce qui suggère que le système d'exploitation, lui, a probablement gardé les fichiers de la table dans son propre cache.

> Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# SELECT * FROM t2;

   id
---------
[...]
Time: 184.529 ms

b1=#  SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+-------
             | 16039
        1249 |    85
       16410 |    64
        1259 |    39
        2659 |    22
[...]
```

Il y en en a un peu plus dans le cache (en fait, 2 fois 32 ko). Plus vous exécuterez la requête, et plus le nombre de blocs présents en cache augmentera. Sur le long terme, les 4425 blocs de la table `t2` peuvent se retrouver dans le cache.

> Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

Pour cela, il faut ouvrir le fichier de configuration `postgresql.conf` et modifier la valeur du paramètre `shared_buffers` à un quart de la mémoire. Par exemple :

```
shared_buffers = 2GB
```

> Redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-14
```

> Se connecter à la base de données `b1` et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
b1=# SELECT * FROM t2;

   id
---------
       1
[...]
Time: 340.444 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+--------
             | 257581
       16410 |   4425
        1249 |     29
[...]
```

PostgreSQL se retrouve avec toute la table directement dans son cache, et ce dès la première exécution.

PostgreSQL est optimisé principalement pour du multi-utilisateurs. Dans ce cadre, il faut pouvoir exécuter plusieurs requêtes en même temps et donc chaque requête ne peut pas monopoliser tout le cache. De ce fait, chaque requête ne peut prendre qu'une partie réduite du cache. Mais plus le cache est gros, plus la partie octroyée est grosse.

> Modifier le contenu de la table `t2` (par exemple avec `UPDATE t2 SET id = 0 WHERE id < 1000;`). Que contient le cache de PostgreSQL ?

```
b1=# UPDATE t2 SET id=0 WHERE id < 1000;
UPDATE 999
```

```
b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4419 | 35 MB
 t2      | t       |    15 | 120 kB
```

15 blocs ont été modifiés (`isdirty` est à `true`), le reste n'a pas bougé.

> Exécuter un `CHECKPOINT`. Que contient le cache de PostgreSQL ?

```
b1=# CHECKPOINT;
CHECKPOINT

b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4434 | 35 MB
```

Les blocs *dirty* ont tous été écrits sur le disque et sont devenus « propres ».

1.10.4 JOURNAUX

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

```
b1=# INSERT INTO t2 SELECT generate_series(1, 10000000);
INSERT 0 10000000

$ ls -al $PGDATA/pg_wal
total 131076
$ ls -al $PGDATA/pg_wal
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:55 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000033
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000034
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000035
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000054
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000055
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000056
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000057
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000058
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000059
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Des journaux de transactions sont écrits lors des écritures dans la base. Leur nombre varie avec l'activité récente.

> Exécuter un `CHECKPOINT`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

```
b1=# CHECKPOINT;
CHECKPOINT

$ ls -al $PGDATA/pg_wal
total 131076
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:56 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:56 000000010000000000000059
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005A
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005B
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000079
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007A
```

```
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007B
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007C
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007D
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007E
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007F
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Le nombre n'a pas forcément décru, mais le dernier journal d'avant le checkpoint est à présent le plus ancien (selon l'ordre des noms des journaux). Les anciens journaux devenus obsolètes sont recyclés, prêts à être remplis à nouveau. Noter que leur date de création n'a pas été mise à jour !

NOTES

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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