

Module M5

VACUUM & autovacuum



24.04

Table des matières

Sur ce document	1
Chers lectrices & lecteurs,	1
À propos de DALIBO	1
Remerciements	2
Forme de ce manuel	2
Licence Creative Commons CC-BY-NC-SA	2
Marques déposées	3
Versions de PostgreSQL couvertes	3
1/ VACUUM et autovacuum	5
1.1 Au menu	6
1.2 VACUUM et autovacuum	7
1.3 Fonctionnement de VACUUM	8
1.3.1 Fonctionnement de VACUUM (suite)	9
1.3.2 Fonctionnement de VACUUM (suite)	10
1.4 Les options de VACUUM	11
1.4.1 Tâches d'un VACUUM	11
1.4.2 Options de performance de VACUUM	13
1.4.3 Options pour un VACUUM en urgence	15
1.4.4 Autres options de VACUUM	16
1.5 Suivi du VACUUM	18
1.5.1 Progression du VACUUM	19
1.6 Autovacuum	22
1.6.1 Paramétrage du déclenchement de l'autovacuum	22
1.6.2 Déclenchement de l'autovacuum	23
1.6.3 Déclenchement de l'autovacuum (suite)	24
1.7 Paramétrage de VACUUM & autovacuum	26
1.7.1 VACUUM vs autovacuum	26
1.7.2 Mémoire	27
1.7.3 Bridage du VACUUM et de l'autovacuum	28
1.7.4 Paramétrage du FREEZE (1)	30
1.7.5 Paramétrage du FREEZE (2)	31
1.8 Autres problèmes courants	35
1.8.1 L'autovacuum dure trop longtemps	35
1.8.2 Arrêter un VACUUM ?	36
1.8.3 Ce qui peut bloquer le VACUUM FREEZE	37
1.9 Résumé des conseils sur l'autovacuum	39
1.9.1 Résumé des conseils sur l'autovacuum (1/2)	39
1.9.2 Résumé des conseils sur l'autovacuum (2/2)	40
1.10 Conclusion	41
1.10.1 Questions	41
1.11 Quiz	42

1.12 Travaux pratiques	43
1.12.1 Traiter la fragmentation	43
1.12.2 Détecter la fragmentation	44
1.12.3 Gestion de l'autovacuum	45
1.13 Travaux pratiques (solutions)	47
1.13.1 Traiter la fragmentation	47
1.13.2 Détecter la fragmentation	50
1.13.3 Gestion de l'autovacuum	52
Les formations Dalibo	59
Cursus des formations	59
Les livres blancs	60
Téléchargement gratuit	60

Sur ce document

Formation	Module M5
Titre	VACUUM & autovacuum
Révision	24.04
PDF	https://dali.bo/m5_pdf
EPUB	https://dali.bo/m5_epub
HTML	https://dali.bo/m5_html
Slides	https://dali.bo/m5_slides
TP	https://dali.bo/m5_tp
TP (solutions)	https://dali.bo/m5_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

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.com¹ !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur <https://dalibo.com/formations>

¹<mailto:formation@dalibo.com>

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, Pierrick Chovelon, 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, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachaires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**². Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

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

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

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

²<http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées³ par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

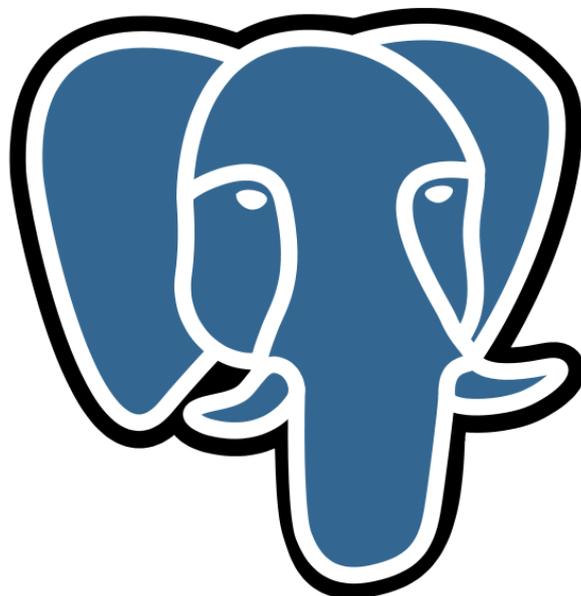
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

³<https://www.postgresql.org/about/policies/trademarks/>

1/ VACUUM et autovacuum



1.1 AU MENU



- Principe & fonctionnement du `VACUUM`
- Options : `VACUUM` seul, `ANALYZE` , `FULL` , `FREEZE`
 - ne pas les confondre !
- Suivi
- Autovacuum
- Paramétrages

`VACUUM` est la contrepartie de la flexibilité du modèle MVCC. Derrière les différentes options de `VACUUM` se cachent plusieurs tâches très différentes. Malheureusement, la confusion est facile. Il est capital de les connaître et de comprendre leur fonctionnement.

Autovacuum permet d'automatiser le `VACUUM` et allège considérablement le travail de l'administrateur.

Il fonctionne généralement bien, mais il faut savoir le surveiller et l'optimiser.

1.2 VACUUM ET AUTOVACUUM



- `VACUUM` : nettoie d'abord les lignes mortes
- Mais aussi d'autres opérations de maintenance
- Lancement
 - manuel
 - par le démon `autovacuum` (seuils)

`VACUUM` est né du besoin de nettoyer les lignes mortes. Au fil du temps il a été couplé à d'autres ordres (`ANALYZE` , `VACUUM FREEZE`) et s'est occupé d'autres opérations de maintenance (création de la *visibility map* par exemple).

`autovacuum` est un processus de l'instance PostgreSQL. Il est activé par défaut, et il fortement conseillé de le conserver ainsi. Dans le cas général, son fonctionnement convient et il ne gênera pas les utilisateurs.

L'autovacuum ne gère pas toutes les variantes de `VACUUM` (notamment pas le `FULL`).

1.3 FONCTIONNEMENT DE VACUUM

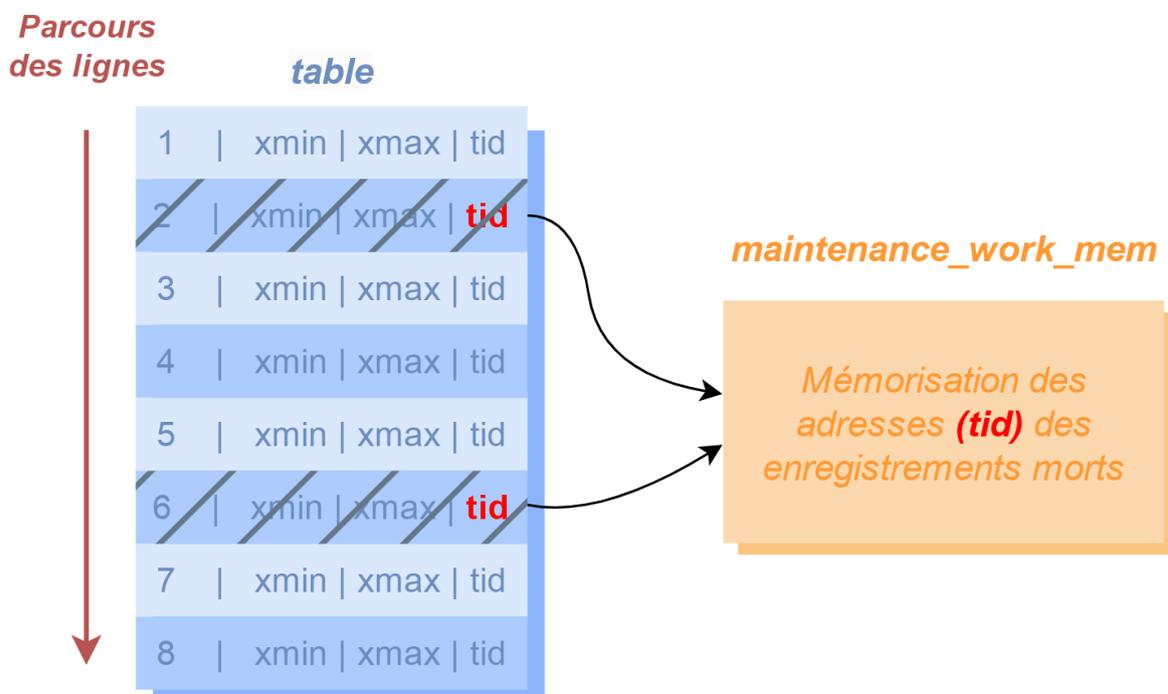


Figure 1/ .1: Phase 1/3 : recherche des enregistrements morts

Un ordre `VACUUM` vise d'abord à nettoyer les lignes mortes.

Le traitement `VACUUM` se déroule en trois passes. Cette première passe parcourt la table à nettoyer, à la recherche d'enregistrements morts. Un enregistrement est mort s'il possède un `xmax` qui correspond à une transaction validée, et que cet enregistrement n'est plus visible dans l'instantané d'aucune transaction en cours sur la base. D'autres lignes mortes portent un `xmin` d'une transaction annulée.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. La session effectuant le vacuum garde en mémoire la liste des adresses des enregistrements morts, à hauteur d'une quantité indiquée par le paramètre `maintenance_work_mem`. Si cet espace est trop petit pour contenir tous les enregistrements morts, `VACUUM` effectue plusieurs séries de ces trois passes.

1.3.1 Fonctionnement de VACUUM (suite)

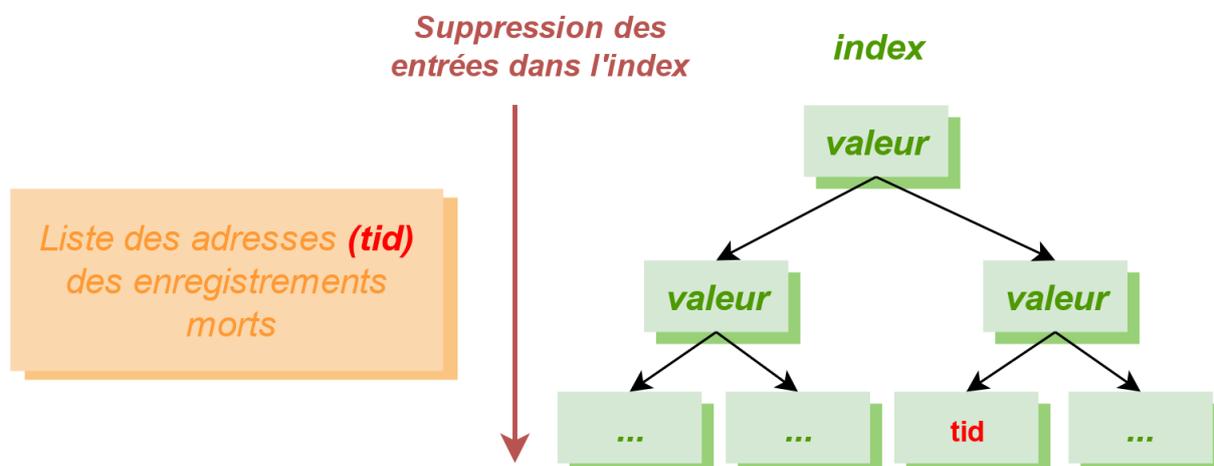


Figure 1/ .2: Phase 2/3 : nettoyage des index

La seconde passe se charge de nettoyer les entrées d'index. `VACUUM` possède une liste de `tid` (`tuple id`) à invalider. Il parcourt donc tous les index de la table à la recherche de ces `tid` et les supprime. En effet, les index sont triés afin de mettre en correspondance une valeur de clé (la colonne indexée par exemple) avec un `tid`. Il n'est par contre pas possible de trouver un `tid` directement. Les pages entièrement vides sont supprimées de l'arbre et stockées dans la liste des pages réutilisables, la *Free Space Map* (FSM).

