−Table des matières

Opérations de maintenance sous PostgreSQL

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 109 (Octobre 2008). Il est disponible maintenant sous licence Creative Commons.

Après avoir installé un serveur PostgreSQL, il est nécessaire de mettre en place un certain nombre de procédures de maintenance. La plus connue, si on aime ses données, est la sauvegarde. Mais d'autres opérations sont importantes pour conserver de bonnes performances.

Opérations de base

Il existe deux opérations de base essentielles et couvrant des domaines très différents : ANALYZE pour des statistiques sur le contenu des tables, et VACUUM pour éviter un grossissement anormal des tables et index.

ANALYZE

Cette opération a pour but de récupérer un certain nombre de statistiques pour un échantillon de lignes de chaque table. Le résultat de ce travail est stocké, colonne par colonne dans une table nommée pg_statistics. Étant difficilement lisible, la vue pg_stats a été créé pour offrir une vue simplifiée des données récupérées. Deux autres informations sont aussi récupérées : une estimation du nombre de lignes (reltuples) et une estimation du nombre de pages disque (relpages). Ces deux informations sont stockées dans le catalogue système pg_class.

Mais à quoi servent ces statistiques ? Disons qu'un utilisateur exécute une requête du type SELECT * FROM t1 WHERE c1=2. Suivant que la valeur 2 est une valeur fréquente ou non de la table, l'optimiseur pourra en déduire le type de parcours optimal. Dans le cas d'une valeur fréquente, il aura tendance à privilégier un parcours séquentiel alors que dans le cas d'une valeur rare, il préférera un parcours d'index.

Testons tout cela. Créons une table et ajoutons-y une ligne :

```
glmf=# CREATE TABLE t1 (c1 integer);
glmf=# INSERT INTO t1 (c1) VALUES (1), (2);
glmf=# \x
glmf=# SELECT relname, reltuples, relpages FROM pg_class WHERE relname='t1';
-[ RECORD 1 ]-
relname   | t1
reltuples | 0
relpages  | 0
glmf=# SELECT * FROM pg_stats WHERE tablename='t1';
(Aucune ligne)
```

Sans exécution d'ANALYZE, les catalogues pg_class et pg_stats n'ont pas de données à jour. Exécutons donc un ANALYZE pour voir la différence :

```
glmf=# ANALYZE t1;
glmf=# SELECT relname, reltuples, relpages FROM pg_class WHERE relname='t1';
-[ RECORD 1 ]-
relname   | t1
reltuples | 2
relpages  | 1
reltuples indique bien deux lignes, relpages une seule page.
glmf=# SELECT * FROM pg_stats WHERE tablename='t1';
-[ RECORD 1 ]-----+-------
schemaname        | public
tablename         | t1
attname           | c1
null_frac         | 0
avg_width         | 4
n_distinct        | -1
most_common_vals  |
most_common_freqs |
histogram_bounds  | {1,2}
correlation       | 1
```

Les colonnes schemaname, tablename et attname précisent la colonne visée (il s'agit respectivement du nom du schéma, du nom de la table et du nom de la colonne). avg_width est la largeur moyenne de la colonne. Pour un type entier, cela ne bougera jamais car une valeur de ce type est toujours stockée sur quatre octets, d'où son alias int4. C'est une colonne beaucoup plus intéressante dans le cas d'un champ texte dont la longueur est variable. La colonne

histogram_bounds précise quelques valeurs de la table. Ajoutons deux lignes :

```
glmf=# INSERT INTO t1 (c1) VALUES (3), (2);
glmf=# ANALYZE t1;
glmf=# SELECT * FROM pg_stats WHERE tablename='t1';
-[ RECORD 1 ]-----+-------
schemaname        | public
tablename         | t1
attname           | c1
null_frac         | 0
avg_width         | 4
n_distinct        | -0.75
most_common_vals  | {2}
most_common_freqs | {0.5}
histogram_bounds  | {1,3}
correlation       | 0.8
```

La table t1 contient maintenant quatre valeurs : 1, 2, 2, 3. Il est donc logique que la colonne most_common_vals ne contient que la valeur 2. Insérons beaucoup plus de données :

```
glmf=# INSERT INTO t1 (c1) SELECT round(random()*10) FROM generate_series(1, 1000000);
glmf=# ANALYZE t1;
glmf=# SELECT relname, reltuples, relpages FROM pg_class WHERE relname='t1';
-[ RECORD 1 ]-----
relname   | t1
reltuples | 999971
relpages  | 3922
```

Première différence avec la réalité : au lieu d'avoir un million et quatre lignes, ANALYZE a estimé le nombre de lignes à 999971. La différence n'est pas très importante, mais il est essentiel de bien comprendre que le nombre indiqué par reltuples n'est qu'une estimation.

```
glmf=# SELECT * FROM pg_stats WHERE tablename='t1';
-[ RECORD 1 ]-----+-----------------------------------
schemaname        | public
tablename         | t1
attname           | c1
null_frac         | 0
avg_width         | 4
n_distinct        | 11
most_common_vals  | {3,4,1,5,9}
most_common_freqs | {0.11,0.102667,0.102,0.101667,0.1}
histogram_bounds  | {0,2,6,7,8,10}
correlation       | 0.072549
```

