Transactions et accès concurrents

Bases de données 2

Thibaut MADELAINE

janvier 2022

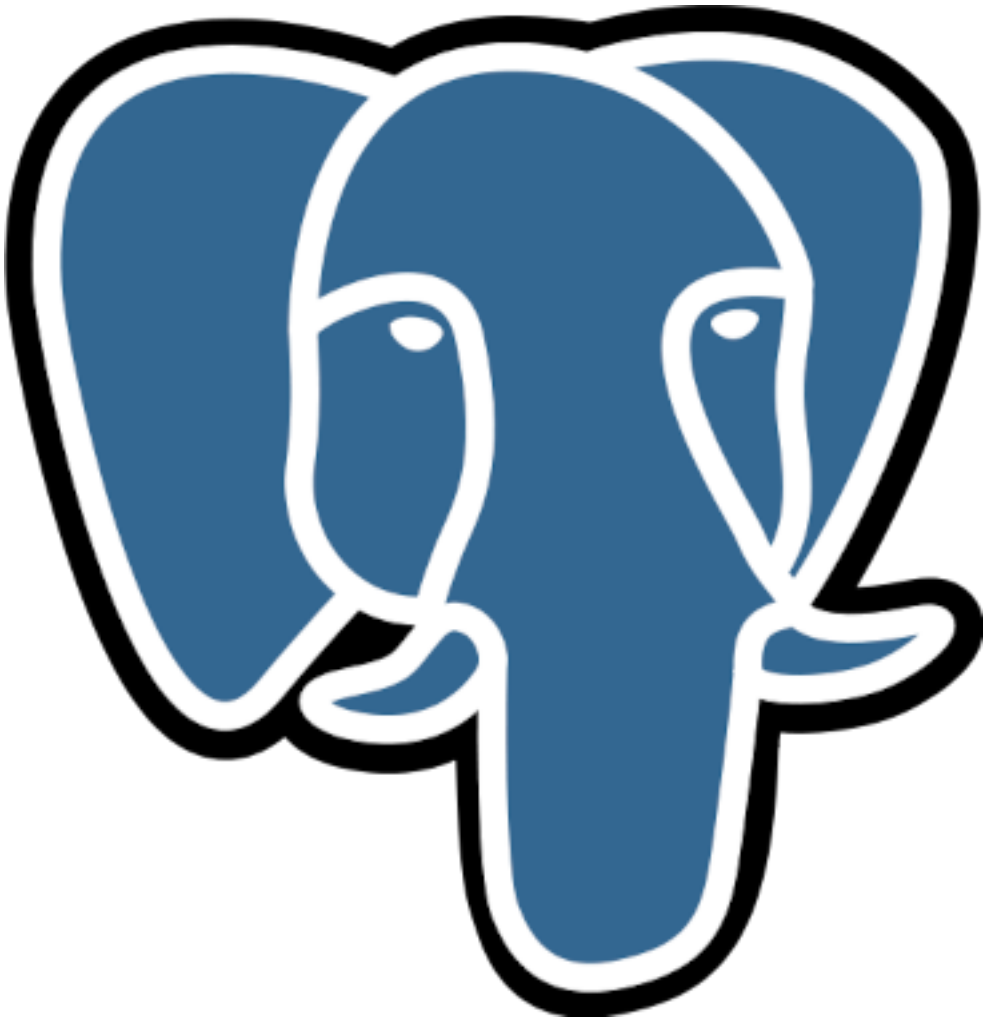
Chers lectrices & lecteurs,

Cette formation PostgreSQL est issue des manuels Dalibo. Ils ont été repris par Thibaut MADELAINE pour rentrer dans le format universitaire avec Cours Magistraux, Travaux Dirigés (sans ordinateurs) et Travaux Pratiques (avec ordinateur).

Au-delà du contenu technique en lui-même, l'intention des auteurs est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de cette formation 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 sur le site gitlab https://gitlab.com/madtibo/cours_dba_pg_universite/-/issues !

Transactions et accès concurrents



Programme de ce cours

- Semaine 1 : découverte de PostgreSQL
- **Semaine 2** : transactions et accès concurrents
  - MultiVersion Concurrency Control
  - Les transactions dans PostgreSQL
  - Le fonctionnement du *VACUUM*
  - Gestion des verrous
- Semaine 3 : missions du DBA
- Semaine 4 : optimisation et indexation

MultiVersion Concurrency Control (MVCC)

- Le « noyau » de PostgreSQL
- Garantit ACID
- Permet les écritures concurrentes sur la même table

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

C'est notamment MVCC qui permet de sauvegarder facilement une base *à chaud* et d'obtenir une sauvegarde cohérente alors même que plusieurs utilisateurs sont potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

MVCC et les verrous

- Une lecture ne bloque pas une écriture
- Une écriture ne bloque pas une lecture

- Une écriture ne bloque pas les autres écritures…
- …sauf pour la mise à jour de la **même ligne**.

MVCC maintient toutes les versions nécessaires de chaque tuple, ainsi **chaque transaction voit une image figée de la base** (appelée *snapshot*). Cette image correspond à l'état de la base lors du démarrage de la requête ou de la transaction, suivant le niveau d'*isolation* demandé par l'utilisateur à PostgreSQL pour la transaction.

MVCC fluidifie les mises à jour en évitant les blocages trop contraignants (verrous sur UPDATE) entre sessions et par conséquent de meilleures performances en contexte transactionnel.

