Module M4

Mécanique du moteur transactionnel & MVCC

24.04

Table des matières

Sur ce document

Formation	Module M4
Titre	Mécanique du moteur transactionnel & MVCC
Révision	24.04
PDF	https://dali.bo/m4_pdf
EPUB	https://dali.bo/m4_epub
HTML	https://dali.bo/m4_html
Slides	https://dali.bo/m4_slides
TP	https://dali.bo/m4_tp
TP (solutions)	https://dali.bo/m4_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.com[1] !

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 Tachoires, 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

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 déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

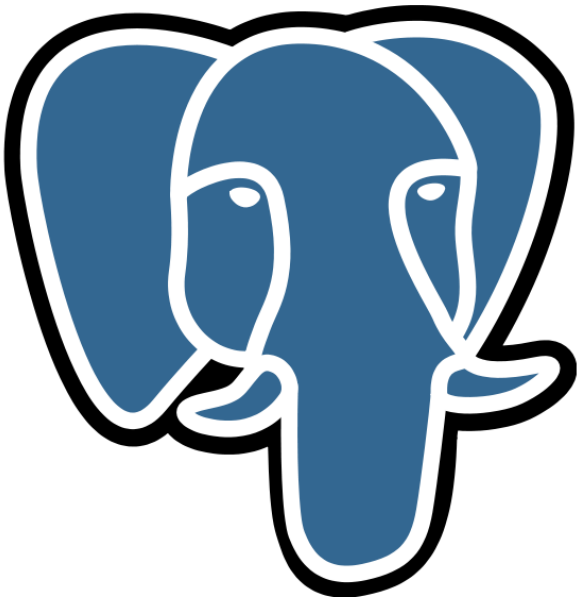
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Mécanique du moteur transactionnel & MVCC



1.1 INTRODUCTION

PostgreSQL utilise un modèle appelé **MVCC** (*Multi-Version Concurrency Control*).

- Gestion concurrente des transactions
- Excellente concurrence
- Impacts sur l'architecture

PostgreSQL s'appuie sur un modèle de gestion de transactions appelé MVCC. Nous allons expliquer cet acronyme, puis étudier en profondeur son implémentation dans le moteur.

Cette technologie a en effet un impact sur le fonctionnement et l'administration de PostgreSQL.

1.2 AU MENU

- Présentation de MVCC
- Niveaux d'isolation
- Implémentation de MVCC de PostgreSQL
- Les verrous
- Le mécanisme TOAST

1.3 PRÉSENTATION DE MVCC

- *MultiVersion Concurrency Control*
- Contrôle de Concurrence Multi-Version
- Plusieurs versions du même enregistrement
- Granularité : l'enregistrement (pas le champ !)

MVCC est un sigle signifiant *MultiVersion Concurrency Control*, ou « contrôle de concurrence multi-version ».

Le principe est de faciliter l'accès concurrent de plusieurs utilisateurs (sessions) à la base en disposant en permanence de plusieurs versions différentes d'un même enregistrement. Chaque session peut travailler simultanément sur la version qui s'applique à son contexte (on parle d'« instantané » ou de *snapshot*).

Par exemple, une transaction modifiant un enregistrement va créer une nouvelle version de cet enregistrement. Mais celui-ci ne devra pas être visible des autres transactions tant que le travail de modification n'est pas validé en base. Les autres transactions *verront* donc une ancienne version de cet enregistrement. La dénomination technique est « lecture cohérente » (*consistent read* en anglais).

Précisons que la granularité des modifications est bien l'enregistrement (ou ligne) d'une table. Modifier un champ (colonne) revient à modifier la ligne. Deux transactions ne peuvent pas modifier deux champs différents d'un même enregistrement sans entrer en conflit, et les verrous portent toujours sur des lignes entières.

1.3.1 Alternative à MVCC : un seul enregistrement en base

- Verrouillage en lecture et exclusif en écriture
- Nombre de verrous ?
- Contention ?
- Cohérence ?
- Annulation ?