Pour gagner du temps si le temps presse, cette phase peut être ignorée de deux manières. La première est de désactiver l'option `INDEX_CLEANUP` :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) nom_table ;
```

À partir de la version 14, un autre mécanisme, automatique cette fois, a été ajouté. Le but est toujours d'exécuter rapidement le `VACUUM`, mais uniquement pour éviter le *wraparound*. Quand la table atteint l'âge, très élevé, de 1,6 milliard de transactions (défaut des paramètres `vacuum_failsafe_age` et `vacuum_multixact_failsafe_age`), un `VACUUM` simple va automatiquement désactiver le nettoyage des index pour nettoyer plus rapidement la table et permettre d'avancer l'identifiant le plus ancien de la table.

À partir de la version 13, cette phase de nettoyage des index peut être parallélisée (clause `PARALLEL`), chaque index pouvant être traité par un CPU.

1.3.2 Fonctionnement de VACUUM (suite)

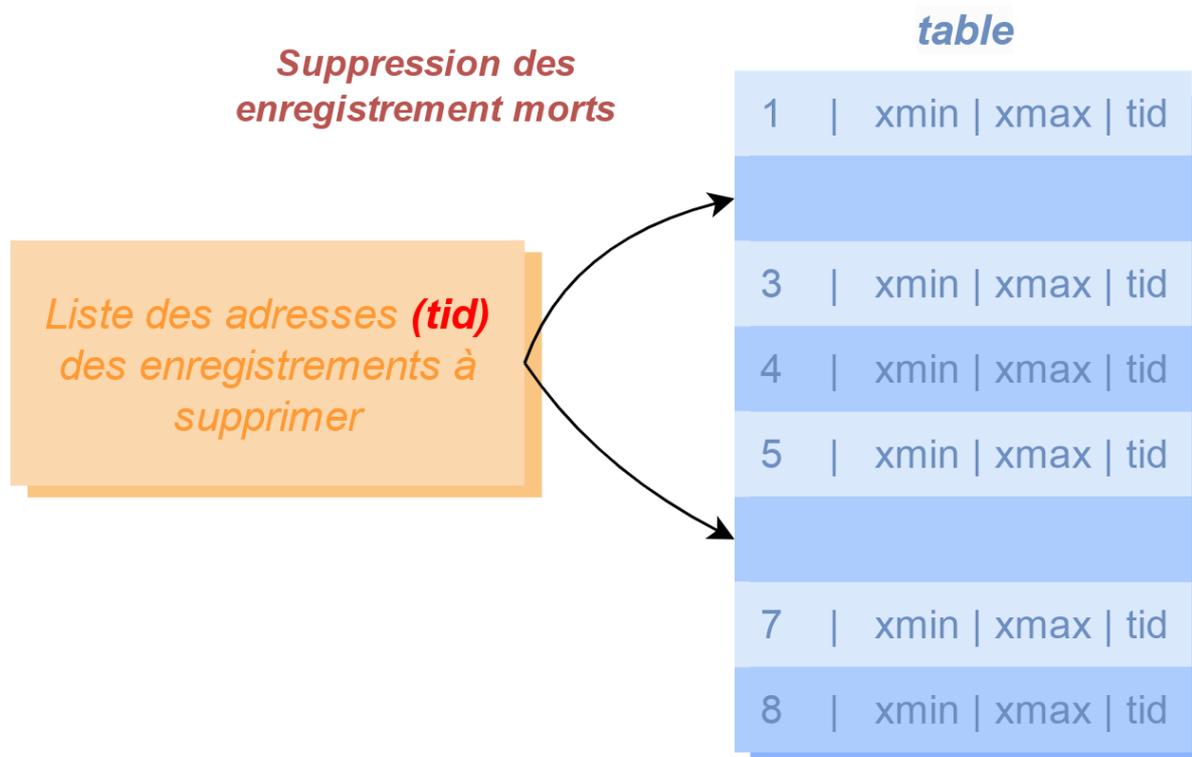


Figure 1/ .3: Phase 3/3 : suppression des enregistrements morts



NB : L'espace est rarement rendu à l'OS !

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements morts identifiés, ceux-ci peuvent disparaître. C'est le rôle de cette passe. Quand un enregistrement est supprimé d'un bloc, ce bloc est complètement réorganisé afin de consolider l'espace libre. Cet espace est renseigné dans la *Free Space Map* (FSM).

Une fois cette passe terminée, si le parcours de la table n'a pas été terminé lors de la passe précédente, le travail reprend où il en était du parcours de la table.

Si les derniers blocs de la table sont vides, ils sont rendus au système (si le verrou nécessaire peut être obtenu, et si l'option `TRUNCATE` n'est pas `off`). C'est le seul cas où `VACUUM` réduit la taille de la table. Les espaces vides (et réutilisables) au milieu de la table constituent le *bloat* (littéralement « boursoufflure » ou « gonflement », que l'on peut aussi traduire par fragmentation).

Les statistiques d'activité sont aussi mises à jour.

1.4 LES OPTIONS DE VACUUM



- Quelle tâche ?
- Comment améliorer les performances ?
- Quelles options en cas d'urgence ?
- Autres options

1.4.1 Tâches d'un VACUUM



Ne pas confondre :

- `VACUUM` seul
 - nettoyage des lignes mortes, *visibility map*, *hint bits*
- `ANALYZE`
 - statistiques sur les données
- `VACUUM (ANALYZE)`
 - nettoyage & statistiques
- `VACUUM (FREEZE)`
 - gel des lignes
 - parfois gênant ou long
- `VACUUM FULL`
 - bloquant !
 - jamais lancé par l'autovacuum

VACUUM

Par défaut, `VACUUM` procède principalement au nettoyage des lignes mortes. Pour que cela soit efficace, il met à jour la *visibility map*, et la crée au besoin. Au passage, il peut geler certaines lignes rencontrées.

L'autovacuum le déclenchera sur les tables en fonction de l'activité.

Le verrou `SHARE UPDATE EXCLUSIVE` posé protège la table contre les modifications simultanées du schéma, et ne gêne généralement pas les opérations, sauf les plus intrusives (il empêche par exemple

un `LOCK TABLE`). L'autovacuum arrêtera spontanément un `VACUUM` qu'il aurait lancé et qui gênerait ; mais un `VACUUM` lancé manuellement continuera jusqu'à la fin.

VACUUM ANALYZE

`ANALYZE` existe en tant qu'ordre séparé, pour rafraîchir les statistiques sur un échantillon des données, à destination de l'optimiseur. L'autovacuum se charge également de lancer des `ANALYZE` en fonction de l'activité.

L'ordre `VACUUM ANALYZE` (ou `VACUUM (ANALYZE)`) force le calcul des statistiques sur les données en même temps que le `VACUUM` .

VACUUM FREEZE

`VACUUM FREEZE` procède au « gel » des lignes visibles par toutes les transactions en cours sur l'instance, afin de parer au problème du *wraparound* des identifiants de transaction.

Un ordre `FREEZE` n'existe pas en tant que tel.

Préventivement, lors d'un `VACUUM` simple, l'autovacuum procède au gel de certaines des lignes rencontrées. De plus, il lancera un `VACUUM FREEZE` sur une table dont les plus vieilles transactions dépassent un certain âge. Ce peut être très long, et très lourd en écritures si une grosse table doit être entièrement gelée d'un coup. Autrement, l'activité n'est qu'exceptionnellement gênée (voir plus bas).

L'opération de gel sera détaillée plus loin.

VACUUM FULL

L'ordre `VACUUM FULL` permet de reconstruire la table sans les espaces vides. C'est une opération très lourde, risquant de bloquer d'autres requêtes à cause du verrou exclusif qu'elle pose (on ne peut même plus lire la table !), mais il s'agit de la seule option qui permet de réduire la taille de la table au niveau du système de fichiers de façon certaine.

Il faut prévoir l'espace disque (la table est reconstruite à côté de l'ancienne, puis l'ancienne est supprimée). Les index sont reconstruits au passage. Un `VACUUM FULL` gèle agressivement les lignes, et effectue donc au passage l'équivalent d'un `FREEZE` .

L'autovacuum ne lancera jamais un `VACUUM FULL` !

Il existe aussi un ordre `CLUSTER` , qui permet en plus de trier la table suivant un des index.

1.4.2 Options de performance de VACUUM



- Index :
 - `PARALLEL` (v13+)
- Taille du *buffer ring* (v16+)
 - `BUFFER_USAGE_LIMIT`
 - `vacuum_buffer_usage_limit` (256 ko)
- `SKIP_DATABASE_STATS`, `ONLY_DATABASE_STATS` (v16+)
- Éviter les verrous
 - `SKIP_LOCKED`
 - `SET lock_timeout = '1s'`

PARALLEL :

Apparue avec PostgreSQL 13, l'option `PARALLEL` permet le traitement parallélisé des index. Le nombre indiqué après `PARALLEL` précise le niveau de parallélisation souhaité. Par exemple :

```
VACUUM (VERBOSE, PARALLEL 4) matable ;
```

```
INFO: vacuuming "public.matable"
INFO: launched 3 parallel vacuum workers for index cleanup (planned: 3)
```

SKIP_DATABASE_STATS, ONLY_DATABASE_STATS :

En fin d'exécution d'un `VACUUM`, même sur une seule table, le champ `pg_database.datfrozenxid` est mis à jour. Il contient le numéro de la transaction la plus ancienne non encore gélée dans toute la base de données. Cette opération impose de parcourir `pg_class` pour récupérer le plus ancien des numéros de transaction de chaque table (`relfrozenxid`). Or cette mise à jour n'est utile que pour l'autovacuum et le `VACUUM FREEZE`, et a rarement un caractère d'urgence.

Depuis la version 16, l'option `SKIP_DATABASE_STATS` demande au `VACUUM` d'ignorer la mise à jour de l'identifiant de transaction. Le principe est d'activer cette option pour les nettoyages en masse. À l'inverse, l'option `ONLY_DATABASE_STATS` demande de ne faire que la mise à jour du `datfrozenxid`, ce qui peut être fait une seule fois en fin de traitement.

L'outil `vacuumdb` procède ainsi automatiquement si le serveur est en version 16 minimum. Par exemple :

```
$ vacuumdb --echo --all
```

```

SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
  JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
  LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
  WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
  ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
...
...
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
...
...
VACUUM (ONLY_DATABASE_STATS);

```

BUFFER_USAGE_LIMIT :

Cette option apparue en version 16 permet d'augmenter la taille de la mémoire partagée que peuvent utiliser `VACUUM`, et `ANALYZE`. Par défaut, cet espace nommé *buffer ring* n'est que de 256 ko de mémoire partagée, une valeur assez basse, cela pour protéger le cache (*shared buffers*). Si cette mémoire ne suffit pas, PostgreSQL doit recycler certains de ces buffers, d'où une écriture possible de journaux sur disque, avec un ralentissement à la clé.

Monter la taille du *buffer ring* avec `BUFFER_USAGE_LIMIT` permet une exécution plus rapide de `VACUUM`, et générer moins de journaux, mais il peut y avoir un impact négatif sur les autres requêtes si la mémoire partagée disponible se réduit, ou si le débit en lecture ou écriture du `VACUUM` augmente trop. Les valeurs vont de 128 ko à 16 Go ; 0 désactive le *buffer ring*, il n'y a alors aucune limite en terme d'accès aux buffers.

Exemples d'utilisation :

```

ANALYZE (BUFFER_USAGE_LIMIT '4MB');
VACUUM (BUFFER_USAGE_LIMIT '256kB');
VACUUM (ANALYZE, BUFFER_USAGE_LIMIT 0);
vacuumdb --analyze --buffer-usage-limit=1GB --echo -d pgbench
...
VACUUM (SKIP_DATABASE_STATS, BUFFER_USAGE_LIMIT '1GB') public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS, BUFFER_USAGE_LIMIT '1GB') pg_catalog.pg_proc;
...

```

De manière globale, on peut aussi modifier le paramètre `vacuum_buffer_usage_limit`, à 256 ko par défaut. Les machines modernes permettent de monter sans problème ce paramètre à quelques mé-

gaocets pour un gain très appréciable. La valeur 0 est envisageable dans le cas d'une plage de maintenance où une purge du cache de PostgreSQL n'aurait pas de gros impact.

Les verrous, SKIP_LOCKED et lock_timeout :

À partir de la version 12, l'option `SKIP_LOCKED` permet d'ignorer toute table pour laquelle la commande `VACUUM` ne peut pas obtenir immédiatement son verrou. Cela évite de bloquer le `VACUUM` sur une table, et permet d'éviter un empilement des verrous derrière celui que le `VACUUM` veut poser, surtout en cas de `VACUUM FULL`. La commande passe alors à la table suivante à traiter. Exemple :

```
VACUUM (FULL, SKIP_LOCKED) t_un_million_int, t_cent_mille_int ;
```

```
WARNING: skipping vacuum of "t_un_million_int" --- lock not available
VACUUM
```

Une technique un peu différente est de paramétrer dans la session un petit délai avant abandon en cas de problème de verrou. Là encore, cela vise à limiter les empilements de verrou sur une base active. Par contre, comme l'ordre tombe immédiatement en erreur après le délai, il est plus adapté aux ordres ponctuels sur une table.