Voici un exemple concret :

```
# SELECT now();
              now
-------------------------------
 2022-01-23 16:28:13.679663+02
(1 row)

# BEGIN;
BEGIN
# SELECT now();
             now
-------------------------------
 2022-01-23 16:28:34.888728+02
(1 row)

# SELECT pg_sleep(2);
 pg_sleep
----------

(1 row)

# SELECT now();
             now
-------------------------------
 2022-01-23 16:28:34.888728+02
(1 row)
```

Transactions

- Intimement liées à ACID et MVCC :
  - Une transaction est un ensemble d'opérations atomique
  - Le résultat d'une transaction est « tout ou rien »
- mots clés `BEGIN`, `COMMIT` et `ROLLBACK`

Voici un exemple de transaction:

```
=> BEGIN;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1, 'Haddock', 35);
INSERT 0 1
=> SELECT age FROM capitaines;
 age
-----
  35
(1 ligne)

=> ROLLBACK;
ROLLBACK
=> SELECT age FROM capitaines;
ERROR:  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines;
                        ^
```

On voit que la table capitaine a existé **à l'intérieur** de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final. Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL.

```
=> BEGIN;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1,'Haddock',35);
INSERT 0 1
=> COMMIT;
COMMIT
=> SELECT age FROM capitaines WHERE nom='Haddock';
 age
```

```
-----
  35
(1 row)
```

Grâce au mot clé `COMMIT`, la transaction a été validée. Aucune erreur n'ayant eu lieu, la table est bien créée et l'enregistrement est visible.

```
=> BEGIN;
BEGIN
=> UPDATE capitaines SET age=42;
UPDATE 1
=> SELECT * FROM capitaines;
 id |   nom   | age
----+---------+-----
  1 | Haddock |  42
(1 ligne)

=> DELETE * FROM capitaines;
ERROR:  syntax error at or near "*"
LIGNE 1 : DELETE * FROM capitaines;
                 ^
=> COMMIT;
ROLLBACK
=> SELECT * FROM capitaines;
 id |   nom   | age
----+---------+-----
  1 | Haddock |  35
(1 ligne)
```

Dans ce dernier cas, une erreur a été détectée durant la transaction. La réponse du serveur à la commande de validation, `COMMIT` est `ROLLBACK`. L'ensemble des modifications effectuées lors de cette transaction sont annulées.

Niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point :

  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres

- On peut spécifier le niveau d'isolation au démarrage d'une transaction:
  - `BEGIN ISOLATION LEVEL xxx;`

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

Niveaux d'isolation dans PostgreSQL

- Niveaux d'isolation supportés
  - `READ COMMITED`
  - `REPEATABLE READ`
  - `SERIALIZABLE`

Le standard SQL spécifie quatre niveaux, mais PostgreSQL n'en supporte que trois.

Niveau `READ UNCOMMITED`

- Autorise la lecture de données modifiées mais non validées par d'autres transactions
- Aussi appelé `DIRTY READS` par d'autres moteurs
- Pas de blocage entre les sessions
- Inutile sous PostgreSQL en raison du MVCC
- Si demandé, la transaction s'exécute en `READ COMMITTED`

Ce niveau d'isolation n'est nécessaire que pour les SGBD non-MVCC. Il est très dangereux : on peut lire des données invalides, ou temporaires, puisqu'on lit tous les enregistrements de la table, quel que soit leur état. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

Sous PostgreSQL, ce mode est totalement inutile. Une transaction qui demande le niveau d'isolation `READ UNCOMMITTED` s'exécute en fait en `READ COMMITTED`.

Niveau `READ COMMITTED`

- La transaction ne lit que les données validées en base
- Niveau d'isolation par défaut
- Un ordre SQL s'exécute dans un instantané (les tables semblent figées sur la durée de l'ordre)
- L'ordre suivant s'exécute dans un instantané différent

Ce mode est le mode par défaut, et est suffisant dans de nombreux contextes. PostgreSQL étant MVCC, les écrivains et les lecteurs ne se bloquent pas mutuellement, et chaque ordre s'exécute sur un instantané de la base (ce n'est pas un pré-requis de `READ COMMITTED` dans la norme SQL). On ne souffre plus des lectures d'enregistrements non valides (`dirty reads`). On peut toutefois avoir deux problèmes majeurs d'isolation dans ce mode :

- Les lectures non-répétables (`non-repeatable reads`) : une transaction peut ne pas voir les mêmes enregistrements d'une requête sur l'autre, si d'autres transaction ont validé des modifications entre temps.
- Les lectures fantômes (`phantom reads`) : des enregistrements peuvent ne plus satisfaire une clause `WHERE` entre deux requêtes d'une même transaction.

Niveau `REPEATABLE READ`

- Instantané au début de la transaction
- Ne voit donc plus les modifications des autres transactions
- Voit toujours ses propres modifications
- Peut entrer en conflit avec d'autres transactions en cas de modification des mêmes enregistrements

Ce mode, comme son nom l'indique, permet de ne plus avoir de lectures non-répétables. Deux ordres SQL consécutifs dans la même transaction retourneront les mêmes enregistrements, dans la même version. En lecture seule, ces transactions ne peuvent pas échouer (elles sont entre autres utilisées pour réaliser des exports des données, par pg_dump).

En écriture, par contre (ou `SELECT FOR UPDATE`, `FOR SHARE`), si une autre transaction a modifié les enregistrements ciblés entre temps, une transaction en `REPEATABLE READ` va échouer avec l'erreur suivante :

```
ERROR: could not serialize access due to concurrent update
```

Il faut donc que l'application soit capable de la rejouer au besoin.

Dans la norme, ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements qui ne satisfont plus la même clause `WHERE` entre deux exécutions de requêtes. Cependant, PostgreSQL est plus strict que la norme et ne permet pas ces lectures fantômes en `REPEATABLE READ`.