Avant d'expliquer en détail MVCC, voyons l'autre solution de gestion de la concurrence qui s'offre à nous, afin de comprendre le problème que MVCC essaye de résoudre.

Une table contient une liste d'enregistrements.

- Une transaction voulant consulter un enregistrement doit le verrouiller (pour s'assurer qu'il n'est pas modifié) de façon partagée, le consulter, puis le déverrouiller.
- Une transaction voulant modifier un enregistrement doit le verrouiller de façon exclusive (personne d'autre ne doit pouvoir le modifier ou le consulter), le modifier, puis le déverrouiller.

Cette solution a l'avantage de la simplicité : il suffit d'un gestionnaire de verrous pour gérer l'accès concurrent aux données. Elle a aussi l'avantage de la performance, dans le cas où les attentes de verrous sont peu nombreuses, la pénalité de verrouillage à payer étant peu coûteuse.

Elle a par contre des inconvénients :

- Les verrous sont en mémoire. Leur nombre est donc probablement limité. Que se passe-t-il si une transaction doit verrouiller 10 millions d'enregistrements ? Des mécanismes de promotion de verrou sont implémentés. Les verrous lignes deviennent des verrous bloc, puis des verrous table. **Le nombre de verrous est limité, et une promotion de verrou peut avoir des conséquences dramatiques** ;
- Un processus devant lire un enregistrement devra attendre la fin de la modification de celui-ci. Ceci entraîne rapidement de gros problèmes de contention. **Les écrivains bloquent les lecteurs, et les lecteurs bloquent les écrivains**. Évidemment, les écrivains se bloquent entre eux, mais cela est normal (il n'est pas possible que deux transactions modifient le même enregistrement simultanément, chacune sans conscience de ce qu'a effectué l'autre) ;
- Un ordre SQL (surtout s'il dure longtemps) n'a aucune garantie de voir des données cohérentes du début à la fin de son exécution : si, par exemple, durant un `SELECT` long, un écrivain modifie à la fois des données déjà lues par le `SELECT`, et des données qu'il va lire, le `SELECT` n'aura pas une vue cohérente de la table. Il pourrait y avoir un total faux sur une table comptable par exemple, le `SELECT` ayant vu seulement une partie des données validées par une nouvelle transaction ;
- Comment annuler une transaction ? Il faut un moyen de défaire ce qu'une transaction a effectué, au cas où elle ne se terminerait pas par une validation mais par une annulation.

1.3.2 Implémentation de MVCC par *undo*

- MVCC par *undo* :
  - une version de l'enregistrement dans la table
  - sauvegarde des anciennes versions
  - l'adresse physique d'un enregistrement ne change pas
  - la lecture cohérente est complexe
  - l'*undo* est complexe à dimensionner… et parfois insuffisant
  - l'annulation est lente
- Exemple : Oracle

C'est l'implémentation d'Oracle, par exemple. Un enregistrement, quand il doit être modifié, est recopié précédemment dans le tablespace d'UNDO. La nouvelle version de l'enregistrement est ensuite écrite par-dessus. Ceci implémente le MVCC (les anciennes versions de l'enregistrement sont toujours disponibles), et présente plusieurs avantages :

- Les enregistrements ne sont pas dupliqués dans la table. Celle-ci ne grandit donc pas suite à une mise à jour (si la nouvelle version n'est pas plus grande que la version précédente) ;
- Les enregistrements gardent la même adresse physique dans la table. Les index correspondant à des données non modifiées de l'enregistrement n'ont donc pas à être modifiés eux-mêmes, les index permettant justement de trouver l'adresse physique d'un enregistrement par rapport à une valeur.