Cette fois, les valeurs les plus fréquentes sont les 3, 4, 1, 5 et 9. Vérifions cela :

```
glmf=# \x
glmf=# SELECT c1, count(*) FROM t1 GROUP BY c1 ORDER BY 2 DESC;
 c1 | count
----+--------
  3 | 100337
  7 | 100314
  4 | 100274
  9 | 100234
  1 | 100164
  2 | 100024
  8 | 100017
  5 |  99739
  6 |  99647
 10 |  49717
  0 |  49537
(11 lignes)
```

Les cinq valeurs trouvées par ANALYZE se trouvent dans les huit premières valeurs, preuve que les statistiques ne sont pas forcément très performantes. La colonne most_common_freqs donne la fréquence des valeurs les plus communes. Par exemple, la valeur 3 est trouvée à une fréquence de 11 %. La table faisant en gros un million de lignes, cela sous-entend que la valeur 3 sera disponible sur 110000 lignes. Le résultat de la requête ci-dessus montre que l'estimation est plutôt bonne, la valeur 3 étant réellement disponible sur 100337 lignes. Rappelons qu'il s'agit d'un calcul statistique sur un échantillon, le calcul ne peut pas être exact. La différence peut sembler importante, mais dans le cas de l'utilisation par le planificateur, si la valeur 3 se trouve dans 11 % de la table, la question ne se pose plus, seul un parcours d'index sera le plus efficace. Vérifions cela, ajoutons un index :

```
glmf=# CREATE INDEX i1 ON t1 (c1);
glmf=# EXPLAIN SELECT * FROM t1 WHERE c1=3;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=1820.84..7117.84 rows=110000 width=4)
   Recheck Cond: (c1 = 3)
   ->  Bitmap Index Scan on i1  (cost=0.00..1793.34 rows=110000 width=0)
         Index Cond: (c1 = 3)
(4 lignes)
```

La ligne intéressante est la ligne en gras. Nous avons bien un parcours d'index (exactement un « Bitmap Index Scan ») car le planificateur a estimé qu'il allait trouver 110000 lignes en filtrant avec la condition « c1 = 3 ». La valeur de rows est l'estimation du nombre de lignes récupérées en respectant la condition. Cette estimation provient du planificateur. Ce dernier a fait exactement le même calcul que nous au paragraphe précédent (11% de 999971 lignes, soit 109993, qui s'est trouvé arrondi à 110000).

Autrement dit, pour que le planificateur puisse faire de bons choix, il doit disposer de statistiques à jour et suffisamment précises.

La fréquence d'exécution de la commande ANALYZE dépend essentiellement de la fréquence de changement des données. Donc, pour avoir des statistisques à jour, il suffit de planifier une exécution automatique, soit après un certain délai (par exemple avec cron), soit après un certain nombre de modifications (en utilisant l'autovacuum).

Quant à la précision, nous pouvons l'améliorer grâce à un paramètre global appelé default_statistics_target. Modifions-le et utilisons l'option VERBOSE de la commande ANALYZE pour qu'elle nous détaille les calculs statistiques :

```
glmf=# SET default_statistics_target TO 10;
glmf=# ANALYZE VERBOSE t1;
INFO:  analyse « public.t1 »
INFO:  « t1 » : 3000 pages parcourues sur 3922,
  contenant 764894 lignes à conserver et 0 lignes à supprimer,
  3000 lignes dans l'échantillon,
  999971 lignes totales estimées
```

L'opération ANALYZE a parcouru 3000 pages de la table sur 3922. Elle a récupéré 3000 lignes dans son échantillon pour calculer ses statistiques. Il est évident qu'avec 3000 lignes (sur plus d'un million), les statistiques ne peuvent pas être très précises.

```
glmf=# SET default_statistics_target TO 100;
glmf=# ANALYZE VERBOSE t1;
INFO:  analyse « public.t1 »
INFO:  « t1 » : 3922 pages parcourues sur 3922,
  contenant 1000004 lignes à conserver et 0 lignes à supprimer,
  30000 lignes dans l'échantillon,
  1000004 lignes totales estimées
```

Cette fois, l'échantillon utilise 30000 lignes, ce qui ne peut qu'améliorer la précision des statistiques. Le facteur de multiplication de default_statistics_target est appliqué au nombre de lignes utilisées pour l'échantillon. Autre constatation, le nombre de lignes estimé est bien plus précis. En fait, il est même exact, ce qui est logique car l'opération a parcouru toutes les pages de la table. Voyons maintenant l'état des statistiques :

```
glmf=# SELECT * FROM pg_stats WHERE tablename='t1';
-[ RECORD 1 ]-----+----------------------------------------------------------------------------------------------------
schemaname        | public
tablename         | t1
attname           | c1
null_frac         | 0
avg_width         | 4
n_distinct        | 11
most_common_vals  | {1,2,9,8,7,5,6,3,4,0,10}
most_common_freqs | {0.1018,0.101167,0.100167,0.100067,0.0991,0.0990333,0.0989333,0.0986,0.0983667,0.0514333,0.0513333}
histogram_bounds  |
correlation       | 0.0941376
```

Le nombre de valeurs fréquentes a augmenté, ce qui améliore de nouveau les estimations.

Évidemment, l'amélioration des estimations a un prix : l'ANALYZE met plus de temps à s'exécuter (car plus de lignes lues, donc plus d'accès disque) et l'espace disque utilisé pour stocker les statistiques augmente. Cependant, la valeur par défaut de default_statistics_target est souvent considérée comme trop basse. Beaucoup la placent directement à 100.