Niveau `SERIALIZABLE`

- Niveau d'isolation maximum
- Plus de lectures non répétables
- Plus de lectures fantômes
- Instantané au démarrage de la transaction
- Verrouillage informatif des enregistrements consultés (verrouillage des prédicats)
- Erreurs de sérialisation en cas d'incompatibilité

Le niveau `SERIALIZABLE` permet de développer comme si chaque transaction se déroulait seule sur la base. En cas d'incompatibilité entre les opérations réalisées par plusieurs transactions, PostgreSQL annule celle qui déclenchera le moins de perte de données. Tout comme dans le mode `REPEATABLE READ`, il est essentiel de pouvoir rejouer une transaction si on développe en mode `SERIALIZABLE`. Par contre, on simplifie énormément tous les autres points du développement.

Ce mode empêche les erreurs dues à une transaction effectuant un `SELECT` d'enregistrements, puis d'autres traitements, pendant qu'une autre transaction modifie les enregistrements vus par le `SELECT` : il est probable que le `SELECT` initial de notre transaction utilise les enregistrements récupérés, et que le reste du traitement réalisé par notre transaction dépende de ces enregistrements. Si ces enregistrements sont modifiés par une transaction concurrente, notre transaction ne s'est plus déroulée comme si elle était seule sur la base, et on a donc une violation de sérialisation.

L'implémentation MVCC de PostgreSQL

- Colonnes `ctid`/`xmin`/`xmax`
- Fichiers `clog`
- Avantages/inconvénients
- Opération `VACUUM`

- `Wrap-Around`

`ctid`

- Colonne masquée par défaut
- Codée sur 6 octets
  - 4 octets pour la page
  - 2 octets pour la ligne
- Fournit une adresse physique dans une table

La localisation physique de la version de ligne au sein de sa table. Bien que le `ctid` puisse être utilisé pour trouver la version de ligne très rapidement, le `ctid` d'une ligne change si la ligne est actualisée ou déplacée par un `VACUUM FULL`. Le `ctid` est donc inutilisable comme identifiant de ligne sur le long terme.

---

xmin et xmax (1/4)

Table initiale :

xmin	xmax	Nom	Solde
100		M. Durand	1500
100		Mme Martin	2200

PostgreSQL stocke des informations de visibilité dans chaque version d'enregistrement.

- `xmin` : l'identifiant de la transaction créant cette version.
- `xmax` : l'identifiant de la transaction invalidant cette version.

Ici, les deux enregistrements ont été créés par la transaction 100. Il s'agit peut-être, par exemple, de la transaction ayant importé tous les soldes à l'initialisation de la base.

---

xmin et xmax (2/4)

```
BEGIN;
UPDATE soldes SET solde=solde-200 WHERE nom = 'M. Durand';
```

xmin	xmax	Nom	Solde
100	**150**	M. Durand	1500
100		Mme Martin	2200
150		M. Durand	1300

On décide d'enregistrer un virement de 200 € du compte de M. Durand vers celui de Mme Martin. Ceci doit être effectué dans une seule transaction : l'opération doit être atomique, sans quoi de l'argent pourrait apparaître ou disparaître de la table.

Nous allons donc tout d'abord démarrer une transaction (ordre SQL `BEGIN`). PostgreSQL fournit donc à notre session un nouveau numéro de transaction (150 dans notre exemple). Puis nous effectuerons :

```
UPDATE soldes SET solde=solde-200 WHERE nom = 'M. Durand';
```

---

xmin et xmax (3/4)

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'Mme Martin';
```

xmin	xmax	Nom	Solde
100	150	M. Durand	1500
100	**150**	Mme Martin	2200
150		M. Durand	1300
150		Mme Martin	2400

Puis nous effectuerons :

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'Mme Martin';
```

Nous avons maintenant deux versions de chaque enregistrement.

Notre session ne voit bien sûr plus que les nouvelles versions de ces enregistrements, sauf si elle décidait d'annuler la transaction, auquel cas elle reverrait les anciennes données.

Pour une autre session, la version visible de ces enregistrements dépend de plusieurs critères :

- La transaction 150 a-t-elle été validée ? Sinon elle est invisible
- La transaction 150 est-elle *postérieure* à la nôtre (numéro supérieur au notre), et sommes-nous dans un niveau d'isolation (*serializable*) qui nous interdit de voir les modifications faites depuis le début de notre transaction ?
- La transaction 150 a-t-elle été validée après le démarrage de la requête en cours ? Une requête, sous PostgreSQL, voit un instantané cohérent de la base, ce qui implique que toute transaction validée après le démarrage de la requête doit être ignorée.

Dans le cas le plus simple, 150 ayant été validée, une transaction 160 ne verra pas les premières versions : `xmax` valant 150, ces enregistrements ne sont pas visibles. Elle verra les secondes versions, puisque `xmin`=150, et pas de `xmax`.

---

xmin et xmax (4/4)

xmin	xmax	Nom	Solde
100	150	M. Durand	1500
100	**150**	Mme Martin	2200
150		M. Durand	1300
150		Mme Martin	2400

- Comment est effectuée la suppression d'un enregistrement ?
- Comment est effectuée l'annulation de la transaction 150 ?

- La suppression d'un enregistrement s'effectue simplement par l'écriture d'un `xmax` dans la version courante.
- Il n'y a rien à écrire dans les tables pour annuler une transaction. Il suffit de marquer la transaction comme étant annulée dans la `CLOG`.

CLOG

- La CLOG (Commit Log) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de CLOG

La CLOG est stockée dans une série de fichiers de 256 ko, stockés dans le répertoire pg_xact de PGDATA (répertoire racine de l'instance PostgreSQL).

Chaque transaction est créée dans ce fichier dès son démarrage et est encodée sur deux bits puisqu'une transaction peut avoir quatre états.