```
SET lock_timeout TO '100ms' ;
-- Un LOCK TABLE a été fait dans une autre session
VACUUM (verbose) pgbench_history,pgbench_tellers;
ERROR: canceling statement due to lock timeout
Durée : 1000,373 ms (00:01,000)
RESET lock_timeout ;
```

1.4.3 Options pour un VACUUM en urgence



- `INDEX_CLEANUP` `off`
- `PROCESS_TOAST` `off` (v14+)
- `TRUNCATE` `off`

Ces options sont surtout destinées à désactiver certaines étapes d'un `VACUUM` quand le temps presse vraiment.

INDEX_CLEANUP :

L'option `INDEX_CLEANUP` (par défaut à `on` jusque PostgreSQL 13 compris) déclenche systématiquement le nettoyage des index. La commande `VACUUM` va supprimer les enregistrements de l'index qui pointent vers des lignes mortes de la table. Quand il faut nettoyer des lignes mortes urgemment dans une grosse table, la valeur `off` fait gagner beaucoup de temps :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) unetable ;
```

Les index peuvent être nettoyés plus tard lors d'un autre `VACUUM`, ou reconstruits (`REINDEX`).

Cette option existe aussi sous la forme d'un paramètre de stockage (`vacuum_index_cleanup`) propre à la table pour que l'autovacuum en tienne aussi compte.

En version 14, le nouveau défaut est `auto`, qui indique que PostgreSQL décide de faire ou non le nettoyage des index suivant la quantité d'entrées à nettoyer. Il faut au minimum 2 % d'éléments à nettoyer pour que le nettoyage ait lieu.

PROCESS_TOAST :

À partir de la version 14, cette option active ou non le traitement de la partie TOAST associée à la table (parfois la partie la plus volumineuse d'une table). Elle est activée par défaut. Son utilité est la même que pour `INDEX_CLEANUP`.

```
VACUUM (VERBOSE, PROCESS_TOAST off) unetable ;
```

TRUNCATE :

L'option `TRUNCATE` (à `on` par défaut) permet de tronquer les derniers blocs vides d'une table. `TRUNCATE off` évite d'avoir à poser un verrou exclusif certes court, mais parfois gênant.

Cette option existe aussi sous la forme d'un paramètre de stockage de table (`vacuum_truncate`).

1.4.4 Autres options de VACUUM



- `VERBOSE`
- Ponctuellement :
 - `DISABLE_PAGE_SKIPPING`

VERBOSE :

Cette option affiche un grand nombre d'informations sur ce que fait la commande. En général, c'est une bonne idée de l'activer :

```
VACUUM (VERBOSE) pgbench_accounts_5 ;
```

```
INFO: vacuuming "public.pgbench_accounts_5"
INFO: scanned index "pgbench_accounts_5_pkey" to remove 9999999 row versions
DÉTAIL : CPU: user: 12.16 s, system: 0.87 s, elapsed: 18.15 s
INFO: "pgbench_accounts_5": removed 9999999 row versions in 163935 pages
DÉTAIL : CPU: user: 0.16 s, system: 0.00 s, elapsed: 0.20 s
INFO: index "pgbench_accounts_5_pkey" now contains 100000000 row versions in 301613
↪ pages
DÉTAIL : 9999999 index row versions were removed.
```

```
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO: "pgbench_accounts_5": found 10000001 removable,
      100000051 nonremovable row versions in 327870 out of 1803279 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 1071186825
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 1475409 frozen pages.
0 pages are entirely empty.
CPU: user: 13.77 s, system: 0.89 s, elapsed: 19.81 s.
VACUUM
```

DISABLE_PAGE_SKIPPING :

Par défaut, PostgreSQL ne traite que les blocs modifiés depuis le dernier `VACUUM`, ce qui est un gros gain en performance (l'information est stockée dans la *Visibility Map*, qui est généralement un tout petit fichier).

Activer l'option `DISABLE_PAGE_SKIPPING` force l'analyse de tous les blocs de la table. La table est intégralement reparcourue. Ce peut être utile en cas de problème, notamment pour reconstruire cette *Visibility Map*.

Mélange des options :

Il est possible de mélanger toutes ces options presque à volonté, et de préciser plusieurs tables à nettoyer :

```
VACUUM (VERBOSE, ANALYZE, INDEX_CLEANUP off, TRUNCATE off,
        DISABLE_PAGE_SKIPPING) bigtable, smalltable ;
```

1.5 SUIVI DU VACUUM



- `pg_stat_activity` ou `top`
- La table est-elle suffisamment nettoyée ?
- Vue `pg_stat_user_tables`
 - `last_vacuum` / `last_autovacuum`
 - `last_analyze` / `last_autoanalyze`
- `log_autovacuum_min_duration`

Un `VACUUM`, y compris lancé par l'autovacuum, apparaît dans `pg_stat_activity` et le processus est visible comme processus système avec `top` ou `ps` :

```
$ ps faux
...
postgres 3470724 0.0 0.0 12985308 6544 ? Ss 13:58 0:02 \_ postgres: 13/main:
  ↪ autovacuum launcher
postgres 795432 7.8 0.0 14034140 13424 ? Rs 16:22 0:01 \_ postgres: 13/main:
  ↪ autovacuum worker
                                                    pgbench1000p10
...
```

Il est fréquent de se demander si l'autovacuum s'occupe suffisamment d'une table qui grossit ou dont les statistiques semblent périmées. La vue `pg_stat_user_tables` contient quelques informations. Dans l'exemple ci-dessous, nous distinguons les dates des `VACUUM` et `ANALYZE` déclenchés automatiquement ou manuellement (en fait par l'application `pgbench`). Si 44 305 lignes ont été modifiées depuis le rafraîchissement des statistiques, il reste 2,3 millions de lignes mortes à nettoyer (contre 10 millions vivantes).

```
# SELECT * FROM pg_stat_user_tables WHERE relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]-----+-----
reloid          | 489050
schemaname      | public
relname         | pgbench_accounts
seq_scan        | 1
seq_tup_read    | 10
idx_scan        | 686140
idx_tup_fetch   | 2686136
n_tup_ins       | 0
n_tup_upd       | 2343090
n_tup_del       | 452
n_tup_hot_upd   | 118551
n_live_tup      | 10044489
n_dead_tup      | 2289437
```

n_mod_since_analyze	44305
n_ins_since_vacuum	452
last_vacuum	2020-01-06 18:42:50.237146+01
last_autovacuum	2020-01-07 14:30:30.200728+01
last_analyze	2020-01-06 18:42:50.504248+01
last_autoanalyze	2020-01-07 14:30:39.839482+01
vacuum_count	1
autovacuum_count	1
analyze_count	1
autoanalyze_count	1

Activer le paramètre `log_autovacuum_min_duration` avec une valeur relativement faible (dépendant des tables visées de la taille des logs générés), voire le mettre à 0, est également courant et conseillé.

1.5.1 Progression du VACUUM



- Pour `VACUUM simple` / `VACUUM FREEZE`
 - vue `pg_stat_progress_vacuum`
 - blocs parcourus / nettoyés
 - nombre de passes dans l'index
- Partie `ANALYZE`
 - `pg_stat_progress_analyze` (v13)
- Manuel ou via autovacuum
- Pour `VACUUM FULL`
 - vue `pg_stat_progress_cluster` (v12)

La vue `pg_stat_progress_vacuum` contient une ligne par `VACUUM` (simple ou `FREEZE`) en cours d'exécution.

Voici un exemple :

```
SELECT * FROM pg_stat_progress_vacuum ;
```

```
-[ RECORD 1 ]-----+-----
pid          | 4299
datid        | 13356
datname      | postgres
reloid       | 16384
phase        | scanning heap
```

```

heap_blks_total      | 127293
heap_blks_scanned   | 86665
heap_blks_vacuumed  | 86664
index_vacuum_count  | 0
max_dead_tuples     | 291
num_dead_tuples     | 53

```

Dans cet exemple, le `VACUUM` exécuté par le PID 4299 a parcouru 86 665 blocs (soit 68 % de la table), et en a traité 86 664.

Dans le cas d'un `VACUUM ANALYZE`, la seconde partie de recueil des statistiques pourra être suivie dans `pg_stat_progress_analyze` (à partir de PostgreSQL 13) :

```
SELECT * FROM pg_stat_progress_analyze ;
```

```

-[ RECORD 1 ]-----+-----
pid          | 1938258
datid        | 748619
datname      | grossetable
relid        | 748698
phase        | acquiring inherited sample rows
sample_blks_total | 1875
sample_blks_scanned | 1418
ext_stats_total | 0
ext_stats_computed | 0
child_tables_total | 16
child_tables_done | 6
current_child_table_relid | 748751

```

Les vues précédentes affichent aussi bien les opérations lancées manuellement que celles décidées par l'autovacuum.

Par contre, pour un `VACUUM FULL`, il faudra suivre la progression au travers de la vue `pg_stat_progress_cluster`, qui renvoie par exemple :

```
$ psql -c 'VACUUM FULL big' &
```

```

$ psql
postgres=# \x
Affichage étendu activé.

```

```
postgres=# SELECT * FROM pg_stat_progress_cluster ;
```

```

-[ RECORD 1 ]-----+-----
pid          | 21157
datid        | 13444
datname      | postgres
relid        | 16384
command      | VACUUM FULL
phase        | seq scanning heap
cluster_index_relid | 0
heap_tuples_scanned | 13749388
heap_tuples_written | 13749388
heap_blks_total | 199105
heap_blks_scanned | 60839
index_rebuild_count | 0

```

Cette vue est utilisable aussi avec l'ordre `CLUSTER`, d'où le nom.

1.6 AUTOVACUUM



- Processus autovacuum
- But : ne plus s'occuper de `VACUUM`
- Suit l'activité
- Seuil dépassé => worker dédié
- Gère : `VACUUM`, `ANALYZE`, `FREEZE`
 - mais pas `FULL`

Le principe est le suivant :

Le démon `autovacuum launcher` s'occupe de lancer des *workers* régulièrement sur les différentes bases. Ce nouveau processus inspecte les statistiques sur les tables (vue `pg_stat_all_tables`) : nombres de lignes insérées, modifiées et supprimées. Quand certains seuils sont dépassés sur un objet, le *worker* effectue un `VACUUM`, un `ANALYZE`, voire un `VACUUM FREEZE` (mais jamais, rappelons-le, un `VACUUM FULL`).

Le nombre de ces *workers* est limité, afin de ne pas engendrer de charge trop élevée.

1.6.1 Paramétrage du déclenchement de l'autovacuum



- `autovacuum` (on !)
- `autovacuum_naptime` (1 min)
- `autovacuum_max_workers` (3)
 - plusieurs *workers* simultanés sur une base
 - un seul par table

`autovacuum` (on par défaut) détermine si l'autovacuum doit être activé.



Il est fortement conseillé de laisser `autovacuum` à `on` !

S'il le faut vraiment, il est possible de désactiver l'autovacuum sur une table précise :

```
ALTER TABLE nom_table SET (autovacuum_enabled = off);
```

mais cela est très rare. La valeur `off` n'empêche pas le déclenchement d'un `VACUUM FREEZE` s'il devient nécessaire.

`autovacuum_naptime` est le temps d'attente entre deux périodes de vérification sur la même base (1 minute par défaut). Le déclenchement de l'autovacuum suite à des modifications de tables n'est donc pas instantané.

`autovacuum_max_workers` est le nombre maximum de *workers* que l'autovacuum pourra déclencher simultanément, chacun s'occupant d'une table (ou partition de table). Chaque table ne peut être traitée simultanément que par un unique *worker*. La valeur par défaut (3) est généralement suffisante. Néanmoins, s'il y a fréquemment trois *autovacuum workers* travaillant en même temps, et surtout si cela dure, il peut être nécessaire d'augmenter ce paramètre. Cela est fréquent quand il y a de nombreuses petites tables. Noter qu'il faudra sans doute augmenter les ressources allouées au nettoyage (paramètre `autovacuum_vacuum_cost_limit`, voir plus bas), car les *workers* se les partagent.

1.6.2 Déclenchement de l'autovacuum



Seuil de déclenchement =
 $threshold + scale\ factor \times nb\ lignes\ de\ la\ table$

L'autovacuum déclenche un `VACUUM` ou un `ANALYZE` à partir de seuils calculés sur le principe d'un nombre de lignes minimal (*threshold*) et d'une proportion de la table existante (*scale factor*) de lignes modifiées, insérées ou effacées. (Pour les détails précis sur ce qui suit, voir la documentation officielle¹.)