Les inconvénients mentionnés ci-dessus ont amené les développeurs à proposer un default_statistics_target configurable pour chaque colonne de chaque table. Ce paramètre est configurable avec la commande « ALTER TABLE ». Évidemment, cela ne joue pas sur la taille de l'échantillon sélectionnée, mais uniquement sur le nombre de valeurs communes. Voici comment modifier ce paramètre pour la colonne c1 de la table t1 :

```
glmf=# ALTER TABLE t1 ALTER COLUMN c1 SET STATISTICS 1000;
```

Un ANALYZE est nécessaire ensuite pour mettre à jour les statistiques.

VACUUM simple

Avant de chercher à savoir ce que fait cette opération de maintenance, commençons par regarder le travail réalisé par PostgreSQL sur une table à la suite d'opérations de type DDL (modification de données). Créons une table simple et insérons une ligne :

```
glmf=# CREATE TABLE t2 (c1 integer);
glmf=# INSERT INTO t2 (c1) VALUES (1);
glmf=# SELECT * FROM t2;
 c1
----
  1
(1 ligne)
```

Cette ligne a bien été enregistrée dans la table t2 qui dispose d'une seule colonne, c1. Plus exactement d'une seule colonne utilisateur car elle comprend plusieurs colonnes systèmes que nous pouvons aussi interroger :

```
glmf=# SELECT tableoid, xmin, xmax, cmin, cmax, ctid, * FROM t2;
 tableoid | xmin | xmax | cmin | cmax | ctid  | c1
----------+------+------+------+------+-------+----
    72609 | 4310 |    0 |    0 |    0 | (0,1) |  1
(1 ligne)
```

Voici la signification de chaque colonne système :

- tableoid, identifiant de la table ;
- xmin, identifiant de la transaction qui a procédé à l'insertion de cette version de ligne ;
- xmax, identifiant de transaction qui a exécuté la destruction de cette version de ligne ;
- cmin, identifiant de commande au sein de la transaction d'insertion ;
- cmax, identifiant de commande au sein de la transaction de destruction ;
- ctid, emplacement physique de l'enregistrement.

Note : Vous pouvez aussi avoir une colonne oid. À partir de la version 8.1, cette colonne n'est pas disponible par défaut car elle est inutile . De plus, nous gagnons quatre octets par lignes. Néanmoins, vous pouvez de nouveau les avoir en créant la table avec la clause WITH OID ou en activant le paramètre default_with_oids dans le fichier de configuration.

Nous pouvons donc en conclure que la transaction 4310 (valeur de la colonne xmin) a créé cette ligne, qu'elle est valable pour toutes les transactions après la 4310 (mais seulement une fois que la transaction 4310 aura été validée, ce qui est le cas pour cette transaction implicite). Nous apprenons aussi qu'elle est en position 1 dans le fichier, ce qui est logique pour la première insertion d'une ligne dans une table nouvellement créée.

Ajoutons une deuxième ligne :

```
glmf=# INSERT INTO t2 (c1) VALUES (2);
glmf=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4310 |    0 | (0,1) |  1
 4311 |    0 | (0,2) |  2
(2 lignes)
```

Cette ligne ayant été créée après une nouvelle transaction, elle se voit affectée d'un identifiant de transaction supérieur. Dans ce cas très particulier où un seul client est connecté à la base, le xmin est seulement incrémenté de 1, mais la différence pourrait être plus importante. Cette ligne se trouve en position 2 (colonne ctid).

Insérons une troisième ligne, puis mettons à jour la ligne 2 :

```
glmf=# INSERT INTO t2 (c1) VALUES (3);
glmf=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4310 |    0 | (0,1) |  1
 4311 |    0 | (0,2) |  2
 4312 |    0 | (0,3) |  3
(3 lignes)
glmf=# UPDATE t2 SET c1=-2 WHERE c1=2;
glmf=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4310 |    0 | (0,1) |  1
 4312 |    0 | (0,3) |  3
 4313 |    0 | (0,4) | -2
(3 lignes)
```

La nouvelle insertion se passe bien en ligne 3. Par contre, la mise à jour est plus étonnante. Elle se trouve en position 4. En effet, dans le cas d'une mise à jour, PostgreSQL va gérer deux versions de la même ligne. La première version est la version initiale, située dans notre cas à l'emplacement 2. Elle a comme transaction de création la 4311 et comme transaction de destruction la 4313. Elle est invisible pour le SELECT car nous nous trouvons après cette transaction. La deuxième version est la nouvelle version, créée suite à la mise à jour. Elle est située à l'emplacement 4, a comme transaction de création la 4313 et n'a pas encore de transaction de destruction. En fait, si on pouvait réellement voir ce que contient le fichier, on verrait ceci :

```
 xmin | xmax | ctid  | c1
------+------+-------+----
 4310 |    0 | (0,1) |  1
 4311 | 4313 | (0,2) |  2
 4312 |    0 | (0,3) |  3
 4313 |    0 | (0,4) | -2
```

Un moyen de le voir est de commencer une transaction de modification avec un premier client psql, et de vérifier le contenu de la table avec un deuxième client. Le premier aura pour invite glmf1, et le deuxième glmf2 :

```
guillaume@laptop$ psql -q -v PROMPT1="glmf1=# " glmf
glmf1=# BEGIN;
glmf1=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4318 |    0 | (0,3) |  3
 4319 |    0 | (0,4) | -2
(3 lignes)
glmf1=# UPDATE t2 SET c1=2 WHERE c1=-2;
glmf1=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4318 |    0 | (0,3) |  3
 4320 |    0 | (0,5) |  2
(3 lignes)
```