- TRANSACTION_STATUS_IN_PROGRESS : transaction en cours, c'est l'état initial
- TRANSACTION_STATUS_COMMITTED : la transaction a été validée
- TRANSACTION_STATUS_ABORTED : la transaction a été annulée
- TRANSACTION_STATUS_SUB_COMMITTED : ceci est utilisé dans le cas où la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

On a donc un million d'états de transactions par fichier de 256 ko.

Annuler une transaction (ROLLBACK) est quasiment instantané sous PostgreSQL : il suffit d'écrire TRANSACTION_STATUS_ABORTED dans l'entrée de CLOG correspondant à la transaction.

Toute modification dans la CLOG, comme toute modification d'un fichier de données (table, index, séquence), est bien sûr enregistrée tout d'abord dans les journaux de transactions (fichiers WAL dans le répertoire pg_wal).

Avantages du MVCC PostgreSQL

- Avantages :
  - avantages classiques de MVCC (concurrence d'accès)
  - implémentation simple et performante
  - peu de sources de contention
  - verrouillage simple d'enregistrement
  - rollback instantané
  - données conservées aussi longtemps que nécessaire
- Les lecteurs ne bloquent pas les écrivains, ni les écrivains les lecteurs.

- Le code gérant les instantanés est simple, ce qui est excellent pour la fiabilité, la maintenabilité et les performances.
- Les différentes sessions ne se gênent pas pour l'accès à une ressource commune (l'UNDO).
- Un enregistrement est facilement identifiable comme étant verrouillé en écriture : il suffit qu'il ait une version ayant un `xmax` correspondant à une transaction en cours.
- L'annulation est instantanée : il suffit d'écrire le nouvel état de la transaction dans la `clog`. Pas besoin de restaurer les valeurs précédentes, elles redeviennent automatiquement visibles.
- Les anciennes versions restent en ligne aussi longtemps que nécessaire. Elles ne pourront être effacées de la base qu'une fois qu'aucune transaction ne les considérera comme visibles.

Inconvénients du MVCC PostgreSQL

- Inconvénients :
  - Nettoyage des enregistrements (VACUUM)
  - Tables plus volumineuses
  - Pas de visibilité dans les index

Comme toute solution complexe, l'implémentation MVCC de PostgreSQL est un compromis. Les avantages cités précédemment sont obtenus au prix de concessions :

- Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande VACUUM. On peut aussi voir ce point comme un avantage : contrairement à la solution UNDO, ce travail de nettoyage n'est pas effectué par le client faisant des mises à jour (et créant donc des enregistrements morts). Le ressenti est donc meilleur.
- Les tables sont forcément plus volumineuses que dans l'implémentation par UNDO, pour deux raisons :
  - Les informations de visibilité qui y sont stockées. Il y a un surcoût d'une douzaine d'octets par enregistrement.
  - Il y a toujours des enregistrements morts dans une table, une sorte de *fond de roulement*, qui se stabilise quand l'application est en régime stationnaire. Ces enregistrements sont recyclés à chaque passage de VACUUM.
- Les index n'ont pas d'information de visibilité. Il est donc nécessaire d'aller vérifier dans la table associée que l'enregistrement trouvé dans l'index est bien visible. Cela a un impact sur le temps d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes.

- Le VACUUM ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale).

Fonctionnement de VACUUM (1/3)

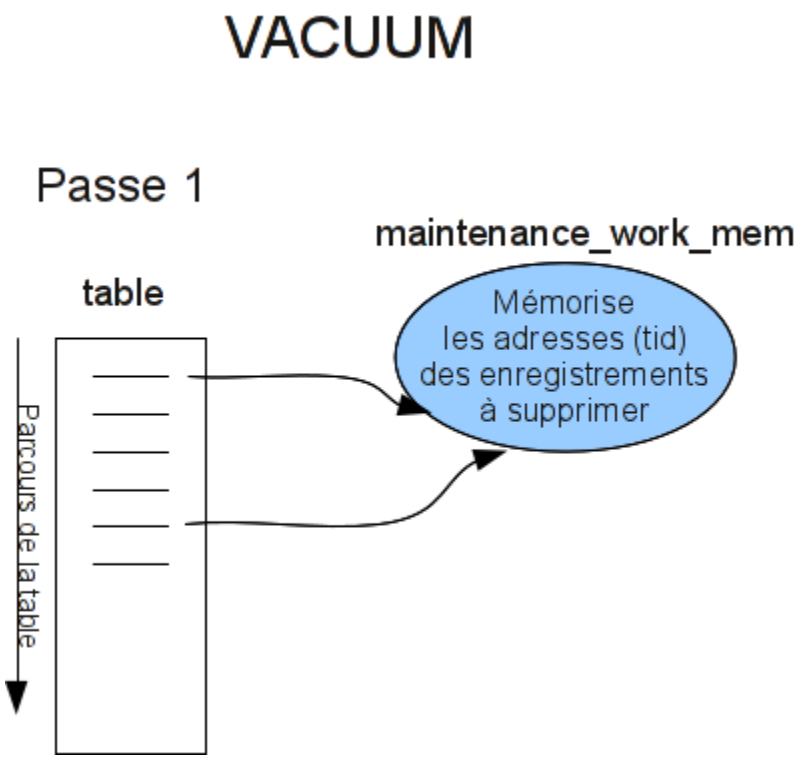


Figure 1: Algorithme du vacuum 1/3

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.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. Les adresses (`tid` ou `tuple id`) des enregistrements sont donc mémorisés par la session effectuant le vacuum, dans un espace mémoire dont la taille est à hauteur de `maintenance_work_mem`. Si `maintenance_work_mem` est trop petit pour contenir tous les enregistrements morts en une seule passe, vacuum effectue plusieurs séries de ces trois passes.

Un `tid` est composé du numéro de bloc et du numéro d'enregistrement dans le bloc.