Elle a aussi des défauts :

- La gestion de l'*undo* est très complexe : comment décider ce qui peut être purgé ? Il arrive que la purge soit trop agressive, et que des transactions n'aient plus accès aux vieux enregistrements (erreur `SNAPSHOT TOO OLD` sous Oracle, par exemple) ;
- La lecture cohérente est complexe à mettre en œuvre : il faut, pour tout enregistrement modifié, disposer des informations permettant de retrouver l'image avant modification de l'enregistrement (et la bonne image, il pourrait y en avoir plusieurs). Il faut ensuite pouvoir le reconstituer en mémoire ;
- Il est difficile de dimensionner correctement le fichier d'*undo*. Il arrive d'ailleurs qu'il soit trop petit, déclenchant l'annulation d'une grosse transaction. Il est aussi potentiellement une source de contention entre les sessions ;
- L'annulation (`ROLLBACK`) est très lente : il faut, pour toutes les modifications d'une transaction, défaire le travail, donc restaurer les images contenues dans l'*undo*, les réappliquer aux tables (ce qui génère de nouvelles écritures). Le temps d'annulation peut être supérieur au temps de traitement initial devant être annulé.

1.3.3 L'implémentation MVCC de PostgreSQL

- *Copy On Write* (duplication à l'écriture)
- Une version d'enregistrement n'est jamais modifiée
- Toute modification entraîne une nouvelle version
- Pas d'*undo* : pas de contention, `ROLLBACK` instantané

Dans une table PostgreSQL, un enregistrement peut être stocké dans plusieurs versions. Une modification d'un enregistrement entraîne l'écriture d'une nouvelle version de celui-ci. Une ancienne version ne peut être recyclée que lorsqu'aucune transaction ne peut plus en avoir besoin, c'est-à-dire qu'aucune transaction n'a un instantané de la base plus ancien que l'opération de modification de cet enregistrement, et que cette version est donc invisible pour tout le monde. Chaque version

d'enregistrement contient bien sûr des informations permettant de déterminer s'il est visible ou non dans un contexte donné.

Les avantages de cette implémentation stockant plusieurs versions dans la table principale sont multiples :

- La lecture cohérente est très simple à mettre en œuvre : à chaque session de lire la version qui l'intéresse. La visibilité d'une version d'enregistrement est simple à déterminer ;
- Il n'y a pas d'*undo*. C'est un aspect de moins à gérer dans l'administration de la base ;
- Il n'y a pas de contention possible sur l'*undo* ;
- Il n'y a pas de recopie dans l'*undo* avant la mise à jour d'un enregistrement. La mise à jour est donc moins coûteuse ;
- L'annulation d'une transaction est instantanée : les anciens enregistrements sont toujours disponibles.

Cette implémentation a quelques défauts :

- Il faut supprimer régulièrement les versions obsolètes des enregistrements ;
- Il y a davantage de maintenance d'index (mais moins de contentions sur leur mise à jour) ;
- Les enregistrements embarquent des informations de visibilité, qui les rendent plus volumineux.

1.4 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
- Le niveau d'isolation au démarrage d'une transaction peut être spécifié :
  - `BEGIN ISOLATION LEVEL xxx` ;

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé est 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). Quatre niveaux sont définis, ils ne sont pas tous implémentés par PostgreSQL.

1.4.1 Niveau READ UNCOMMITTED

- Non disponible sous PostgreSQL
  - si demandé, s'exécute en `READ COMMITTED`
- Lecture de données modifiées par d'autres transactions **non** validées
- Aussi appelé *dirty reads*
- Dangereux
- Pas de blocage entre les sessions

Ce niveau d'isolation n'est disponible que pour les SGBD non-MVCC. Il est très dangereux : il est possible de lire des données invalides, ou temporaires, puisque tous les enregistrements de la table sont lus, quels que soient leurs états. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

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

1.4.2 Niveau READ COMMITTED

- Niveau d'isolation par défaut
- La transaction ne lit que les données validées en base
- 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 prérequis de `READ COMMITTED` dans la norme SQL). Il n'y a plus de lectures d'enregistrements non valides (*dirty reads*). Il est toutefois possible d'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 transactions 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.