Sur le premier client, on voit ce qu'on avait auparavant, à savoir que la ligne 4 a disparu pour laisser la place à la ligne 5. Maintenant, voyons ce que dit l'autre client :

```
guillaume@laptop$ psql -q -v PROMPT1="glmf2=# " glmf
glmf2=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4318 |    0 | (0,3) |  3
 4319 | 4320 | (0,4) | -2
(3 lignes)
```

Pour lui, la ligne 4 existe toujours car la transaction du premier client n'est pas validée. Cette valeur sera toujours visible tant que la validation n'aura pas eu lieu. Vérifions cela :

```
glmf1=# COMMIT;
glmf1=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
```

```
 4318 |    0 | (0,1) |  1
 4318 |    0 | (0,3) |  3
 4320 |    0 | (0,5) |  2
(3 lignes)
glmf2=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4318 |    0 | (0,3) |  3
 4320 |    0 | (0,5) |  2
(3 lignes)
```

Et voilà ! La nouvelle version de ligne est disponible pour le second client dès que la transaction du premier est validée.

Ce dont nous nous apercevons là est qu'une table vit. Des données sont insérées, des données sont mises à jour ou supprimées et cela résulte en la modification et l'ajout de lignes dans la table. Certaines lignes ne deviennent plus visibles malgré qu'elles soient toujours présentes dans le fichier et valides pour certaines processus. Ce qui va directement nous amener à une fragmentation de la table lorsque certaines lignes deviendront invisibles pour tous les clients. Si nous récapitulons les lignes de cette table, nous nous apercevons que les lignes 1, 3 et 5 sont visibles (vivante est un terme très utilisé dans ce contexte) alors que les lignes 2 et 4 ne le sont plus (elles sont dites mortes quand plus aucun processus ne peut les lire). PostgreSQL ne peut pas réutiliser ces lignes car il ne sait pas qu'elles sont réellement mortes. Elles ont un identifiant de transaction de fin, mais rien ne dit qu'un processus postgres ne les voit pas encore. Parcourir la table pour trouver un espace prendrait trop de temps pour être envisageable.

L'opération de maintenance VACUUM est là pour forcer PostgreSQL à vérifier la visibilité des lignes pour les transactions en cours. Le VACUUM parcourt toute la table et vérifie pour chaque ligne ayant un xmax renseigné si un processus postgres est encore capable de lire cette ligne. Dans le cas contraire, il déclare la ligne morte et enregistre en mémoire le fait que cette ligne est disponible pour être ré-utilisée. Exécutons un VACUUM en mode verbeux pour voir le travail réalisé par le VACUUM :

```
glmf=# VACUUM VERBOSE t2;
INFO:  exécution du VACUUM sur « public.t2 »
INFO:  « t1 » : 2 versions de ligne supprimables, 3 non supprimables
parmi 1 pages
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 1 pointeurs d'éléments inutilisés.
1 pages contiennent de l'espace libre utile.
0 pages sont entièrement vides.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

C'est principalement la ligne INFO: « t2 » : 2 versions de ligne supprimables, 3 non supprimables parmi 1 pages qui nous intéresse actuellement. Elle indique que le VACUUM a trouvé deux versions de lignes supprimables (les lignes 2 et 4) et 3 non supprimables. Donc, si on insère ou si on met à jour une ligne, elle devrait pouvoir réutiliser les lignes 2 et 4. Testons cela :

```
glmf=# INSERT INTO t2 (c1) VALUES (4);
glmf=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4332 |    0 | (0,2) |  4
 4318 |    0 | (0,3) |  3
 4320 |    0 | (0,5) |  2
(4 lignes)
glmf=# UPDATE t2 SET c1=-2 WHERE c1=2;
glmf=# SELECT xmin, xmax, ctid, * FROM t2;
 xmin | xmax | ctid  | c1
------+------+-------+----
 4318 |    0 | (0,1) |  1
 4332 |    0 | (0,2) |  4
 4318 |    0 | (0,3) |  3
 4333 |    0 | (0,4) | -2
(4 lignes)
```

Donc, le VACUUM nous permet après coup de réutiliser les lignes mortes d'une table. PostgreSQL se souvient de ces lignes grâce à une structure placée en mémoire. La FSM (pour Free Space Map, autrement dit carte des espaces libres) contient des pointeurs vers chaque espace libre de chaque table… à condition que la taille de cette structure soit suffisante pour tout contenir. Le fichier de configuration de PostgreSQL nommé postgresql.conf permet de configurer le nombre de relations suivies par cette structure ainsi que le nombre de pages qu'il est possible de suivre. Le premier correspond au paramètre max_fsm_relations. Il vaut par défaut 1000, ce qui convient dans la majorité des cas. Le second se nomme max_fsm_pages et a une valeur habituellement basse. Il convient assez fréquemment de l'augmenter. Il faut savoir que chaque page de 8 Ko surveillée prend, en mémoire, 6 octets. Il serait donc dommage de ne pas augmenter ce paramètre à une valeur plus importante. En effet, si la structure est trop petite, les informations sur les espaces libres trouvés par le VACUUM ne seront pas enregistrées et les tables continueront à être fragmentées.

Note : Les variables de configuration max_fsm_pages et max_fsm_relations sont globales à toutes les bases de données. Par exemple, max_fsm_relations doit être supérieur au nombre des tables et index de toutes les bases de données du système.

Note : Il est possible de connaître le contenu de la structure FSM grâce au module contrib pg_freespacemap.

Le VACUUM peut aussi faire gagner un peu de place sur disque. Dans le cas où le dernier bloc de 8 Ko ne contient que des lignes mortes et si VACUUM peut poser un verrou exclusif à ce moment (en temps normal, cette opération ne pose aucun verrou exclusif), VACUUM est capable de supprimer ce bloc :