Fonctionnement de VACUUM (2/3)

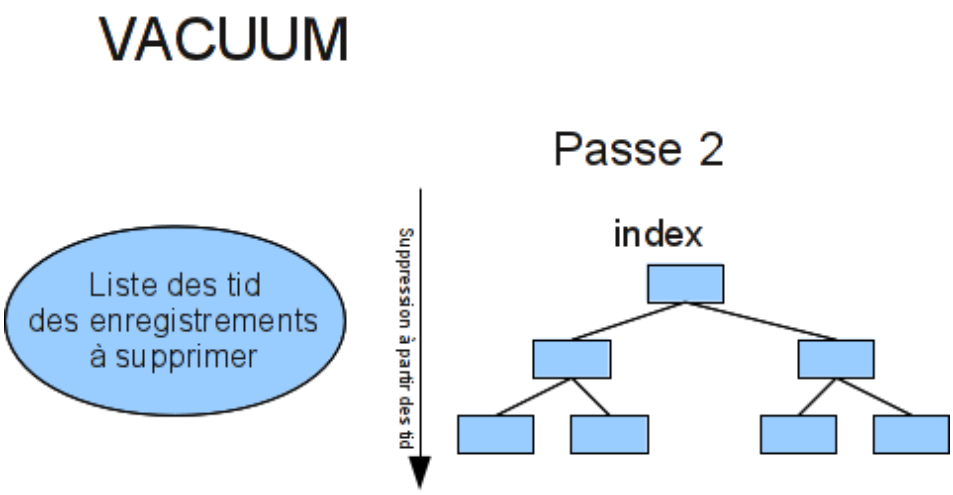


Figure 2: Algorithme du vacuum 2/3

La seconde passe se charge de nettoyer les entrées d'index. Vacuum possède une liste de `tid` 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`).

Fonctionnement de VACUUM (3/3)

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements identifiés, nous pouvons supprimer les enregistrements de la table elle-même. C'est le rôle de cette passe, qui quant à elle, peut accéder directement aux enregistrements. Quand un enregistrement est supprimé d'un bloc, ce bloc est réorganisé afin de consolider l'espace libre, et cet espace libre est consolidé 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 1 (la `maintenance_work_mem` était pleine), le travail reprend où il en était du parcours de la table.

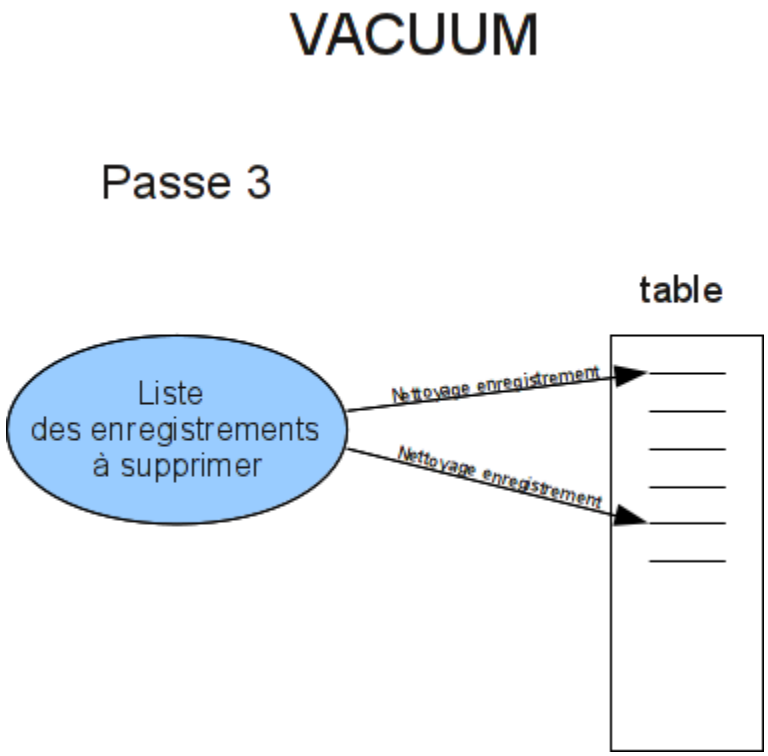


Figure 3: Algorithme du vacuum 3/3

Le problème du Wraparound

Wraparound : bouclage d'un compteur

- Le compteur de transactions : 32 bits
- 4 milliards de transactions
- Qu'arrive-t-il si on boucle ?
- Quelles protections ?

Le compteur de transactions de PostgreSQL est stocké sur 32 bits. Il peut donc, en théorie, y avoir un dépassement de ce compteur au bout de 4 milliards de transactions. En fait, le compteur est cyclique, et toute transaction considère que les 2 milliards de transactions supérieures à la sienne sont dans le futur, et les 2 milliards inférieures dans le passé. Le risque de bouclage est donc plus proche des 2 milliards.

En théorie, si on bouclait, de nombreux enregistrements deviendraient invisibles, car validés par des transactions futures. Heureusement PostgreSQL l'empêche. Au fil des versions, la protection est devenue plus efficace.

- Le moteur trace la plus vieille transaction d'une base, et refuse toute nouvelle transaction à partir du moment où le stock de transaction disponible est à 10 millions. Il suffit de lancer un `VACUUM` sur la base incriminée à ce point, qui débloquera la situation, en nettoyant les plus anciens `xmin`.
- Depuis l'arrivée d'`AUTOVACUUM`, celui-ci déclenche automatiquement un `VACUUM` quand le *Wraparound* se rapproche trop. Ceci se produit même si `AUTOVACUUM` est désactivé.

Si vous voulez en savoir plus, la documentation officielle[1] contient un paragraphe sur ce sujet.