1.4.3 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 si 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. Ceci est possible car la transaction voit une image de la base figée. L'image est figée non au démarrage de la transaction, mais à la première commande non TCL (*Transaction Control Language*) de la transaction, donc généralement au premier `SELECT` ou à la première modification.

Cette image sera utilisée pendant toute la durée de la transaction. En lecture seule, ces transactions ne peuvent pas échouer. Elles sont entre autres utilisées pour réaliser des exports des données : c'est ce que fait `pg_dump`.

Dans le standard, ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements différents pour une même clause `WHERE` entre deux exécutions de requêtes. Cependant, PostgreSQL est plus strict et ne permet pas ces lectures fantômes en `REPEATABLE READ`. Autrement dit, un même `SELECT` renverra toujours le même résultat.

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.

1.4.4 Niveau SERIALIZABLE

- Niveau d'isolation le plus élevé
- Chaque transaction se croit seule sur la base
  - sinon annulation d'une transaction en cours
- Avantages :
  - pas de « lectures fantômes »
  - évite des verrous, simplifie le développement
- Inconvénients :
  - pouvoir rejouer les transactions annulées
  - toutes les transactions impliquées doivent être sérialisables

PostgreSQL fournit un mode d'isolation appelé `SERIALIZABLE` :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
…
COMMIT / ROLLBACK ;
```

Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base, et comme si elles se déroulaient les unes à la suite des autres. Dès que cette garantie ne peut plus être apportée, PostgreSQL annule celle qui entraînera le moins de perte de données.

Le niveau `SERIALIZABLE` est utile quand le résultat d'une transaction peut être influencé par une transaction tournant en parallèle, par exemple quand des valeurs de lignes dépendent de valeurs d'autres lignes : mouvements de stocks, mouvements financiers… avec calculs de stocks. Autrement dit, si une transaction lit des lignes, elle a la garantie que leurs valeurs ne seront pas modifiées jusqu'à son `COMMIT`, y compris par les transactions qu'elle ne voit pas — ou bien elle tombera en erreur.

Au niveau `SERIALIZABLE` (comme en `REPEATABLE READ`), il est donc essentiel de pouvoir rejouer une transaction en cas d'échec. Par contre, nous simplifions énormément tous les autres points du développement. Il n'y a plus besoin de `SELECT FOR UPDATE`, solution courante mais très gênante pour les transactions concurrentes. Les triggers peuvent être utilisés sans soucis pour valider des opérations.

Ce mode doit être mis en place globalement, car toute transaction non sérialisable peut en théorie s'exécuter n'importe quand, ce qui rend inopérant le mode sérialisable sur les autres.

La sérialisation utilise le « verrouillage de prédicats ». Ces verrous sont visibles dans la vue `pg_locks` sous le nom `SIReadLock`, et ne gênent pas les opérations habituelles, du moins tant que la sérialisation est respectée. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation.

Le wiki PostgreSQL[1], et la documentation officielle[2] donnent des exemples, et ajoutent quelques conseils pour l'utilisation de transactions sérialisables. Afin de tenter de réduire les verrous et le nombre d'échecs :

- faire des transactions les plus courtes possibles (si possible uniquement ce qui a trait à l'intégrité) ;