```
glmf=# SELECT pg_size_pretty(pg_relation_size('t2'));
 pg_size_pretty
----------------
 8192 bytes
(1 ligne)
glmf=# INSERT INTO t2 SELECT x FROM generate_series(1, 300) AS x;
glmf=# SELECT pg_size_pretty(pg_relation_size('t2'));
 pg_size_pretty
----------------
 16 kB
(1 ligne)
```

Après l'insertion de 300 valeurs, la table fait 16 Ko, soit deux blocs de 8 Ko.

```
glmf=# DELETE FROM t2 WHERE c1>100;
glmf=# SELECT pg_size_pretty(pg_relation_size('t2'));
 pg_size_pretty
----------------
 16 kB
(1 ligne)
```

Après avoir supprimé 200 lignes, la table fait toujours la même taille.

```
glmf=# VACUUM VERBOSE t2;
INFO:  exécution du VACUUM sur « public.t2 »
INFO:  « t2 » : 100 versions de ligne supprimées parmi 2 pages
INFO:  « t2 » : 100 versions de ligne supprimables, 204 non supprimables
parmi 2 pages
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 1 pointeurs d'éléments inutilisés.
2 pages contiennent de l'espace libre utile.
0 pages sont entièrement vides.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  « t1 » : 2 pages tronqués en 1
DÉTAIL : CPU 0.00s/0.00u sec elapsed 0.00 sec.
glmf=# SELECT pg_size_pretty(pg_relation_size('t2'));
 pg_size_pretty
----------------
 8192 bytes
(1 ligne)
```

Comme les 200 lignes supprimées étaient les 200 dernières lignes de la table, la commande VACUUM a pu libérer un bloc, ce qu'il indique d'ailleurs dans les traces avec le message :

```
INFO:  « t2 » : 2 pages tronqués en 1
```

Mais c'est bien le seul cas où un VACUUM simple peut relâcher de l'espace disque sur le système de fichiers.

Pour résumé, l'utilisation normale d'une table va aboutir à une fragmentation plus ou moins importante de cette table. Pour éviter une fragmentation trop importante, il est nécessaire de réaliser fréquemment un VACUUM. Il est aussi important d'avoir configuré correctement la taille de la structure FSM.

Note : Le module contrib pgstattuple permet de connaître le taux de fragmentation d'une table ou d'un index. Veillez toutefois à ne pas exécuter trop fréquemment la fonction de ce module car elle utilise un verrou bloquant les autres processus, sans compter qu'elle lit intégralement la table, ce qui peut poser des soucis d'accès aux disques.

Malheureusement, cette opération a un coût, notamment au niveau des entrées/sorties disque. Chaque table concernée par le VACUUM devra être entièrement lue. Sur notre exemple, cela a peu d'importance, la table faisant 8 Ko, mais dans le cas de tables de plusieurs Go, cela peut avoir des répercussions fâcheuses sur l'accès au disque par les autres processus. Les développeurs de PostgreSQL en étaient bien conscients et des paramètres sont apparus pour lâcher la pression au niveau disque.

Le plus simple, et le plus efficace, est maintenance_work_mem. Il précise la quantité de mémoire utilisable pour certaines opérations de maintenance comme, entre autres, le VACUUM et la création d'index. Il ne faut pas hésiter à allouer beaucoup de mémoire à ce type d'opération car, n'étant pas commune, elle ne risque pas d'être allouée beaucoup de fois en même temps. Des valeurs de 512 Mo à plusieurs Go, suivant la mémoire disponible, sont tout à fait acceptables.

Comme le VACUUM est une opération non bloquante, un des moyens d'améliorer l'interactivité des autres processus est de faire s'exécuter le VACUUM sur une période plus longue en l'endormissant de temps en temps. En fait, l'opération de VACUUM va compter un coût et, après avoir atteint une limite de coût, mettra son exécution en pause pendant un certain temps. Le coût se calcule par rapport à l'accès au disque. Une page récupérée en mémoire aura un coût faible (paramètre vacuum_cost_page_hit) alors qu'une page récupérée sur disque aura un coût plus important (paramètre vacuum_cost_page_miss). Ce sera encore pire pour une page récupérée dans le cache mais à écrire sur disque car modifiée (paramètre vacuum_cost_page_dirty). Les coûts de chaque page additionnés sont comparés à une limite fixée par le paramètre vacuum_cost_limit. Si cette limite est atteinte, le processus réalisant le VACUUM est endormi pendant une période configurée avec le paramètre vacuum_cost_delay (délai en millisecondes), laissant ainsi le temps aux autres processus de profiter de l'accès aux disques. Ces paramètres sont généralement peu modifiés. Il s'avère important de le faire quand un VACUUM ralentit l'exécution des autres processus.

Note : Un grand nombre de rapports sur des VACUUM prennant trop de temps se révèle avoir pour cause une mauvaise configuration de vacuum_cost_delay. La valeur est souvent trop importante. En tout cas, le problème revient suffisamment souvent pour que les développeurs se penchent dessus et prévoient des modifications pour corriger cela en 8.4.

Pour résumer le VACUUM simple :

- nécessaire pour que PostgreSQL puisse réutiliser l'espace libre à l'intérieur des tables ;
- pas de verrou exclusif, donc non bloquant pour les autres processus ;
- ne fait pas gagner d'espace sur le disque sauf en de très rares occasions ;
- taille de la structure FSM à configurer, sinon l'opération de VACUUM est moins efficace.

Opérations avancées

VACUUM FULL

Dans certains cas, une table peut se trouver tellement fragmentée qu'un simple VACUUM ne suffira pas. Cela se produit suite à une erreur de l'administrateur (par exemple si ce dernier a oublié de planifier une exécution périodique du VACUUM, ou s'il a configuré la structure FSM trop justement). Cela arrive aussi quand il est nécessaire de vider un grand pourcentage d'une table.

Dans tous ces cas, la solution la plus directe est d'exécuter un VACUUM avec l'option FULL. Cela effectue une défragmentation de la table. Comme la table sera fortement impactée par les écritures, un verrou exclusif est posé sur la table en cours de traitement. C'est l'un des gros inconvénients de cette

méthode : personne ne peut lire ou modifier cette table pendant cette opération. Et comme on exécute cette opération généralement sur des grosses tables, cela revient à empêcher l'utilisation de cette table pendant un long moment.

Créons une table, et insérons un grand nombre de données :