Ces seuils pourront être adaptés table par table.

¹<https://docs.postgresql.fr/current/routine-vacuuming.html#AUTOVACUUM>

1.6.3 Déclenchement de l'autovacuum (suite)



- Pour `VACUUM`
 - `autovacuum_vacuum_scale_factor` (20 %)
 - `autovacuum_vacuum_threshold` (50)
 - (v13) `autovacuum_vacuum_insert_threshold` (1000)
 - (v13) `autovacuum_vacuum_insert_scale_factor` (20 %)
- Pour `ANALYZE`
 - `autovacuum_analyze_scale_factor` (10 %)
 - `autovacuum_analyze_threshold` (50)

- Adapter pour une grosse table :

```
ALTER TABLE table_name SET (autovacuum_vacuum_scale_factor = 0.1);
```

Pour le `VACUUM`, si on considère les enregistrements morts (supprimés ou anciennes versions de lignes), la condition de déclenchement est :

```
nb_enregistrements_morts (pg_stat_all_tables.n_dead_tup) >=
  autovacuum_vacuum_threshold
  + autovacuum_vacuum_scale_factor × nb_enregs (pg_class.reltuples)
```

où, par défaut :

- `autovacuum_vacuum_threshold` vaut 50 lignes ;
- `autovacuum_vacuum_scale_factor` vaut 0,2 soit 20 % de la table.

Donc, par exemple, dans une table d'un million de lignes, modifier 200 050 lignes provoquera le passage d'un `VACUUM`.

Pour les grosses tables avec de l'historique, modifier 20 % de la volumétrie peut être extrêmement long. Quand l'autovacuum lance enfin un `VACUUM`, celui-ci a donc beaucoup de travail et peut durer longtemps et générer beaucoup d'écritures. Il est donc fréquent de descendre la valeur de `autovacuum_vacuum_scale_factor` à quelques pour cent sur les grosses tables. (Une alternative est de monter `autovacuum_vacuum_threshold` à un nombre de lignes élevé et de descendre `autovacuum_vacuum_scale_factor` à 0, mais il faut alors calculer le nombre de lignes qui déclenchera le nettoyage, et cela dépend fortement de la table et de sa fréquence de mise à jour.)

S'il faut modifier un paramètre, il est préférable de ne pas le faire au niveau global mais de cibler les tables où cela est nécessaire. Par exemple, l'ordre suivant réduit à 5 % de la table le nombre de lignes

à modifier avant que l'`autovacuum` y lance un `VACUUM` :

```
ALTER TABLE nom_table SET (autovacuum_vacuum_scale_factor = 0.05);
```

À partir de PostgreSQL 13, le `VACUUM` est aussi lancé quand il n'y a que des insertions, avec deux nouveaux paramètres et un autre seuil de déclenchement :

```
nb_enregistrements_insérés (pg_stat_all_tables.n_ins_since_vacuum) >=
  autovacuum_vacuum_insert_threshold
  + autovacuum_vacuum_insert_scale_factor × nb_enregs (pg_class.reltuples)
```

Pour l'`ANALYZE`, le principe est le même. Il n'y a que deux paramètres, qui prennent en compte toutes les lignes modifiées *ou* insérées, pour calculer le seuil :

```
nb_insert + nb_updates + nb_delete (n_mod_since_analyze) >=
  autovacuum_analyze_threshold + nb_enregs × autovacuum_analyze_scale_factor
```

où, par défaut :

- `autovacuum_analyze_threshold` vaut 50 lignes ;
- `autovacuum_analyze_scale_factor` vaut 0,1, soit 10 %.

Dans notre exemple d'une table, modifier 100 050 lignes provoquera le passage d'un `ANALYZE`.

Là encore, il est fréquent de modifier les paramètres sur les grosses tables pour rafraîchir les statistiques plus fréquemment.



Les insertions ont toujours été prises en compte pour `ANALYZE`, puisqu'elles modifient le contenu de la table. Par contre, jusque PostgreSQL 12 inclus, `VACUUM` ne tenait pas compte des lignes insérées pour déclencher son nettoyage. Or, cela avait des conséquences pour les tables à insertion seule (gel de lignes retardé, *Index Only Scan* impossibles...) Pour cette raison, à partir de la version 13, les insertions sont aussi prises en compte pour déclencher un `VACUUM`.

1.7 PARAMÉTRAGE DE VACUUM & AUTOVACUUM



- VACUUM vs autovacuum
- Mémoire
- Gestion des coûts
- Gel des lignes

En fonction de la tâche exacte, de l'agressivité acceptable ou de l'urgence, plusieurs paramètres peuvent être mis en place.

Ces paramètres peuvent différer (par le nom ou la valeur) selon qu'ils s'appliquent à un `VACUUM` lancé manuellement ou par script, ou à un processus lancé par l'autovacuum.

1.7.1 VACUUM vs autovacuum

VACUUM manuel	autovacuum
Urgent	Arrière-plan
Pas de limite	Peu agressif
Paramètres	Les mêmes + paramètres de surcharge

Quand on lance un ordre `VACUUM`, il y a souvent urgence, ou l'on est dans une période de maintenance, ou dans un batch. Les paramètres que nous allons voir ne cherchent donc pas, par défaut, à économiser des ressources.

À l'inverse, un `VACUUM` lancé par l'autovacuum ne doit pas gêner une production peut-être chargée. Il existe donc des paramètres `autovacuum_*` surchargeant les précédents, et beaucoup plus conservateurs.

1.7.2 Mémoire



- Quantité de mémoire allouable
 - `maintenance_work_mem` / `autovacuum_work_mem`
 - monté souvent à ½ à 1 Go
- Impact
 - `VACUUM`
 - construction d'index

`maintenance_work_mem` est la quantité de mémoire qu'un processus effectuant une opération de maintenance (c'est-à-dire n'exécutant pas des requêtes classiques comme `SELECT`, `INSERT`, `UPDATE` ...) est autorisé à allouer pour sa tâche de maintenance.

Cette mémoire est utilisée lors de la construction d'index ou l'ajout de clés étrangères. et, dans le contexte de `VACUUM`, pour stocker les adresses des enregistrements pouvant être recyclés. Cette mémoire est remplie pendant la phase 1 du processus de `VACUUM`, tel qu'expliqué plus haut.

Rappelons qu'une adresse d'enregistrement (`tid`, pour `tuple id`) a une taille de 6 octets et est composée du numéro dans la table, et du numéro d'enregistrement dans le bloc, par exemple `(0,1)`, `(3164,98)` ou `(5351510,42)`.

Le défaut de 64 Mo est assez faible. Si tous les enregistrements morts d'une table ne tiennent pas dans `maintenance_work_mem`, `VACUUM` est obligé de faire plusieurs passes de nettoyage, donc plusieurs parcours complets de chaque index. Une valeur assez élevée de `maintenance_work_mem` est donc conseillée : s'il est déjà possible de stocker plusieurs dizaines de millions d'enregistrements à effacer dans 256 Mo, 1 Go peut être utile lors de grosses purges. Attention, plusieurs `VACUUM` peuvent tourner simultanément.

Un `maintenance_work_mem` à plus de 1 Go est inutile pour le `VACUUM` (il ne sait pas utiliser plus), par contre il peut accélérer l'indexation de grosses tables.

`autovacuum_work_mem` permet de surcharger `maintenance_work_mem` spécifiquement pour l'autovacuum. Par défaut les deux sont identiques.

1.7.3 Bridage du VACUUM et de l'autovacuum



- Pauses régulières après une certaine activité
- Par bloc traité
 - `vacuum_cost_page_hit` / `_miss` / `_dirty` (1/ 10 ou 2 /20)
 - jusque total de : `vacuum_cost_limit` (200)
 - pause : `vacuum_cost_delay` (en manuel : 0 ms !)
- Surcharge pour l'autovacuum
 - `autovacuum_vacuum_cost_limit` (identique)
 - `autovacuum_vacuum_cost_delay` (2 ms)
 - => débit en écriture max : ~40 Mo/s
- Pour accélérer : augmenter la limite

Principe :

Les paramètres suivant permettent d'éviter qu'un `VACUUM` ne gêne les autres sessions en saturant le disque. Le principe est de provoquer une pause après qu'une certaine activité a été réalisée.

Paramètres de coûts :

Ces trois paramètres « quantifient » l'activité de `VACUUM`, affectant un coût arbitraire à chaque fois qu'une opération est réalisée :

- `vacuum_cost_page_hit` : coût d'accès à chaque page présente dans le cache de PostgreSQL (valeur : 1) ;
- `vacuum_cost_page_miss` : coût d'accès à chaque page hors de ce cache (valeur : 2 à partir de la PostgreSQL 14, 10 auparavant) ;
- `vacuum_cost_page_dirty` : coût de modification d'une page, et donc de son écriture (valeur : 20).

Il est déconseillé de modifier ces paramètres de coût.

Pause :

Quand le coût cumulé atteint un seuil, l'opération de nettoyage marque une pause. Elle est gouvernée par deux paramètres :

- `vacuum_cost_limit` : coût cumulé à atteindre avant de déclencher la pause (défaut : 200) ;
- `vacuum_cost_delay` : temps à attendre (défaut : 0 ms !)

En conséquence, les `VACUUM` lancés manuellement (en ligne de commande ou via `vacuumdb`) ne sont **pas** freinés par ce mécanisme et peuvent donc entraîner de fortes écritures ! Mais c'est généralement ce que l'on veut dans un batch ou en urgence, et il vaut mieux alors être le plus rapide possible.

(Pour les urgences, rappelons que l'option `INDEX_CLEANUP off` ou `PROCESS_TOAST off` permettent aussi d'ignorer le nettoyage des index ou des TOAST.)

Paramétrage pour le VACUUM manuel :

Il est conseillé de ne pas toucher au paramétrage par défaut de `vacuum_cost_limit` et `vacuum_cost_delay`.

Si on doit lancer un `VACUUM` manuellement en limitant son débit, procéder comme suit dans une session :

```
-- Reprise pour le VACUUM du paramétrage d'autovacuum
```

```
SET vacuum_cost_limit = 200 ;
```

```
SET vacuum_cost_delay = '2ms' ;
```

```
VACUUM (VERBOSE) matable ;
```

Avec `vacuumdb`, il faudra passer par la variable d'environnement `PGOPTIONS`.

Paramétrage pour l'autovacuum :

Les `VACUUM` d'autovacuum, eux, sont par défaut limités en débit pour ne pas gêner l'activité normale de l'instance. Deux paramètres surchargent les précédents :

- `autovacuum_cost_limit` vaut par défaut -1, donc reprend la valeur 200 de `vacuum_cost_limit` ;
- `autovacuum_vacuum_cost_delay` vaut par défaut 2 ms.

Un (`autovacuum_`) `vacuum_cost_limit` à 200 consiste à traiter au plus 200 blocs lus en cache (car `vacuum_cost_page_hit` = 1), soit 1,6 Mo, avant de faire la pause de 2 ms. Si ces blocs doivent être écrits, on descend en-dessous de 10 blocs traités avant chaque pause (`vacuum_cost_page_dirty` = 20) avant la pause de 2 ms, d'où un débit en écriture maximal de l'autovacuum de 40 Mo/s. Cela s'observe aisément par exemple avec `iotop`.

Ce débit est partagé équitablement entre les différents *workers* lancés par l'autovacuum (sauf paramétrage spécifique au niveau de la table).

Pour rendre l'`autovacuum` plus rapide, il est préférable d'augmenter `autovacuum_vacuum_cost_limit` au-delà de 200, plutôt que de réduire `autovacuum_vacuum_cost_delay` qui n'est qu'à 2 ms, pour ne pas monopoliser le disque. (Exception : les versions antérieures à la 12, car `autovacuum_vacuum_cost_delay` valait alors 20 ms et le débit en écriture saturait à 4 Mo/s ! La valeur 2 ms tient mieux compte des disques actuels.)

La prise en compte de la nouvelle valeur de la limite par les *workers* en cours sur les tables est automatique à partir de PostgreSQL 16. Dans les versions précédentes, il faut arrêter les *workers* en cours (avec `pg_cancel_backend()`) et attendre que l'`autovacuum` les relance. Quand `autovacuum_max_workers` est augmenté, prévoir aussi d'augmenter la limite. Il faut pouvoir assumer le débit supplémentaire pour les disques.