Ce qu'il convient d'en retenir, c'est que le système empêchera le wraparound en se bloquant. En cas de blocage, une protection contre ce phénomène se déclenche automatiquement en déclenchant un `VACUUM`, quel que soit le paramétrage d'`AUTOVACUUM`.

---

commande VACUUM

- Lancement manuel du `VACUUM`

[1] https://docs.postgresql.fr/current/maintenance.html

- option FULL
  - défragmente la table
  - verrou exclusif
- option ANALYZE
  - met en plus à jour les statistiques

La commande VACUUM permet de récupérer l'espace utilisé par les lignes non visibles afin d'éviter un accroissement continuel du volume occupé sur le disque.

Cependant, un VACUUM fait rarement gagner de l'espace disque. Il faut utiliser l'option FULL pour cela. La commande VACUUM FULL libère l'espace consommé par les lignes périmées ou les colonnes supprimées, et le rend au système d'exploitation.

Cette variante de la commande VACUUM acquiert un verrou exclusif sur chaque table. Elle peut donc avoir un effet extrêmement négatif sur les performances de la base de données.

Quand faut-il utiliser VACUUM ?

- pour des nettoyages réguliers ;
- il s'agit d'une maintenance de base.

Quand faut-il utiliser VACUUM FULL ?

- après des suppressions massives de données ;
- lorsque la base n'est pas en production ;
- il s'agit d'une maintenance exceptionnelle.

Des VACUUM standards et une fréquence modérée sont une meilleure approche que des VACUUM FULL, même non fréquents, pour maintenir des tables mises à jour fréquemment.

VACUUM FULL est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, de façon à ce que la taille de la table soit réduite de façon conséquente.
Un REINDEX est exécuté lors d'un VACUUM FULL.

La commande vacuumdb permet d'exécuter facilement un VACUUM sur une ou toutes les bases, elle permet également la parallélisation de VACUUM sur plusieurs tables.

Verrouillage et MVCC

La gestion des verrous est liée à l'implémentation de MVCC.

- Verrouillage d'objets en mémoire
- Verrouillage d'objets sur disque
- Paramètres

Le gestionnaire de verrous

PostgreSQL possède un gestionnaire de verrous

- Verrous d'objet
- Niveaux de verrouillage
- Deadlock
- Vue pg_locks

PostgreSQL dispose d'un gestionnaire de verrous, comme tout SGBD.

Ce gestionnaire de verrous est capable de gérer des verrous sur des tables, sur des enregistrements, sur des ressources virtuelles. De nombreux types de verrous - 8 - sont disponibles, chacun entrant en conflit avec d'autres.

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler.

Les noms des verrous peuvent prêter à confusion : `ROW SHARE` par exemple est un verrou de table, pas un verrou d'enregistrement. Il signifie qu'on a pris un verrou sur une table pour y faire des `SELECT FOR UPDATE` par exemple. Ce verrou est en conflit avec les verrous pris pour un `DROP TABLE`, ou pour un `LOCK TABLE`.

Le gestionnaire de verrous détecte tout verrou mortel (`deadlock`) entre deux sessions. Un `deadlock` est la suite de prise de verrous entraînant le blocage mutuel d'au moins deux sessions, chacune étant en attente d'un des verrous acquis par l'autre.

On peut accéder aux verrous actuellement utilisés sur un cluster par la vue `pg_locks`.

Verrous sur enregistrement

- Le gestionnaire de verrous possède des verrous sur enregistrements.
- Ils sont :
  - transitoires
  - pas utilisés pour prendre les verrous définitifs
- Utilisation de verrous sur disque.
- Pas de risque de pénurie de verrous.

Le gestionnaire de verrous fournit des verrous sur enregistrement. Ceux-ci sont utilisés pour verrouiller un enregistrement le temps d'y écrire un `xmax`, puis libérés immédiatement.

Le verrouillage réel est implémenté comme suit :

- Chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.
- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. S'il constate que ce `xmax` est celui d'une transaction en cours, il demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction. Qui ne lui est naturellement pas accordé. Il est donc placé en attente.
- Quand la transaction possédant le verrou se termine (`COMMIT` ou `ROLLBACK`), son verrou sur l'objet « identifiant de transaction » est libéré, débloquant ainsi l'autre transaction, qui peut reprendre son travail.

Ce mécanisme ne nécessite pas un nombre de verrous mémoire proportionnel au nombre d'enregistrements à verrouiller, et simplifie le travail du gestionnaire de verrous, celui-ci ayant un nombre bien plus faible de verrous à gérer.

Le mécanisme exposé ici est légèrement simplifié. Pour une explication approfondie, n'hésitez pas à consulter l'article suivant[2] issu de la base de connaissance Dalibo.

Conclusion

- PostgreSQL dispose d'une implémentation MVCC complète, permettant :
  - Que les lecteurs ne bloquent pas les écrivains
  - Que les écrivains ne bloquent pas les lecteurs
  - Que les verrous en mémoire soient d'un nombre limité

[2] https://kb.dalibo.com/verrouillage

- Cela impose par contre une mécanique un peu complexe, dont les parties visibles sont la commande VACUUM et le processus d'arrière-plan Autovacuum.