- limiter le nombre de connexions actives ;
- utiliser les transactions en mode `READ ONLY` dès que possible, voire en `SERIALIZABLE READ ONLY DEFERRABLE` (au risque d'un délai au démarrage) ;
- augmenter certains paramètres liés aux verrous, c'est-à-dire augmenter la mémoire dédiée ; car si elle manque, des verrous de niveau ligne pourraient être regroupés en verrous plus larges et plus gênants ;
- éviter les parcours de tables (*Seq Scan*), et donc privilégier les accès par index.

[1] https://wiki.postgresql.org/wiki/SSI/fr
[2] https://docs.postgresql.fr/current/transaction-iso.html#XACT-SERIALIZABLE

1.5 STRUCTURE D'UN BLOC

- 1 bloc = 8 ko
- `ctid` = (bloc, item dans le bloc)

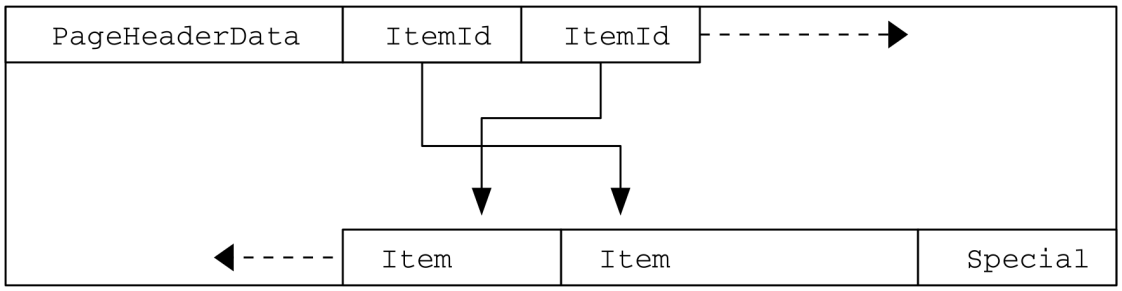


Figure 1/ .1: Répartition des lignes au sein d'un bloc (schéma de la documentation officielle, licence PostgreSQL)

Le bloc (ou page) est l'unité de base de transfert pour les I/O, le cache mémoire… Il fait généralement 8 ko (ce qui ne peut être modifié qu'en recompilant). Les lignes y sont stockées avec des informations d'administration telles que décrites dans le schéma ci-dessus. Une ligne ne fait jamais partie que d'un seul bloc (si cela ne suffit pas, un mécanisme que nous verrons plus tard, nommé TOAST, se déclenche).

Nous distinguons dans ce bloc :

- un entête de page avec diverses informations, notamment la somme de contrôle (si activée) ;
- des identificateurs de 4 octets, pointant vers les emplacements des lignes au sein du bloc ;
- les lignes, stockées à rebours depuis la fin du bloc ;
- un espace spécial, vide pour les tables ordinaires, mais utilisé par les blocs d'index.

Le `ctid` identifie une ligne, en combinant le numéro du bloc (à partir de 0) et l'identificateur dans le bloc (à partir de 1). Comme la plupart des champs administratifs liés à une ligne, il suffit de l'inclure dans un `SELECT` pour l'afficher. L'exemple suivant affiche les premiers et derniers éléments des deux blocs d'une table et vérifie qu'il n'y a pas de troisième bloc :

```
# CREATE TABLE deuxblocs AS SELECT i, i  AS j FROM generate_series(1, 452) i;
SELECT 452
```

```
# SELECT ctid, i,j FROM deuxblocs
WHERE ctid in ( '(1, 1)', '(0, 226)', '(1, 1)', '(1, 226)', '(1, 227)', '(2, 0)' );
```

```
  ctid   |  i  |  j
---------+-----+-----
 (0,1)   |   1 |   1
 (0,226) | 226 | 226
 (1,1)   | 227 | 227
 (1,226) | 452 | 452
```

Un `ctid` ne doit jamais servir à désigner une ligne de manière pérenne et ne doit pas être utilisé dans des requêtes ! Il peut changer n'importe quand, notamment en cas d'`UPDATE` ou de `VACUUM FULL` !

La documentation officielle[3] contient évidemment tous les détails.

Pour observer le contenu d'un bloc, à titre pédagogique, vous pouvez utiliser l'extension pageinspect[4].

[3] https://docs.postgresql.fr/current/storage-page-layout.html
[4] https://docs.postgresql.fr/current/pageinspect.html#id-1.11.7.33.5

1.6 XMIN & XMAX

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.

1.6.1 xmin & xmax (suite)

```
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

Nous décidons 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';
```

1.6.2 xmin & xmax (suite)

```
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 a-t-elle été validée après le démarrage de la transaction en cours, et sommes-nous dans un niveau d'isolation (*repeatable read* ou *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`.