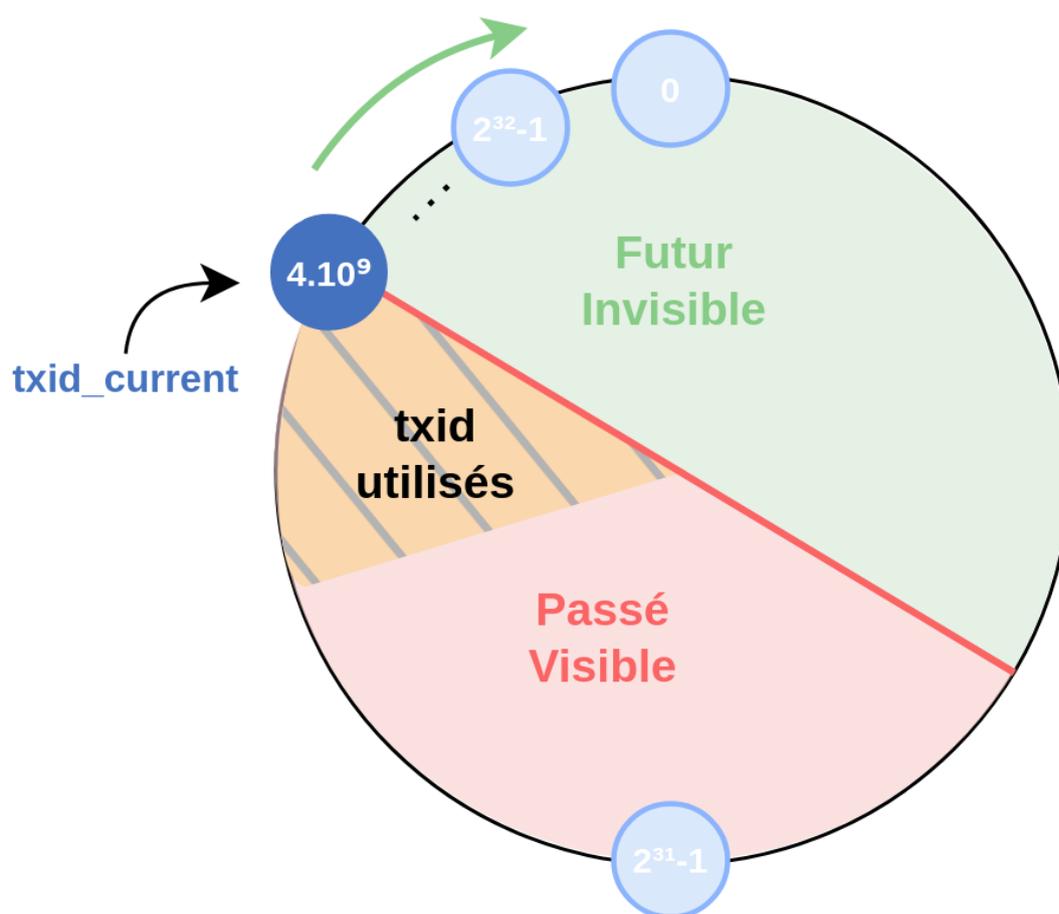
Sur le sujet, voir la conférence de Robert Haas à PGconf.EU 2023 à Prague².

²<https://www.postgresql.eu/events/pgconfeu2023/sessions/session/4847/slides/432/Understanding%20and%20Fixing%20Autovacuum%20-%20PGCONF.EU%202023.pdf#page=14>

1.7.4 Paramétrage du FREEZE (1)



Le but est de geler les numéros de transaction assez vite :



Rappelons que les numéros de transaction stockés sur les lignes ne sont stockés que sur 32 bits, et sont recyclés. Il y a donc un risque de mélanger l'avenir et le futur des transactions lors du rebouclage (*wraparound*). Afin d'éviter ce phénomène, l'opération `VACUUM FREEZE` « gèle » les vieux enregistrements, afin que ceux-ci ne se retrouvent pas brusquement dans le futur.

Concrètement, il s'agit de positionner un *hint bit* dans les entêtes des lignes concernées, indiquant qu'elle est plus vieille que tous les numéros de transactions actuellement actifs. (Avant la 9.4, la colonne système `xmin` était simplement remplacée par un `FrozenXid`, ce qui revient au même).

1.7.5 Paramétrage du FREEZE (2)



Quand le `VACUUM` gèle-t-il les lignes ?

- « Âge » d'une table : `age (pgclass.relrozenxid)`
 - Les blocs nettoyés/gelés sont notés dans la *visibility map*
- `vacuum_freeze_min_age` (50 Mtrx)
 - âge des lignes *rencontrées* à geler
- `vacuum_freeze_table_age` (150 Mtrx)
 - agressif (toute la table)
- Au plus tard, par l'autovacuum sur toute la table :
 - `autovacuum_freeze_max_age` (200 Mtrx)
- Attention après des imports en masse !
 - `VACUUM FREEZE` préventif en période de maintenance

Geler une ligne ancienne implique de réécrire le bloc et donc des écritures dans les journaux de transactions et les fichiers de données. Il est inutile de geler trop tôt une ligne récente, qui sera peut-être bientôt réécrite.

Paramétrage :

Plusieurs paramètres règlent ce fonctionnement. Leurs valeurs par défaut sont satisfaisantes pour la plupart des installations et ne sont pour ainsi dire jamais modifiées. Par contre, il est important de bien connaître le fonctionnement pour ne pas être surpris.

Le numéro de transaction le plus ancien connu au sein d'une table est porté par `pgclass.relrozenxid`, et est sur 32 bits. Il faut utiliser la fonction `age()` pour connaître l'écart par rapport au numéro de transaction courant (géré sur 64 bits en interne).

```
SELECT relname, relrozenxid, round(age(relrozenxid) /1e6,2) AS "age_Mtrx"
FROM pg_class c
WHERE relname LIKE 'pgbench%' AND relkind='r'
ORDER BY age(relrozenxid) ;
```

relname	relrozenxid	age_Mtrx
pgbench_accounts_7	882324041	0.00
pgbench_accounts_8	882324041	0.00
pgbench_accounts_2	882324041	0.00
pgbench_history	882324040	0.00
pgbench_accounts_5	848990708	33.33

pgbench_tellers	832324041	50.00
pgbench_accounts_3	719860155	162.46
pgbench_accounts_9	719860155	162.46
pgbench_accounts_4	719860155	162.46
pgbench_accounts_6	719860155	162.46
pgbench_accounts_1	719860155	162.46
pgbench_branches	719860155	162.46
pgbench_accounts_10	719860155	162.46

Une partie du gel se fait lors d'un `VACUUM` normal. Si ce dernier rencontre un enregistrement plus vieux que `vacuum_freeze_min_age` (par défaut 50 millions de transactions écoulées), alors le *tuple* peut et doit être gelé. Cela ne concerne que les lignes dans des blocs qui ont des lignes mortes à nettoyer : les lignes dans des blocs un peu statiques y échappent. (Y échappent aussi les lignes qui ne sont pas forcément visibles par toutes les transactions ouvertes.)

`VACUUM` doit donc périodiquement déclencher un nettoyage plus agressif de toute la table (et non pas uniquement des blocs modifiés depuis le dernier `VACUUM`), afin de nettoyer tous les vieux enregistrements. C'est le rôle de `vacuum_freeze_table_age` (par défaut 150 millions de transactions). Si la table a atteint cet âge, un `VACUUM` (manuel ou automatique) lancé dessus deviendra « agressif » :

```
VACUUM (VERBOSE) pgbench_tellers ;
INFO: aggressively vacuuming "public.pgbench_tellers"
```

C'est équivalent à l'option `DISABLE_PAGE_SKIPPING` : les blocs ne contenant que des lignes vivantes seront tout de même parcourus. Les lignes non gelées qui s'y trouvent et plus vieilles que `vacuum_freeze_min_age` seront alors gelées. Ce peut être long, ou pas, en fonction de l'efficacité de l'étape précédente.

À côté des numéros de transaction habituels, les identifiants `multixact`, utilisés pour supporter le verrouillage de lignes par des transactions multiples évitent aussi le *wraparound* avec des paramètres spécifiques (`vacuum_multixact_freeze_min_age`, `vacuum_multixact_freeze_table_age`) qui ont les mêmes valeurs que leurs homologues.

Enfin, il faut traiter le cas de tables sur lesquelles un `VACUUM` complet ne s'est pas déclenché depuis très longtemps. L'autovacuum y veille : `autovacuum_freeze_max_age` (par défaut 200 millions de transactions) est l'âge maximum que doit avoir une table. S'il est dépassé, un `VACUUM` agressif est automatiquement lancé sur cette table. Il est visible dans `pg_stat_activity` avec la mention caractéristique *to prevent wraparound* :

```
autovacuum: VACUUM public.pgbench_accounts (to prevent wraparound)
```



Ce traitement est lancé même si `autovacuum` est désactivé (c'est-à-dire à `off`).

En fait, un `VACUUM FREEZE` lancé manuellement équivaut à un `VACUUM` avec les paramètres `vacuum_freeze_table_age` (âge minimal de la table) et `vacuum_freeze_min_age` (âge minimal des lignes pour les geler) à 0. Il va geler toutes les lignes qui le peuvent, même « jeunes ».

Charge induite par le gel :

Le gel des lignes peut être très lourd s'il y a beaucoup de lignes à geler, ou très rapide si l'essentiel du travail a été fait par les nettoyages précédents. Si la table a déjà été entièrement gelée (parfois depuis des centaines de millions de transactions), il peut juste s'agir d'une mise à jour du `relfrozenxid`.

Les blocs déjà entièrement gelés sont recensés dans la *visibility map* (qui recense aussi les blocs sans ligne morte). ils ne seront pas reparcourus s'ils ne sont plus modifiés. Cela accélère énormément le `FREEZE` sur les grosses tables (avant PostgreSQL 9.6, il y avait forcément au moins un parcours complet de la table !) Si le `VACUUM` est interrompu, ce qu'il a déjà gelé n'est pas perdu, il ne faut donc pas hésiter à l'interrompre au besoin.

L'âge de la table peut dépasser `autovacuum_freeze_max_age` si le nettoyage est laborieux, ce qui explique la marge par rapport à la limite fatidique des 2 milliards de transactions.

Quelques problèmes possibles sont évoqués plus bas.

Âge d'une base :

Nous avons vu que l'âge d'une base est en fait l'âge de la table la plus ancienne, qui se calcule à partir de la colonne `pg_database.datfrozenxid` :

```
SELECT datname, datfrozenxid, age (datfrozenxid)
FROM pg_database ORDER BY 3 DESC ;
```

datname	datfrozenxid	age
pgbench	1809610092	149835222
template0	1957896953	1548361
template1	1959012415	432899
postgres	1959445305	9

Concrètement, on verra l'âge d'une base de données approcher peu à peu des 200 millions de transactions, ce qui correspondra à l'âge des plus « vieilles » tables, souvent celles sur lesquelles l'autovacuum ne passe jamais. L'âge des tables évolue même si l'essentiel de leur contenu, voire la totalité, est déjà gelé (car il peut rester le `pg_class.relfrozenxid` à mettre à jour, ce qui sera bien sûr très rapide). Cet âge va retomber quand un gel sera forcé sur ces tables, puis remonter, etc.

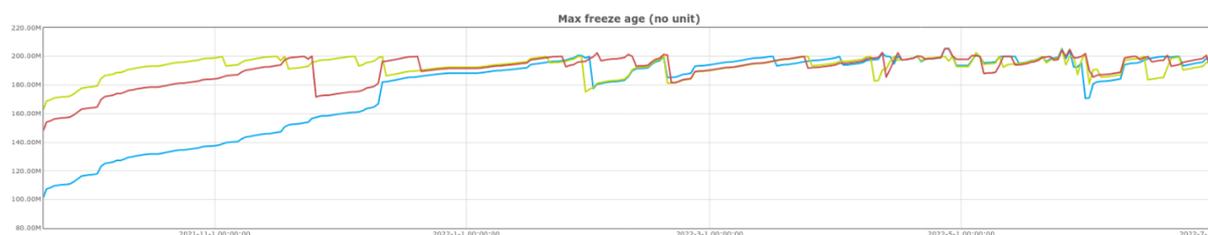


Figure 1/ .4: Évolution de l'âge des bases sur une instance



Rappelons que le `FREEZE` génère de fait la réécriture de tous les blocs concernés. Il peut être quasi-instantané, mais le déclenchement inopiné d'un `VACUUM FREEZE` sur l'intégralité d'une grosse table assez statique est une mauvaise surprise assez fréquente.