Travaux Dirigés 2

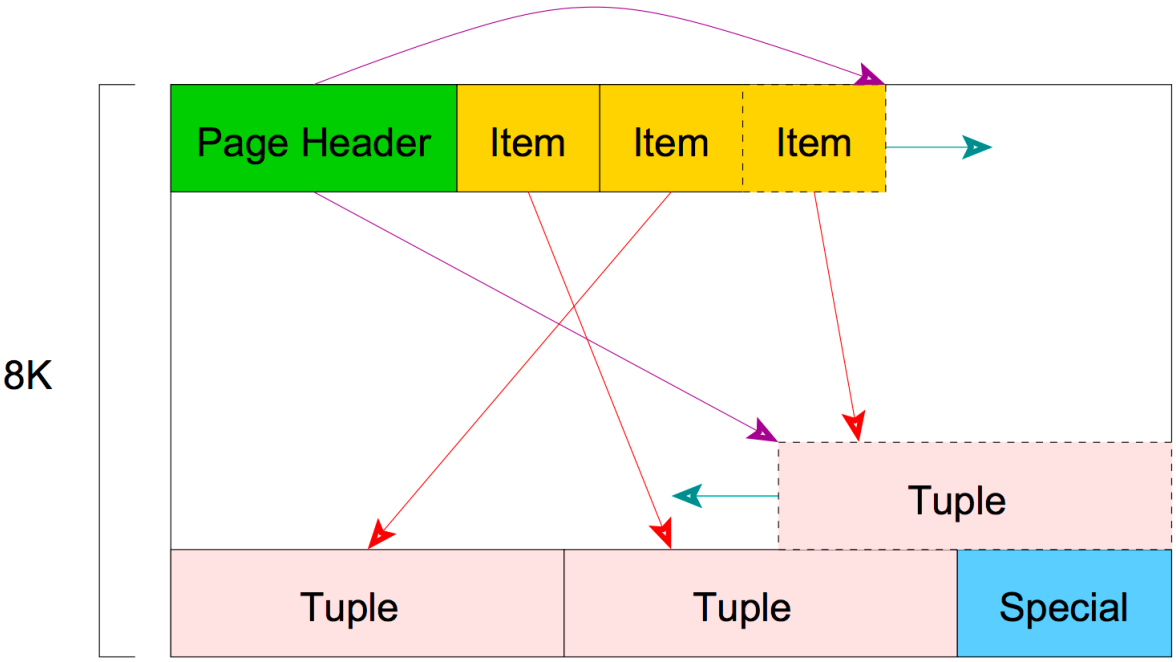
- *MVCC* et Vacuum

Enoncé

Question de cours

- Que signifie l'acronyme ACID ?
- Combien existe-t-il de niveaux d'isolations dans PostgreSQL ?
- Quels sont les inconvénients du MVCC ?

Limites dans PostgreSQL « *block_size (integer): reports the size of a disk block. It is determined by the value of BLCKSZ when building the server. The default value is 8192 bytes. The meaning of some configuration variables (such as shared_buffers) is influenced by block_size.* »



« The `oid` type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. »

« *`ctid`: The physical location of the row version within its table. Note that although the `ctid` can be used to locate the row version very quickly, a row's `ctid` will change if it is updated or moved by VACUUM FULL. Therefore `ctid` is useless as a long-term row identifier. A primary key should be used to identify logical rows.* »

« *`tid`, or tuple identifier (row identifier). This is the data type of the system column `ctid`. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.* »

Définitions dans le fichier source `src/include/storage/block.h` :

```
typedef uint32 BlockNumber;
#define InvalidBlockNumber      ((BlockNumber) 0xFFFFFFFF)
#define MaxBlockNumber          ((BlockNumber) 0xFFFFFFFE)
```

Questions

- Quelle est le nombre maximal de lignes dans une table ?
- Quelle est la taille maximum d'une table sur disque ?

Exploration des `ctid` Soit la table t1 :

```
=# CREATE TABLE t1 (c1 int, c2 text);
CREATE TABLE
=# INSERT INTO t1 (c1, c2) SELECT i, md5(i::text) FROM
 ↪  generate_series(1,1000) i;
INSERT 0 1000
=# SELECT ctid, xmin, xmax, c1, c2 FROM t1;
  ctid   |  xmin   | xmax |  c1  |                c2
---------+---------+------+------+----------------------------------
 (0,1)   | 2760651 |    0 |    1 | c4ca4238a0b923820dcc509a6f75849b
 (0,2)   | 2760651 |    0 |    2 | c81e728d9d4c2f636f067f89cc14862c
 (0,3)   | 2760651 |    0 |    3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
(...)
 (0,119) | 2760651 |    0 |  119 | 07e1cd7dca89a1678042477183b7ac3f
 (0,120) | 2760651 |    0 |  120 | da4fb5c6e93e74d3df8527599fa62642
 (1,1)   | 2760651 |    0 |  121 | 4c56ff4ce4aaf9573aa5dff913df997a
 (1,2)   | 2760651 |    0 |  122 | a0a080f42e6f13b3a2df133f073095dd
(...)
 (8,38)  | 2760651 |    0 |  998 | 9ab0d88431732957a618d4a469a0d4c3
```

```
 (8,39)  | 2760651 |    0 |  999 | b706835de79a2b4e80506f582af3676a
 (8,40)  | 2760651 |    0 | 1000 | a9b7ba70783b617e9998dc4dd82eb3c5

(1000 lignes)
```

Et la table t2 :

```
=# CREATE TABLE t2 (c1 int);
CREATE TABLE
=# INSERT INTO t2 (c1) SELECT i FROM generate_series(1,1000) i;
INSERT 0 1000
=# SELECT ctid, xmin, xmax, c1 FROM t2;
   ctid   |  xmin   | xmax |  c1
----------+---------+------+-------
 (0,1)    | 2760654 |    0 |     1
 (0,2)    | 2760654 |    0 |     2
 (0,3)    | 2760654 |    0 |     3
 (0,4)    | 2760654 |    0 |     4
 (0,5)    | 2760654 |    0 |     5
 (0,6)    | 2760654 |    0 |     6
(...)
 (0,224)  | 2760654 |    0 |   224
 (0,225)  | 2760654 |    0 |   225
 (0,226)  | 2760654 |    0 |   226
 (1,1)    | 2760654 |    0 |   227
 (1,2)    | 2760654 |    0 |   228
 (1,3)    | 2760654 |    0 |   229
(...)
 (4,94)   | 2760657 |    0 |   998
 (4,95)   | 2760657 |    0 |   999
 (4,96)   | 2760657 |    0 |  1000

(1000 lignes)
```

Questions

- Pourquoi le nombre de blocs utilisé est-il différent pour ces 2 tables ?
- Pourquoi la valeur maximum du deuxième champ du `ctid` (l'index du tuple) n'est pas le même pour les 2 tables ?
- Sachant que les informations stockées dans le page header font 24 octets et qu'un item occupe 4 octets, estimer la taille d'un tuple pour ces 2 tables.