```
glmf=# CREATE TABLE t3 (c1 integer);
glmf=# INSERT INTO t3 SELECT x FROM generate_series(1, 1000000) AS x;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 31 MB
(1 ligne)
```

Une fois le million de lignes inséré, la table fait 31 Mo. Supprimons-en la majeure partie et vérifions la taille de la table :

```
glmf=# DELETE FROM t3 WHERE c1 < 950000;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 31 MB
(1 ligne)
```

Aucune différence, ce qui est logique, PostgreSQL a juste placé l'identifiant de transaction de destruction sur chaque ligne supprimée. Exécutons maintenant un VACUUM simple :

```
glmf=# VACUUM t3;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 31 MB
(1 ligne)
```

Aucune différence non plus. Le fait que nous n'avons pas ajouté le mot clé VERBOSE ne change rien à cela. Un VACUUM simple ne modifie pas la taille d'une table si les derniers éléments de cette table sont toujours valides (nous n'avons supprimée que les 950000 premières lignes, pas les dernières).

Maintenant, exécutons un VACUUM FULL, mais avant cela récupérons l'OID et le relfilenode de la table t3 :

```
glmf=# SELECT oid, relname, relfilenode FROM pg_class WHERE relname='t3';
  oid  | relname | relfilenode
-------+---------+-------------
 72618 | t3      |       72618
(1 ligne)
glmf=# VACUUM FULL t3;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 1576 kB
(1 ligne)
glmf=# SELECT oid, relname, relfilenode FROM pg_class WHERE relname='t3';
  oid  | relname | relfilenode
-------+---------+-------------
 72618 | t3      |       72618
(1 ligne)
```

Nous pouvons donc faire deux constatations :

- la table a grandement diminué en taille car nous sommes passés de 31 Mo à 1,5 Mo ;
- le fichier n'a pas été modifié (OID et relfilenode identiques avant et après le VACUUM FULL)… il s'agit donc bien d'une défragmentation et pas d'une création d'une table à l'identique.

Malgré le fait que le VACUUM FULL réalise une défragmentation directement sur le même fichier, l'opération est annulable :

```
glmf=# DROP TABLE t3;
glmf=# CREATE TABLE t3 (c1 integer);
glmf=# INSERT INTO t3 SELECT x FROM generate_series(1, 1000000) AS x;
glmf=# DELETE FROM t3 WHERE c1 < 950000;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 31 MB
(1 ligne)

glmf=# VACUUM FULL t3;
Cancel request sent
ERREUR:  annulation de la requête à la demande de l'utilisateur
glmf=# SELECT pg_size_pretty(pg_relation_size('t3'));
 pg_size_pretty
----------------
 31 MB
(1 ligne)

glmf=# SELECT count(*) from t3;
 count
-------
 50001
(1 ligne)
```

Après annulation du VACUUM FULL, la table fait toujours 31 Mo et elle contient bien le bon nombre de lignes. C'est tout à fait logique. C'est une opération comme les autres, elle est tracée dans les journaux de transactions, elle bénéficie des mêmes possibilités.

En dehors du verrou exclusif, inconvénient majeur dans le cadre d'opérations concurrentes, le VACUUM FULL va aussi baisser l'efficacité des index. Chaque ligne qui se verra déplacé pour permettre la défragmentation de la table va générer un pointeur supplémentaire dans les index associés.