1.6.3 xmin & xmax (suite)

xmin	xmax	Nom	Solde
100	150	M. Durand	1500

xmin	xmax	Nom	Solde
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.

1.7 CLOG

- La CLOG (*Commit Log*) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de CLOG
- `COMMIT` ou `ROLLBACK` très rapide

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` signifie que la transaction en cours, c'est l'état initial ;
- `TRANSACTION_STATUS_COMMITTED` signifie que la la transaction a été validée ;
- `TRANSACTION_STATUS_ABORTED` signifie que la transaction a été annulée ;
- `TRANSACTION_STATUS_SUB_COMMITTED` signifie que la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

Nous avons 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, vue matérialisée), est bien sûr enregistrée tout d'abord dans les journaux de transactions (dans le répertoire `pg_wal/`).

1.8 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

Reprenons les avantages du MVCC tel qu'implémenté par PostgreSQL :

- 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 annulée dans la CLOG. Pas besoin de restaurer les valeurs précédentes, elles sont toujours là ;
- 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.

(Précisons toutefois que ceci est une vision un peu simplifiée pour les cas courants. La signification du `xmax` est parfois altérée par des bits positionnés dans des champs systèmes inaccessibles par l'utilisateur. Cela arrive, par exemple, quand des transactions insèrent des lignes portant une clé étrangère, pour verrouiller la ligne pointée par cette clé, laquelle ne doit pas disparaître pendant la durée de cette transaction.)

1.9 INCONVÉNIENTS DU MVCC POSTGRESQL

- Nettoyage des enregistrements
  - `VACUUM`
  - automatisation : autovacuum
- Tables plus volumineuses
- Pas de visibilité dans les index
- Colonnes supprimées impliquent reconstruction

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

1.9.0.1 VACUUM

Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande `VACUUM`. Il a un avantage sur la technique de l'*undo* : le nettoyage n'est pas effectué par un client faisant des mises à jour (et créant donc des enregistrements morts), et le ressenti est donc meilleur.

`VACUUM` peut se lancer à la main, mais dans le cas général on s'en remet à l'autovacuum, un démon qui lance les `VACUUM` (et bien plus) en arrière-plan quand il le juge nécessaire. Tout cela sera traité en détail par la suite.

1.9.0.2 Bloat

Les tables sont forcément plus volumineuses que dans l'implémentation par *undo*, pour deux raisons :

- les informations de visibilité y sont stockées, il y a donc 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`.