Etude du MVCC Session 1 :

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT ctid, xmin, xmax, c1 FROM t2;
```

Session 2 :

```
UPDATE t2 SET c1=c1+1;
```

- Que contient physiquement la table t2 ?

Session 1 :

```
TRUNCATE t2;
```

Session 2 :

```
SELECT ctid, xmin, xmax, c1 FROM t2;
```

- Quel est le résulat de la requête dans la session 2 ?

Session 1 :

```
ROLLBACK;
VACUUM t2;
INSERT INTO t2 (c1) SELECT i FROM generate_series(1002,1500) i;
SELECT ctid, xmin, xmax, c1 FROM t2;
```

- Que contient physiquement la table t2 ?

Travaux Pratiques 2

- *MVCC*, *VACUUM* et verrous

Rappel

Durant ces travaux pratiques, nous allons utiliser la machine virtuelle du TP 1 pour héberger notre serveur de base de données PostgreSQL.

Effectuez les manipulations nécessaires pour réaliser les actions listées dans la section *Énoncés*.

Vous pouvez vous aider du cours, du dernier TD, des précédents TP, ainsi que de l'aide en ligne ou des pages de manuels (`man`).

Enoncés

Effets de MVCC

- Créez une nouvelle base de données et s'y connecter.
- Créez une nouvelle table t1 avec une colonne d'entier et une colonne de texte.
- Ajoutez cinq lignes dans cette table.
- Lisez la table.
- Commencez une transaction et modifiez une ligne.
- Lisez la table.
- Que remarquez-vous ?
- Ouvrez une autre session et lisez la table.
- Qu'observez-vous ?
- Lisez la table ainsi que les informations systèmes *xmin* et *xmax* pour les deux sessions.
- Récupérez maintenant en plus le *ctid*.
- Validez la transaction.
- Installez l'extension *pageinspect*.
- Décodez le bloc 0 de la table t1 à l'aide de cette extension.
- Que remarquez-vous ?

Traiter la fragmentation `VACUUM VERBOSE` :

- Exécutez un `VACUUM VERBOSE` sur la table t1.
- Des lignes ont-elles été nettoyées ?

Suppression en début de table :

- Créez une table t2 avec une colonne de type integer.
- Désactivez l'autovacuum pour cette table.
- Insérez un million de lignes dans cette table.
- Récupérez la taille de la table.
- Supprimez les 500000 premières lignes.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un `VACUUM`.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un `VACUUM FULL`.
- Récupérez la taille de la table. Qu'en déduisez-vous ?

Suppression en fin de table :

- Créez une table t3 avec une colonne de type integer.

- Désactivez l'autovacuum pour cette table.
- Insérez un million de lignes dans cette table.
- Récupérez la taille de la table.
- Supprimez les 500000 dernières lignes.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un `VACUUM`.
- Récupérez la taille de la table. Qu'en déduisez-vous ?

Détecter la fragmentation

- Installez l'extension *pg_freespacemap*.
- Créez une nouvelle table t4 avec une colonne d'entier et une colonne de texte.
- Désactivez l'autovacuum pour cette table.
- Insérer un million de lignes dans cette table.
- Que rapporte *pg_freespacemap* quant à l'espace libre de la table ?
- Modifier les 200 000 premières lignes.
- Que rapporte *pg_freespacemap* quant à l'espace libre de la table ?
- Exécutez un `VACUUM` sur la table.
- Que rapporte *pg_freespacemap* quant à l'espace libre de la table ? Qu'en déduisez-vous ?
- Récupérez la taille de la table.
- Exécutez un `VACUUM FULL` sur la table.
- Récupérez la taille de la table et l'espace libre rapporté par pg_freespacemap. Qu'en déduisez-vous ?

Gestion de l'autovacuum

- Créez une table t5 avec une colonne d'entier.
- Insérer un million de lignes dans cette table.
- Que contient la vue *pg_stat_user_tables* pour cette table ?
- Modifier les 200 000 premières lignes.
- Attendez une minute.
- Que contient la vue _pg_stat_user__tables pour cette table ?
- Modifier 60 lignes supplémentaires de cette table.
- Attendez une minute.
- Que contient la vue pg_stat_user_tables pour cette table ? Qu'en déduisez-vous ?
- Descendez le facteur d'échelle de cette table à 10 % pour le `VACUUM`.
- Modifiez les 200 000 lignes suivantes de cette table ?
- Attendez une minute.

- Que contient la vue *pg_stat_user_tables* pour cette table ? Qu'en déduisez-vous ?

Verrous

- Ouvrez une transaction et lisez la table t1.
- Ouvrez une autre transaction, et tentez de supprimer la table t1.
- Listez les processus du serveur PostgreSQL. Que remarquez-vous ?
- Récupérez la liste des sessions en attente d'un verrou avec la vue *pg_stat_activity*.
- Récupérez la liste des verrous en attente pour la requête bloquée.
- Récupérez le nom de l'objet dont on n'arrive pas à récupérer le verrou.
- Récupérez la liste des verrous sur cet objet. Quel processus a verrouillé la table t1 ?
- Retrouvez les informations sur la session bloquante.