Une base chargée avec `pg_restore` et peu modifiée peut même voir le `FREEZE` se déclencher sur toutes les tables en même temps. Cela est moins grave depuis les optimisations de la 9.6, mais, après de très gros imports, il reste utile d'opérer un `VACUUM FREEZE` manuel, à un moment où cela gêne peu, pour éviter qu'ils ne se provoquent plus tard en période chargée.

Résumé :

Que retenir de ce paramétrage complexe ?

- le `VACUUM` gèlera une partie des lignes un peu anciennes lors de son fonctionnement habituel ;
- un bloc gelé non modifié ne sera plus à regeler ;
- de grosses tables statiques peuvent engendrer soudainement une grosse charge en écriture ; il vaut mieux être proactif.

1.8 AUTRES PROBLÈMES COURANTS

1.8.1 L'autovacuum dure trop longtemps



- Fréquence de passage ?
- Débit ?
- Nombre de workers ?
- Taille vraiment trop grosse ?

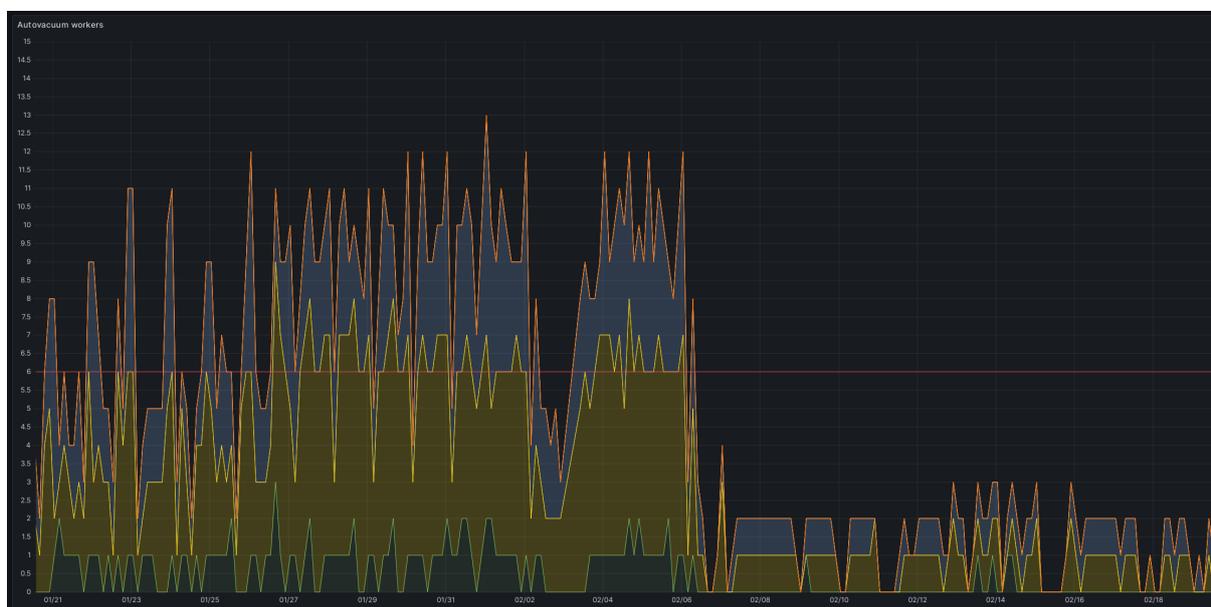


Figure 1/ .5: Nombre de workers avant et après la réduction d'autovacuum_vacuum_cost_delay

Nous avons vu que le paramétrage de l'autovacuum vise à limiter la charge sur la base. Le nettoyage d'une grosse table peut donc être parfois très long. Ce n'est pas forcément un problème majeur si l'opération arrive à terme dans un délai raisonnable, mais il vaut mieux savoir pourquoi. Il peut y avoir plusieurs causes, qui ne s'excluent pas mutuellement.

Il est fréquent que les grosses tables soient visitées trop peu souvent. Rappelons que la propriété `autovacuum_vacuum_scale_factor` de chaque table est par défaut à 20 % : lorsque l'autovacuum se déclenche, il doit donc traiter une volumétrie importante. Il est conseillé de réduire la valeur de ce paramètre (ou de jouer sur `autovacuum_vacuum_threshold`) pour un nettoyage plus fréquent.

Le débit en écriture peut être insuffisant (c'est fréquent sur les anciennes versions), auquel cas, avec des disques corrects, on peut baisser `autovacuum_vacuum_cost_delay` ou monter

`autovacuum_vacuum_cost_limit`. Sur le graphique ci-dessus, issu d'un cas réel, les trois workers semblaient en permanence occupés. Il risquait donc d'y avoir un retard pour nettoyer certaines tables, ou calculer des statistiques. La réduction de `autovacuum_vacuum_cost_delay` de 20 à 2 ms a mené à une réduction drastique de la durée de traitement de chaque worker.

Rappelons qu'un `VACUUM` manuel (ou planifié) n'est soumis à aucun bridage.

Le nombre de workers peut être trop bas, notamment s'il y a de nombreuses tables. Auquel cas ils semblent tous activés en permanence (comme ci-dessus). Monter `autovacuum_max_workers` au-delà de 3 nécessite d'augmenter le débit autorisé avec les paramètres ci-dessus.

Pour des grandes tables, le partitionnement permet de paralléliser l'activité de l'autovacuum. Les workers peuvent en effet travailler dans la même base, sur des tables différentes.

Un grand nombre de bases actives peut devenir un frein et augmenter l'intervalle entre deux nettoyages d'une base, bien que l'`autovacuum_launcher` ignore les bases inutilisées.

Exceptionnellement, l'autovacuum peut tomber en erreur (bloc corrompu, index fonctionnel avec une fonction boguée...) et ne jamais finir (surveiller les traces).

1.8.2 Arrêter un VACUUM ?



- Lancement manuel ou script
 - risque avec certains verrous
- Autovacuum
 - interrompre s'il gêne
- Exception : *to prevent wraparound* lent **et** bloquant
 - `pg_cancel_backend` + `VACUUM FREEZE` manuel

Le cas des `VACUUM` manuels a été vu plus haut : ils peuvent gêner quelques verrous ou opérations DDL. Il faudra les arrêter manuellement au besoin.

C'est différent si l'autovacuum a lancé le processus : celui-ci sera arrêté si un utilisateur pose un verrou en conflit.

La seule exception concerne un `VACUUM FREEZE` lancé quand la table doit être gelée, donc avec la mention *to prevent wraparound* dans `pg_stat_activity` : celui-ci ne sera pas interrompu. Il ne pose qu'un verrou destinée à éviter les modifications de schéma simultanées (SHARE UPDATE EXCLUSIVE). Comme le débit en lecture et écriture est bridé par le paramétrage habituel de l'autovacuum, ce verrou peut durer assez longtemps (surtout avant PostgreSQL 9.6, où toute la table est relue à

chaque `FREEZE`). Cela peut s'avérer gênant avec certaines applications. Une solution est de réduire `autovacuum_vacuum_cost_delay`, surtout avant PostgreSQL 12 (voir plus haut).

Si les opérations sont impactées, on peut vouloir lancer soi-même un `VACUUM FREEZE` manuel, non bridé. Il faudra alors repérer le PID du `VACUUM FREEZE` en cours, l'arrêter avec `pg_cancel_backend`, puis lancer manuellement l'ordre `VACUUM FREEZE` sur la table concernée, (et rapidement avant que l'autovacuum ne relance un processus).

La supervision peut se faire avec `pg_stat_progress_vacuum` et `iotop`.

1.8.3 Ce qui peut bloquer le VACUUM FREEZE



- Causes :
 - sessions *idle in transaction* sur une longue durée
 - slot de réplication en retard/oublié
 - transactions préparées oubliées
 - erreur à l'exécution du `VACUUM`
- Conséquences :
 - processus autovacuum répétés
 - arrêt des transactions
 - mode single...
- Supervision :
 - `check_pg_activity` : `xmin`, `max_freeze_age`
 - surveillez les traces !

Il arrive que le fonctionnement du `FREEZE` soit gêné par un problème qui lui interdit de recycler les plus anciens numéros de transactions. (Ces causes gênent aussi un `VACUUM` simple, mais les symptômes sont alors surtout un gonflement des tables concernées).

Les causes possibles sont :

- des sessions *idle in transactions* durent depuis des jours ou des semaines (voir le statut `idle in transaction` dans `pg_stat_activity`, et au besoin fermer la session) : au pire, elles disparaissent après redémarrage ;
- des slots de réplication pointent vers un secondaire très en retard, voire disparu (consulter `pg_replication_slots`, et supprimer le slot) ;

- des transactions préparées (pas des requêtes préparées !) n'ont jamais été validées ni annulées, (voir `pg_prepared_xacts`, et annuler la transaction) : elles ne disparaissent pas après redémarrage ;
- l'opération de `VACUUM` tombe en erreur : corruption de table ou index, fonction d'index fonctionnel buggée, etc. (voir les traces et corriger le problème, supprimer l'objet ou la fonction, etc.).

Pour effectuer le `FREEZE` en urgence le plus rapidement possible, on peut utiliser, à partir de PostgreSQL 12 :

```
VACUUM (FREEZE, VERBOSE, INDEX_CLEANUP off, TRUNCATE off) ;
```

Cette commande force le gel de toutes les lignes, ignore le nettoyage des index et ne supprime pas les blocs vides finaux (le verrou peut être gênant). Un `VACUUM` classique serait à prévoir ensuite à l'occasion.

En toute rigueur, une version sans l'option `FREEZE` est encore plus rapide : le mode agressif serait déclenché mais les lignes plus récentes que `vacuum_freeze_min_age` (50 millions de transaction) ne seraient pas encore gelées. On peut même monter ce paramètre dans la session pour alléger au maximum la charge sur une table dont les lignes ont des âges bien étalés.

Ne pas oublier de nettoyer toutes les bases de l'instance.

Dans le pire des cas, plus aucune transaction ne devient possible (y compris les opérations d'administration comme `DROP`, ou `VACUUM` sans `TRUNCATE off`) :

```
ERROR: database is not accepting commands to avoid wraparound data loss in database
↳ "db1"
HINT: Stop the postmaster and vacuum that database in single-user mode.
You might also need to commit or roll back old prepared transactions,
or drop stale replication slots.
```

En dernière extrémité, il reste un délai de grâce d'un million de transactions, qui ne sont accessibles que dans le très austère mode monutilisateur³ de PostgreSQL.

Avec la sonde Nagios `check_pgactivity`⁴, et les services `max_freeze_age` et `oldest_xmin`, il est possible de vérifier que l'âge des bases ne dérive pas, ou de trouver quel processus porte le `xmin` le plus ancien. S'il y a un problème, il entraîne généralement l'apparition de nombreux messages dans les traces : lisez-les régulièrement !

³<https://docs.postgresql.fr/current/app-postgres.html#APP-POSTGRES-SINGLE-USER>

⁴https://github.com/OPMDG/check_pgactivity

1.9 RÉSUMÉ DES CONSEILS SUR L'AUTOVACUUM

“Vacuuming is like exercising.
If it hurts, you’re not doing it enough!”

(Robert Haas, PGConf.EU 2023, Prague, 13 décembre 2023)

Certains sont frileux à l'idée de passer un `VACUUM`. En général, ce n'est que reculer pour mieux sauter.

1.9.1 Résumé des conseils sur l'autovacuum (1/2)



- Laisser l'autovacuum faire son travail
- Augmenter le débit autorisé
- Surveiller `last_(auto)analyze` / `last_(auto)vacuum`
- Nombre de *workers*
- Grosses tables, par ex :

```
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.01) ;
ALTER TABLE table_name SET (autovacuum_vacuum_threshold = 1000000) ;
```

- Mais ne pas hésiter à planifier un `vacuumdb` quotidien

L'autovacuum fonctionne convenablement pour les charges habituelles. Il ne faut pas s'étonner qu'il fonctionne longtemps en arrière-plan : il est justement conçu pour ne pas se presser. Au besoin, ne pas hésiter à lancer manuellement l'opération, donc sans bridage en débit.

Si les disques sont bons, on peut augmenter le débit autorisé :

- en réduisant la durée de pause (`autovacuum_vacuum_cost_delay`), surtout avant la version 12 ;
- de préférence, en augmentant le coût à atteindre avant une pause (`autovacuum_vacuum_cost_limit`) ;
- en augmentant `vacuum_buffer_usage_limit` (en version 16).