1.9.0.3 Visibilité

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

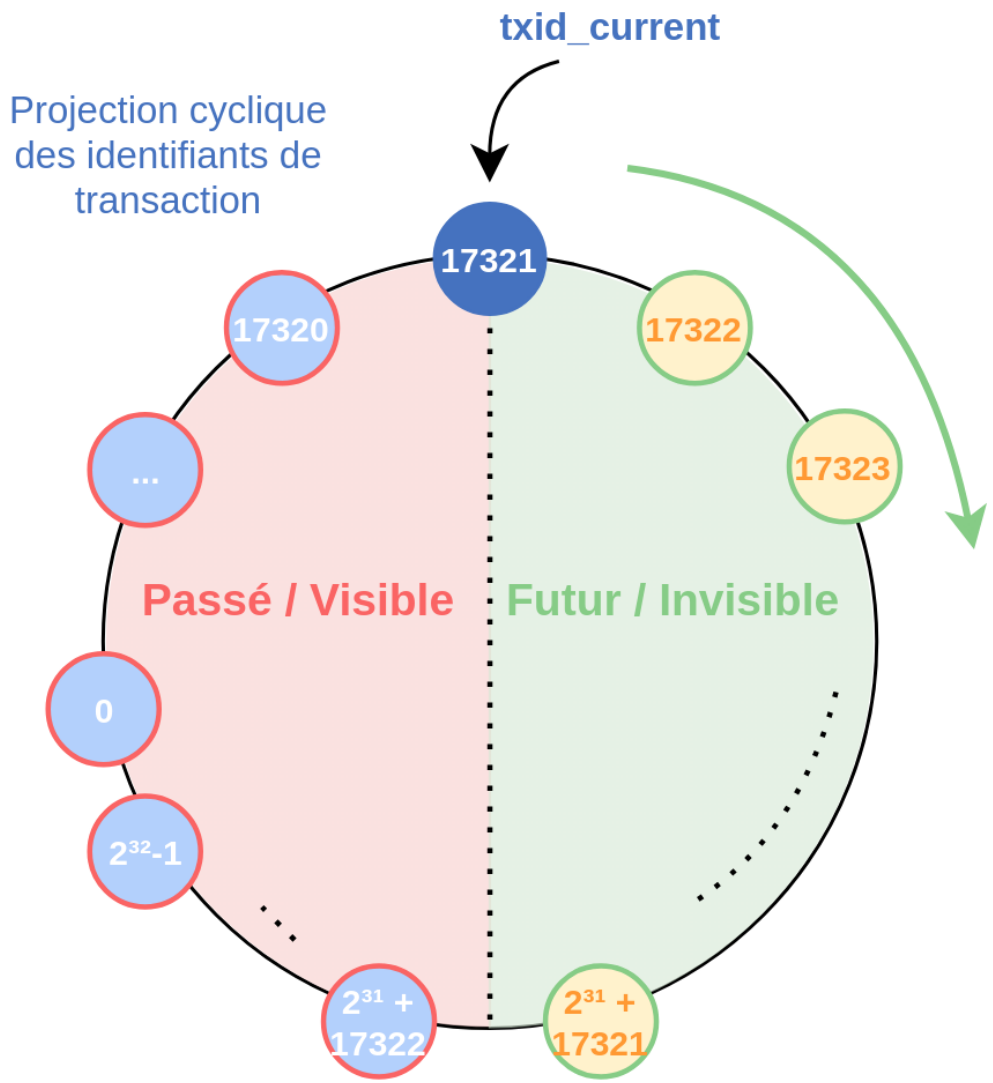
d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes.

1.9.0.4 Colonnes supprimées

Un `VACUUM` ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale). Un `VACUUM FULL` est nécessaire pour reconstruire la table.

1.9.1 Le wraparound (1)

- *Wraparound* : bouclage du compteur de `xmin` / `xmax`
- 32 bits ~ 4 milliards



Le numéro de transaction stocké sur chaque ligne est sur 32 bits, même si PostgreSQL utilise en interne 64 bits. Il y aura donc dépassement de ce compteur au bout de 4 milliards de transactions. Sur les machines actuelles, avec une activité soutenue, cela peut être atteint relativement rapidement.

En fait, ce compteur est cyclique. Cela peut se voir ainsi :

```
SELECT pg_current_xact_id(), *
FROM pg_control_checkpoint()\gx

-[ RECORD 1 ]--------+------------------------
pg_current_xact_id   | 10549379902
checkpoint_lsn       | 62/B902F128
redo_lsn             | 62/B902F0F0
redo_wal_file        | 000000010000006200000B9
timeline_id          | 1
prev_timeline_id     | 1
full_page_writes     | t
next_xid             | 2:1959445310
next_oid             | 24614
next_multixact_id    | 1
next_multi_offset    | 0
oldest_xid           | 1809610092
oldest_xid_dbid      | 16384
oldest_active_xid    | 1959445310
oldest_multi_xid     | 1
oldest_multi_dbid    | 13431
oldest_commit_ts_xid | 0
newest_commit_ts_xid | 0
checkpoint_time      | 2023-12-29 16:39:25+01
```