```
glmf=# DROP TABLE t3;
glmf=# CREATE TABLE t3 (c1 integer);
glmf=# INSERT INTO t3 SELECT x FROM generate_series(1, 1000000) AS x;
glmf=# DELETE FROM t3 WHERE c1 < 950000;
glmf=# CREATE INDEX i1 ON t3 (c1);
glmf=# SELECT pg_size_pretty(pg_relation_size('t3')), pg_size_pretty(pg_relation_size('i1'));
 pg_size_pretty | pg_size_pretty
----------------+----------------
 31 MB          | 896 kB
(1 ligne)
glmf=# VACUUM FULL t3;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3')), pg_size_pretty(pg_relation_size('i1'));
 pg_size_pretty | pg_size_pretty
----------------+----------------
 1576 kB        | 1768 kB
(1 ligne)
```

Avant le VACUUM FULL, l'index faisait 896 Ko. Après, il en arrive à 1,7 Mo. Comparé au 1,5 Mo de la table, nous sommes certains qu'il sera utilisé moins souvent. Une nouvelle création de l'index nous montre qu'il fait de nouveau 896 Ko. Le grossissement de l'index est donc bien une conséquence de l'utilisation du VACUUM FULL.

REINDEX

Nous venons de voir que l'opération de défragmentation d'une table était le VACUUM FULL. Nous avons vu aussi que cette opération avait pour conséquence néfaste une explosion de la taille des index. Il nous faut donc un outil pour re-construire les index, dont l'utilisation sera nécessaire après chaque VACUUM FULL.

REINDEX est cet outil. Il fonctionne de la même manière que CREATE INDEX. Il en a les avantages (des performances améliorées avec un maintenance_work_mem important), et les inconvénients (verrou exclusif sur la table et l'index).

Voici un exemple de son utilisation :

```
glmf=# SELECT pg_size_pretty(pg_relation_size('t3')), pg_size_pretty(pg_relation_size('i1'));
 pg_size_pretty | pg_size_pretty
----------------+----------------
 1576 kB        | 1768 kB
(1 ligne)
glmf=# REINDEX TABLE t3;
glmf=# SELECT pg_size_pretty(pg_relation_size('t3')), pg_size_pretty(pg_relation_size('i1'));
 pg_size_pretty | pg_size_pretty
----------------+----------------
 1576 kB        | 896 kB
(1 ligne)
```

Il est essentiel d'exécuter un REINDEX après un VACUUM FULL pour s'assurer d'avoir des index efficaces.

Automatisation

Maintenant que nous savons pourquoi nous avons besoin fréquemment de certaines opérations de maintenance, il nous reste à automatiser leur exécution périodique.

Par cron

Le moyen le plus simple est d'utiliser cron. Cet outil permet d'exécuter des commandes suivant un planning (par exemple tous les dimanche à 23h).

Sans informations sur l'activité réelle de la base de données, il convient de faire au moins un VACUUM ANALYZE (simple) par jour et un REINDEX par semaine. C'est vraiment un minimum. Il est préférable de faire ses actions-là sur chaque base complète. Le mieux est de passer par les outils vacuumdb et reindexdb, ce qui peut donner par exemple :