Comme le déclenchement d'`autovacuum` est très lié à l'activité, il faut vérifier qu'il passe assez souvent sur les tables sensibles en surveillant `pg_stat_all_tables.last_autovacuum` et `last_autoanalyze`.

Si les statistiques peinent à se rafraîchir, ne pas hésiter à activer plus souvent l'autovacuum sur les grosses tables problématiques ainsi :

```
-- analyze après 5 % de modification au lieu du défaut de 10 %
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.05) ;
```

De même, si la fragmentation s'envole, descendre `autovacuum_vacuum_scale_factor`. (On peut préférer utiliser les variantes en `*_threshold` de ces paramètres, et mettre les `*_scale_factor` à 0).

Dans un modèle avec de très nombreuses tables actives, le nombre de *workers* doit parfois être augmenté.

1.9.2 Résumé des conseils sur l'autovacuum (2/2)



- Mode manuel
 - batchs / tables temporaires / tables à insertions seules (<v13)
 - si pressé !
- Danger du `FREEZE` brutal
 - prévenir
- `VACUUM FULL` : dernière extrémité

L'autovacuum n'est pas toujours assez rapide à se déclencher, par exemple entre les différentes étapes d'un batch : on intercalera des `VACUUM ANALYZE` manuels. Il faudra le faire systématiquement pour les tables temporaires (que l'autovacuum ne voit pas). Pour les tables où il n'y a que des insertions, avant PostgreSQL 13, l'autovacuum ne lance spontanément que l'`ANALYZE` : il faudra effectuer un `VACUUM` explicite pour profiter de certaines optimisations.

Au final, un `vacuumdb` planifié quotidien à un moment calme est généralement une bonne idée : ce qu'il traite ne sera plus à nettoyer en journée, et il apporte la garantie qu'aucune table ne sera négligée. Cela ne dispense pas de contrôler l'activité de l'`autovacuum` sur les grosses tables très sollicitées.

Un point d'attention reste le gel brutal de grosses quantités de données chargées ou modifiées en même temps. Un `VACUUM FREEZE` préventif dans une période calme reste la meilleure solution.

Un `VACUUM FULL` sur une grande table est une opération très lourde, à réserver à la récupération d'une partie significative de son espace, qui ne serait pas réutilisé plus tard.

1.10 CONCLUSION



- `VACUUM` fait de plus en plus de choses au fil des versions
- Convient généralement
- Paramétrage apparemment complexe
 - en fait relativement simple avec un peu d'habitude

1.10.1 Questions



N'hésitez pas, c'est le moment !

1.11 QUIZ



https://dali.bo/m5_quiz

1.12 TRAVAUX PRATIQUES

1.12.1 Traiter la fragmentation



But : Traiter la fragmentation

Créer une table `t3` avec une colonne `id` de type `integer`.

Désactiver l'autovacuum pour la table `t3`.

Insérer un million de lignes dans la table `t3` avec la fonction `generate_series`.

Récupérer la taille de la table `t3`.

Supprimer les 500 000 premières lignes de la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Créer une table `t4` avec une colonne `id` de type `integer`.

Désactiver l'autovacuum pour la table `t4`.

Insérer un million de lignes dans la table `t4` avec `generate_series`.

Récupérer la taille de la table `t4`.

Supprimer les 500 000 **dernières** lignes de la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

Exécuter un `VACUUM` sur la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

1.12.2 Détecter la fragmentation



But : Détecter la fragmentation

Créer une table `t5` avec deux colonnes : `c1` de type `integer` et `c2` de type `text`.

Désactiver l'autovacuum pour la table `t5`.

Insérer un million de lignes dans la table `t5` avec `generate_series`.

- Installer l'extension `pg_freespacemap` (documentation : <https://docs.postgresql.fr/current/pgfreespacemap.html>)
- Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?
- Modifier exactement 200 000 lignes de la table `t5`.
- Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Exécuter un `VACUUM` sur la table `t5`.

Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Récupérer la taille de la table `t5`.

Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

1.12.3 Gestion de l'autovacuum



But : Voir fonctionner l'autovacuum

Créer une table `t6` avec une colonne `id` de type `integer`.

Insérer un million de lignes dans la table `t6` :

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Il faudra peut-être attendre une minute. (Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.)

Vérifier le nombre de lignes dans `pg_class.reltuples`.

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

- Modifier encore 200 000 autres lignes de la table `t6` :

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

1.13 TRAVAUX PRATIQUES (SOLUTIONS)

1.13.1 Traiter la fragmentation

Créer une table `t3` avec une colonne `id` de type `integer`.

```
CREATE TABLE t3(id integer);
```

```
CREATE TABLE
```

Désactiver l'autovacuum pour la table `t3`.

```
ALTER TABLE t3 SET (autovacuum_enabled = false);
```

```
ALTER TABLE
```



La désactivation de l'autovacuum ici a un but uniquement pédagogique. En production, c'est une très mauvaise idée !

Insérer un million de lignes dans la table `t3` avec la fonction `generate_series`.

```
INSERT INTO t3 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

Récupérer la taille de la table `t3`.

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
pg_size_pretty
-----
35 MB
```

Supprimer les 500 000 premières lignes de la table `t3`.

```
DELETE FROM t3 WHERE id <= 500000;
```

```
DELETE 500000
```

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
pg_size_pretty
-----
35 MB
```

`DELETE` seul ne permet pas de regagner de la place sur le disque. Les lignes supprimées sont uniquement marquées comme étant mortes. Comme l'autovacuum est ici désactivé, PostgreSQL n'a pas encore nettoyé ces lignes.

Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

```
VACUUM VERBOSE t3;
```

```
INFO: vacuuming "public.t3"
INFO: "t3": removed 500000 row versions in 2213 pages
INFO: "t3": found 500000 removable, 500000 nonremovable row versions
      in 4425 out of 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 3815272
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.09 s, system: 0.00 s, elapsed: 0.10 s.
VACUUM
```

L'indication :

```
removed 500000 row versions in 2213 pages
```

indique 500 000 lignes ont été nettoyées dans 2213 blocs (en gros, la moitié des blocs de la table).

Pour compléter, l'indication suivante :

```
found 500000 removable, 500000 nonremovable row versions in 4425 out of 4425 pages
```

reprend l'indication sur 500 000 lignes mortes, et précise que 500 000 autres ne le sont pas. Les 4425 pages parcourues correspondent bien à la totalité des 35 Mo de la table complète. C'est la première fois que `VACUUM` passe sur cette table, il est normal qu'elle soit intégralement parcourue.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
pg_size_pretty
-----
35 MB
```

`VACUUM` ne permet pas non plus de gagner en espace disque. Principalement, il renseigne la structure FSM (*free space map*) sur les emplacements libres dans les fichiers des tables.

Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

```
VACUUM FULL t3;
```

```
INFO: vacuuming "public.t3"
INFO: "t3": found 0 removable, 500000 nonremovable row versions in 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.10 s, system: 0.01 s, elapsed: 0.21 s.
VACUUM
```

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));  
pg_size_pretty  
-----  
17 MB
```

Là, par contre, nous gagnons en espace disque. Le `VACUUM FULL` reconstruit la table et la fragmentation disparaît.

Créer une table `t4` avec une colonne `id` de type `integer`.

```
CREATE TABLE t4(id integer);
```

```
CREATE TABLE
```

Désactiver l'autovacuum pour la table `t4`.

```
ALTER TABLE t4 SET (autovacuum_enabled = false);
```

```
ALTER TABLE
```

Insérer un million de lignes dans la table `t4` avec `generate_series`.

```
INSERT INTO t4(id) SELECT generate_series(1, 1000000);  
INSERT 0 1000000
```

Récupérer la taille de la table `t4`.

```
SELECT pg_size_pretty(pg_table_size('t4'));  
pg_size_pretty  
-----  
35 MB
```

Supprimer les 500 000 dernières lignes de la table `t4`.

```
DELETE FROM t4 WHERE id > 500000;
```

```
DELETE 500000
```

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));  
pg_size_pretty  
-----  
35 MB
```

Là aussi, nous n'avons rien perdu.

Exécuter un `VACUUM` sur la table `t4`.

```
VACUUM t4;
```

```
VACUUM
```

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
pg_size_pretty
-----
17 MB
```

En fait, il existe un cas où il est possible de gagner de l'espace disque suite à un `VACUUM` simple : quand l'espace récupéré se trouve en fin de table et qu'il est possible de prendre rapidement un verrou exclusif sur la table pour la tronquer. C'est assez peu fréquent mais c'est une optimisation intéressante.

1.13.2 Détecter la fragmentation

Créer une table `t5` avec deux colonnes : `c1` de type `integer` et `c2` de type `text`.

```
CREATE TABLE t5 (c1 integer, c2 text);
```

```
CREATE TABLE
```

Désactiver l'autovacuum pour la table `t5`.

```
ALTER TABLE t5 SET (autovacuum_enabled=false);
```

```
ALTER TABLE
```

Insérer un million de lignes dans la table `t5` avec `generate_series`.

```
INSERT INTO t5(c1, c2) SELECT i, 'Ligne ' || i FROM generate_series(1, 1000000) AS i;
```

```
INSERT 0 1000000
```

- Installer l'extension `pg_freespacemap` (documentation : <https://docs.postgresql.fr/current/pgfreespacemap.html>)
- Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

```
CREATE EXTENSION pg_freespacemap;
```

```
CREATE EXTENSION
```

Cette extension installe une fonction nommée `pg_freespace`, dont la version la plus simple ne demande que la table en argument, et renvoie l'espace libre dans chaque bloc, en octets, *connu de la Free Space Map*.

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
count | sum
-----+-----
 6274 |    0
```

et donc 6274 blocs (soit 51,4 Mo) sans aucun espace vide.

- Modifier exactement 200 000 lignes de la table `t5`.
- Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

```
UPDATE t5 SET c2 = upper(c2) WHERE c1 <= 200000;
```

```
UPDATE 200000
```

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
count | sum
-----+-----
 7451 |   32
```

La table comporte donc 20 % de blocs en plus, où sont stockées les nouvelles versions des lignes modifiées. Le champ *avail* indique qu'il n'y a quasiment pas de place libre. (Ne pas prendre la valeur de 32 octets au pied de la lettre, la *Free Space Map* ne cherche pas à fournir une valeur précise.)

Exécuter un `VACUUM` sur la table `t5`.

```
VACUUM VERBOSE t5;
```

```
INFO: vacuuming "public.t5"
INFO: "t5": removed 200000 row versions in 1178 pages
INFO: "t5": found 200000 removable, 1000000 nonremovable row versions
      in 7451 out of 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
          There were 0 unused item identifiers.
          Skipped 0 pages due to buffer pins, 0 frozen pages.
          0 pages are entirely empty.
          CPU: user: 0.11 s, system: 0.03 s, elapsed: 0.33 s.
INFO: vacuuming "pg_toast.pg_toast_4160544"
INFO: index "pg_toast_4160544_index" now contains 0 row versions in 1 pages
DÉTAIL : 0 index row versions were removed.
          0 index pages have been deleted, 0 are currently reusable.
          CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO: "pg_toast_4160544": found 0 removable, 0 nonremovable row versions in 0 out of
      ↪ 0 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
          There were 0 unused item identifiers.
          Skipped 0 pages due to buffer pins, 0 frozen pages.
          0 pages are entirely empty.
          CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
count | sum
-----+-----
 7451 | 8806816
```

Il y a toujours autant de blocs, mais environ 8,8 Mo sont à présent repérés comme libres.

Il faut donc bien exécuter un `VACUUM` pour que PostgreSQL nettoie les blocs et mette à jour la structure FSM, ce qui nous permet de déduire le taux de fragmentation de la table.

Récupérer la taille de la table `t5`.

```
SELECT pg_size_pretty(pg_table_size('t5'));
```

```
pg_size_pretty
-----
 58 MB
```

Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

```
VACUUM (FULL, VERBOSE) t5;
```

```
INFO: vacuuming "public.t5"
INFO: "t5": found 200000 removable, 1000000 nonremovable row versions in 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.49 s, system: 0.19 s, elapsed: 1.46 s.
VACUUM
```

Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

```
SELECT count(blkno),sum(avail)FROM pg_freespace('t5'::regclass);
```

```
count | sum
-----+-----
 6274 | 0
```

```
SELECT pg_size_pretty(pg_table_size('t5'));
```

```
pg_size_pretty
-----
 49 MB
```

`VACUUM FULL` a réécrit la table sans les espaces morts, ce qui nous a fait gagner entre 8 et 9 Mo. La taille de la table maintenant correspond bien à celle de l'ancienne table, moins la place prise par les lignes mortes.

1.13.3 Gestion de l'autovacuum

Créer une table `t6` avec une colonne `id` de type `integer`.