```
guillaume@laptop$ crontab -l
# m h  dom mon dow   command
00 23 * * * vacuumdb -z glmf
00 01 * * * reindexdb glmf
```

Il est toujours possible après cela d'avoir des lignes supplémentaires pour exécuter plus fréquemment un VACUUM sur certaines tables très impactées par des écritures. Le meilleur moyen de connaître les tables impactées est de passer par les statistiques lignes disponibles dans la table pg_stat_user_tables. Mais plutôt que d'essayer de configurer finement les opérations dans cron, il est souvent préférable d'utiliser l'autovacuum.

Autovacuum

L'autovacuum est un processus lancé à intervalle régulier par le serveur PostgreSQL pour traiter une base de données différente à chaque lancement. Ce processus va chercher toutes les tables suffisamment modifiées pour permettre l'exécution d'un VACUUM ou d'un ANALYZE sur uniquement ces tables. Des paramètres permettent de configurer le seuil de déclenchement.

Par défaut, l'autovacuum est activé (paramètre autovacuum à on). Il faut aussi que track_counts soit à on car l'autovacuum se base sur les statistiques lignes qu'engrange le collecteur de statistiques.

Le paramètre autovacuum_naptime a changé de signification avec la version 8.3.

Avant la version 8.3, autovacuum s'exécutait tous les autovacuum_naptime secondes. Ce n'était donc pas un démon qui se réveillait de temps à autre, mais un nouveau processus. À chaque exécution, le processus choisissait une base de données et cherchait les tables et index ayant besoin d'un VACUUM ou d'un ANALYZE dans cette base. Autrement dit, sur un système comprenant cinq bases de données et un autovacuum_naptime à une minute, une base de données se verra analysée toutes les cinq minutes.

À partir de la version 8.3, il existe deux types de processus autovacuum : le « autovacuum_launcher » et les « autovacuum workers ». Le premier est

exécuté en permanence. Il fonctionne réellement comme un démon. Il lance un « autovacuum worker » pour chaque base à analyser. Plusieurs « autovacuum worker » peuvent être lancé en même temps (le maximum étant indiqué par le paramètre « autovacuum_workers »), ils peuvent même travailler sur la même base, une partie de mémoire partagée leur permettant de savoir sur quelle table les autres « autovacuum workers » travaillent. Bref, sur un système comprenant cinq bases de données et un autovacuum_naptime à une minute, une base de données se verra analysée toutes les minutes.

L'exécution d'un VACUUM ou d'un ANALYZE sur une table dépend d'un calcul précis. Ce calcul se fait avec une limite minimum (autovacuum_vacuum_threshold) de lignes modifiées et un facteur d'échelle (autovacuum_vacuum_scale_factor). Cela permet respectivement d'éviter d'exécuter un VACUUM sur une table quand très peu de lignes ont été modifiés et en même temps de tenir compte de la taille de la table. L'algorithme est donc le suivant : si autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * le nombre de lignes de la table est plus petit que le nombre de lignes réellement modifiées ou supprimées, alors un VACUUM est exécuté sur cette table.

Note : le processus autovacuum est aussi capable d'exécuter des ANALYZE automatiquement, mais ces derniers dépendent des paramètres autovacuum_analyze_threshold et autovacuum_analyze_scale_factor.

Le VACUUM de l'autovacuum souffre des mêmes problèmes que le VACUUM manuel. Il dispose donc d'un traitement particulier sur le délai d'endormissement (autovacuum_vacuum_cost_delay) et la limite de coût (autovacuum_vacuum_cost_limit).

Toute cette configuration est globale aux bases de données gérées par ce serveur. Il est néanmoins possible d'avoir une configuration plus fine, directement au niveau de chaque table de chaque base. Pour cela, il faut renseigner le catalogue système pg_autovacuum. Ce catalogue contient des colonnes ressemblant beaucoup aux options déjà décrites :

- vacrelid, OID de la table concernée ;
- enabled, s'il vaut true, cette configuration est utilisée, sinon elle est ignorée et la configuration globale est utilisée ;
- vac_base_thresh, équivalent du autovacuum_vacuum_threshold, limite minimale du nombre de lignes modifiées avant d'exécuter un VACUUM ;
- vac_scale_factor, équivalent du autovacuum_vacuum_scale_factor, facteur d'échelle ;
- anl_base_thresh, équivalent du autovacuum_analyze_threshold, limite minimale du nombre de lignes modifiées avant d'exécuter un ANALYZE ;

anl_scale_factor, équivalent du autovacuum_analyze_scale_factor, facteur d'échelle ;

- vac_cost_delay, durée d'endormissement du VACUUM après dépassement de la limite de coût ;
- vac_cost_limit, limite de coût ;
- freeze_min_age ;
- freeze_max_age.

L'un des gros inconvénients de l'autovacuum à son arrivée en version 8.1 est qu'il était impossible de savoir ce qu'il faisait. Un message dans les traces indiquait seulement que l'autovacuum était lancé sur une base et nous pouvions le voir s'exécuter tous les autovacuum_naptime secondes. Mais il était impossible de savoir s'il exécutait des VACUUM et quelles tables étaient traitées. En fait, c'était possible en passant le niveau des traces en DEBUG2 ce qui générait bien trop d'informations pour être réellement utilisable en permanence. La version 8.2 a un peu amélioré la situation en ajoutant les colonnes last_vacuum, last_autovacuum, last_analyze et last_autoanalyze dans le catalogue système pg_stat_user_tables. Elles permettent de connaître les date et heure, respectivement, du dernier VACUUM manuel, du dernier VACUUM exécuté par l'autovacuum, du dernier ANALYZE manuel et du dernier ANALYZE exécuté par l'autovacuum. Mais il est difficile de connaître la fréquence de traitement d'une table sans scruter ces colonnes en permanence. La version 8.3 apporte une amélioration de taille. Le paramètre log_autovacuum_min_duration permet de connaître l'activité de tout processus autovacuum dont la durée d'exécution dépasse ce délai. À -1, la fonctionnalité est désactivée. À 0, tout est tracé. Voici un exemple de ce qu'on obtient dans les journaux applicatifs en activant ce paramètre :

```
2008-02-22 12:59:34 EET [23258]: [2-1] LOG:  automatic vacuum of table "t": index scans: 0
        pages: 0 removed, 15150 remain
        tuples: 0 removed, 2123107 remain
        system usage: CPU 0.01s/0.00u sec elapsed 151.48 sec
```

La trace indique notamment le nombre de pages supprimées et restantes, le nombre de lignes supprimées et restants, ainsi que le temps d'utilisation du processeur, et le durée totale d'exécution.

Que choisir : cron ou autovacuum ?

La vraie question serait plutôt : pourquoi choisir ? Cron permet de traiter toutes les tables à un moment où le système est tranquille et autovacuum permet de traiter dans la journée les tables les plus impactées. Les deux sont donc importants. Il n'y a pas de choix particulier à faire, mais plutôt de configurer les deux.

Nouveautés en 8.3, et prévisions pour la 8.4

La 8.3 a ajouté de nombreuses améliorations pour l'autovacuum : enfin activé par défaut, il se comporte maintenant comme un démon et peut exécuter plusieurs « autovacuum workers » pour traiter les différentes bases de données. Il précise l'heure de début de son exécution dans la table pg_stat_activity, et trace les processus autovacuum plus longs qu'une certaine durée configurable.

La version 8.4 va aussi améliorer certains points. Le gros du travail semble actuellement se faire autour de la structure FSM. Heikki Linnakangas travaille notamment pour ne plus avoir à configurer la taille de cette structure. De plus, Alvaro Herrera a modifié l'autovacuum pour faire en sorte que tables HEAP et tables TOAST soient dissociables au niveau de l'autovacuum. Ce dernier pourra faire un VACUUM uniquement sur la partie HEAP s'il l'estime nécessaire.

Conclusion

Comme nous venons de le voir, après avoir installé un serveur PostgreSQL, il est nécessaire de planifier l'exécution périodique de trois opérations de maintenance : ANALYZE pour avoir des statistiques à jour et permettre ainsi à l'optimiseur d'exécuter plus rapidement les requêtes, VACUUM et REINDEX pour s'assurer que les tables et index ne se fragmentent pas trop.

L'utilisation du processus autovacuum est fortement recommandée pour gérer l'exécution des opérations ANALYZE et VACUUM tout au long de la journée, pour les tables qui en ont réellement besoin.