```
CREATE TABLE t6 (id integer) ;
```

```
CREATE TABLE
```

Insérer un million de lignes dans la table `t6` :

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

```
INSERT 0 1000000
```

Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Il faudra peut-être attendre une minute. (Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.)

```
\x
```

Expanded display is on.

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-----+-----
reloid          | 4160608
schemaname      | public
relname         | t6
seq_scan        | 0
seq_tup_read    | 0
idx_scan        | x
idx_tup_fetch   | x
n_tup_ins       | 1000000
n_tup_upd       | 0
n_tup_del       | 0
n_tup_hot_upd   | 0
n_live_tup      | 1000000
n_dead_tup      | 0
n_mod_since_analyze | 0
n_ins_since_vacuum | 0
last_vacuum     | x
last_autovacuum | 2021-02-22 17:42:43.612269+01
last_analyze    | x
last_autoanalyze | 2021-02-22 17:42:43.719195+01
vacuum_count    | 0
autovacuum_count | 1
analyze_count   | 0
autoanalyze_count | 1
```

Les deux dates `last_autovacuum` et `last_autoanalyze` sont renseignées. Il faudra peut-être attendre une minute que l'autovacuum passe sur la table (voire plus sur une instance chargée par ailleurs).

Le seuil de déclenchement de l'autoanalyze est :

```
autovacuum_analyze_scale_factor × nombre de lignes
```

+ `autovacuum_analyze_threshold`

soit par défaut $10\% \times 0 + 50 = 50$. Quand il n'y a que des insertions, le seuil pour l'autovacuum est :

`autovacuum_vacuum_insert_scale_factor` × nombre de lignes

+ `autovacuum_vacuum_insert_threshold`

soit $20\% \times 0 + 1000 = 1000$.

Avec un million de nouvelles lignes, les deux seuils sont franchis.



Avec PostgreSQL 12 ou antérieur, seule la ligne `last_autoanalyze` sera remplie. S'il n'y a que des insertions, le démon autovacuum ne lance un `VACUUM` spontanément qu'à partir de PostgreSQL 13.

Jusqu'en PostgreSQL 12, il faut donc lancer manuellement :

```
ANALYZE t6 ;
```

Vérifier le nombre de lignes dans `pg_class.reltuples`.

Vérifions que le nombre de lignes est à jour dans `pg_class` :

```
SELECT * FROM pg_class WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-----+-----
oid          | 4160608
relname      | t6
relnamespace | 2200
reltype     | 4160610
reloftype   | 0
relowner     | 10
relam       | 2
relfilenode | 4160608
reltablespace | 0
relpages    | 4425
reltuples   | 1e+06
...
```

L'autovacuum se base entre autres sur cette valeur pour décider s'il doit passer ou pas. Si elle n'est pas encore à jour, il faut lancer manuellement :

```
ANALYZE t6 ;
```

ce qui est d'ailleurs généralement conseillé après un gros chargement.

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.

- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

```
UPDATE t6 SET id = 0 WHERE id <= 150000 ;
```

```
UPDATE 150000
```

Le démon *autovacuum* ne se déclenche pas instantanément après les écritures, attendons un peu :

```
SELECT pg_sleep(60) ;
```

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-----+-----
relid          | 4160608
schemaname     | public
relname        | t6
seq_scan       | 1
seq_tup_read   | 1000000
idx_scan       | x
idx_tup_fetch  | x
n_tup_ins      | 1000000
n_tup_upd      | 150000
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup     | 1000000
n_dead_tup     | 150000
n_mod_since_analyze | 0
n_ins_since_vacuum | 0
last_vacuum    | x
last_autovacuum | 2021-02-22 17:42:43.612269+01
last_analyze   | x
last_autoanalyze | 2021-02-22 17:43:43.561288+01
vacuum_count   | 0
autovacuum_count | 1
analyze_count  | 0
autoanalyze_count | 2
```

Seul `last_autoanalyze` a été modifié, et il reste entre 150 000 lignes morts (`n_dead_tup`). En effet, le démon *autovacuum* traite séparément l'`ANALYZE` (statistiques sur les valeurs des données) et le `VACUUM` (recherche des espaces morts). Si l'on recalcule les seuils de déclenchement, on trouve pour l'`autoanalyze` :

```
autovacuum_analyze_scale_factor × nombre de lignes
+ autovacuum_analyze_threshold
soit par défaut 10 % × 1 000 000 + 50 = 100 050, dépassé ici.
```

Pour l'`autovacuum`, le seuil est de :

```
autovacuum_vacuum_insert_scale_factor × nombre de lignes
+ autovacuum_vacuum_insert_threshold
soit 20 % × 1 000 000 + 50 = 200 050, qui n'est pas atteint.
```

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

```
UPDATE 60000
```

L'autovacuum ne passe pas tout de suite, les 210 000 lignes mortes au total sont bien visibles :

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6';
```

```
-[ RECORD 1 ]-----+-----
reloid          | 4160608
schemaname     | public
relname        | t6
seq_scan       | 3
seq_tup_read   | 3000000
idx_scan       | x
idx_tup_fetch  | x
n_tup_ins      | 1000000
n_tup_upd      | 210000
n_tup_del      | 0
n_tup_hot_upd  | 65
n_live_tup     | 1000000
n_dead_tup     | 210000
n_mod_since_analyze | 60000
n_ins_since_vacuum | 0
last_vacuum    | x
last_autovacuum | 2021-02-22 17:42:43.612269+01
last_analyze   | x
last_autoanalyze | 2021-02-22 17:43:43.561288+01
vacuum_count   | 0
autovacuum_count | 1
analyze_count  | 0
autoanalyze_count | 2
```

Mais comme le seuil de 200 050 lignes modifiées à été franchi, le démon lance un `VACUUM` :

```
-[ RECORD 1 ]-----+-----
reloid          | 4160608
schemaname     | public
relname        | t6
seq_scan       | 3
seq_tup_read   | 3000000
idx_scan       | x
idx_tup_fetch  | x
n_tup_ins      | 1000000
n_tup_upd      | 210000
n_tup_del      | 0
n_tup_hot_upd  | 65
```

n_live_tup		896905
n_dead_tup		0
n_mod_since_analyze		60000
n_ins_since_vacuum		0
last_vacuum		⌘
last_autovacuum		2021-02-22 17:47:43.740962+01
last_analyze		⌘
last_autoanalyze		2021-02-22 17:43:43.561288+01
vacuum_count		0
autovacuum_count		2
analyze_count		0
autoanalyze_count		2

Noter que `n_dead_tup` est revenu à 0. `last_auto_analyze` indique qu'un nouvel `ANALYZE` n'a pas été exécuté : seules 60 000 lignes ont été modifiées (voir `n_mod_since_analyze`), en-dessous du seuil de 100 050.

Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

```
ALTER TABLE t6 SET (autovacuum_vacuum_scale_factor=0.1);
```

```
ALTER TABLE
```

- Modifier encore 200 000 autres lignes de la table `t6` :

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

- Attendre une minute.

- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?

- Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

```
UPDATE 200000
```

```
SELECT pg_sleep(60);
```

```
SELECT * FROM pg_stat_user_tables WHERE relname='t6' ;
```

```
-[ RECORD 1 ]-----+-----
reloid          | 4160608
schemaname      | public
relname         | t6
seq_scan        | 4
seq_tup_read    | 4000000
idx_scan        | ⌘
idx_tup_fetch   | ⌘
n_tup_ins       | 1000000
n_tup_upd       | 410000
n_tup_del       | 0
n_tup_hot_upd   | 65
n_live_tup      | 1000000
n_dead_tup      | 0
n_mod_since_analyze | 0
```

```
n_ins_since_vacuum | 0
last_vacuum        | x
last_autovacuum    | 2021-02-22 17:53:43.563671+01
last_analyze       | x
last_autoanalyze   | 2021-02-22 17:53:43.681023+01
vacuum_count       | 0
autovacuum_count   | 3
analyze_count      | 0
autoanalyze_count  | 3
```

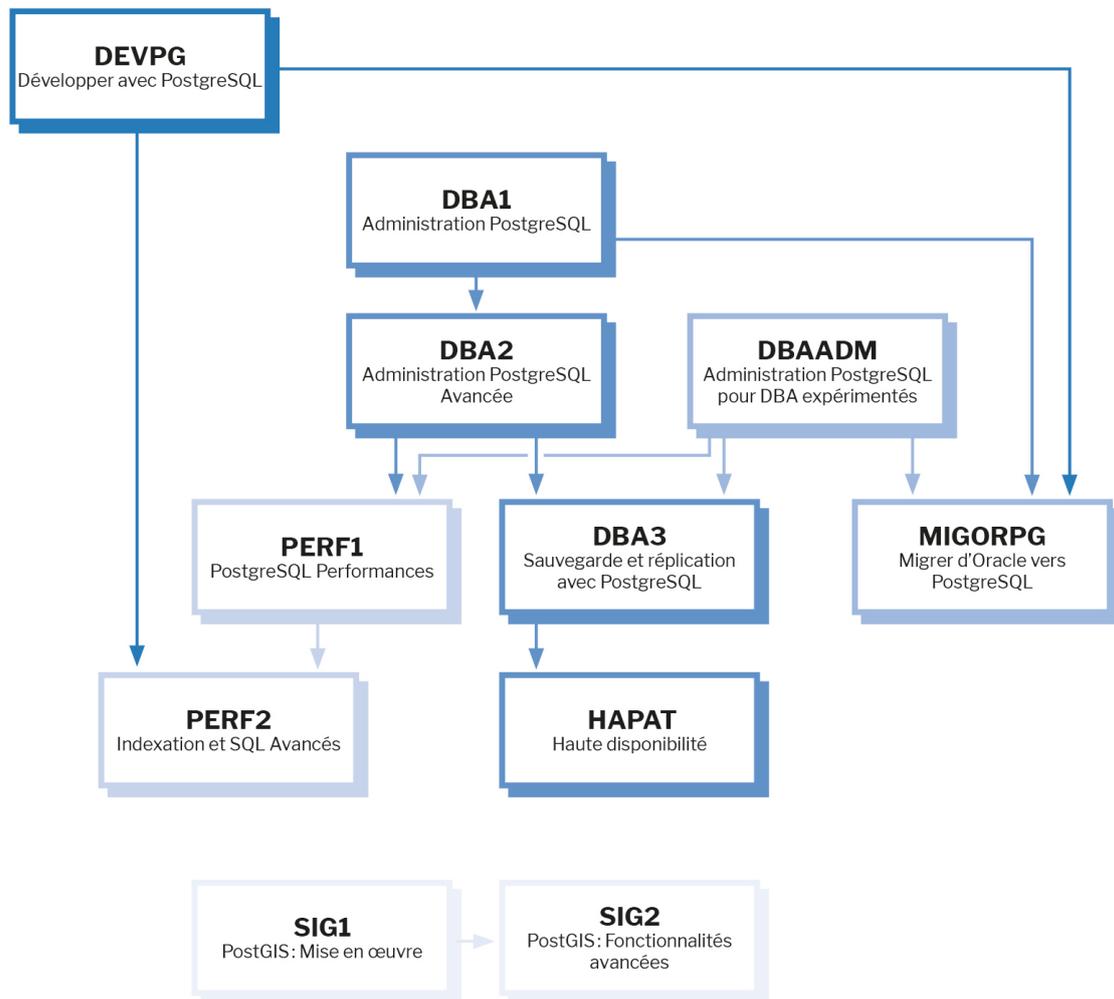
Le démon a relancé un `VACUUM` et un `ANALYZE`. Avec un facteur d'échelle à 10 %, il ne faut plus attendre que la modification de 100 050 lignes pour que le `VACUUM` soit déclenché par le démon. C'était déjà le seuil pour l'`ANALYZE`.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur <https://dali.bo/formation>

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- 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>

Les livres blancs

- Migrer d'Oracle à PostgreSQL
<https://dali.bo/dlb01>
- Industrialiser PostgreSQL
<https://dali.bo/dlb02>
- Bonnes pratiques de modélisation avec PostgreSQL
<https://dali.bo/dlb04>
- Bonnes pratiques de développement avec PostgreSQL
<https://dali.bo/dlb05>

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