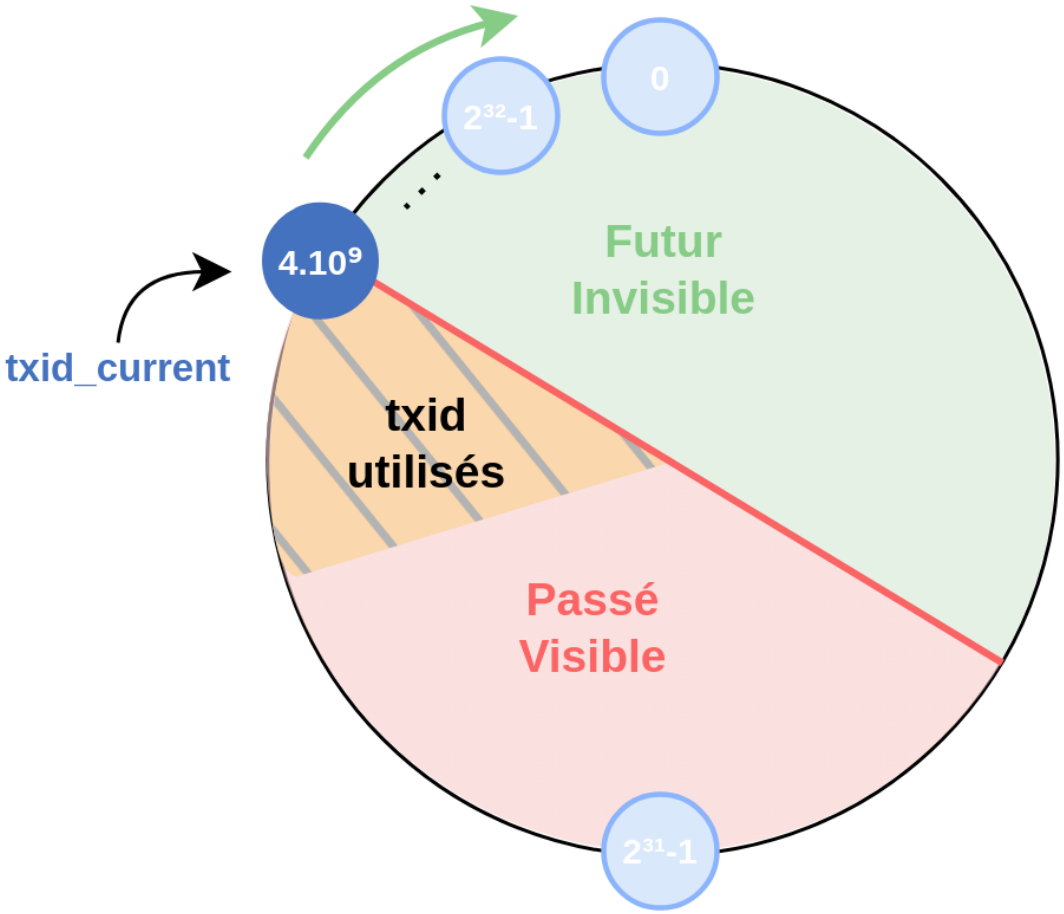
`pg_current_xact_id()` renvoie le numéro de transaction en cours, soit 10 549 379 902 (sur 64 bits). Le premier chiffre de la ligne `next_xid` indique qu'il y a déjà eu deux « époques », et le numéro de transaction sur 32 bits dans ce troisième cycle est 1 959 445 310. On vérifie que 10 549 379 902 = 1 959 445 310 + 2×2^{32}.

Le nombre d'époques n'est enregistré ni dans les tables ni dans les lignes. Pour savoir quelles transactions sont visibles depuis la transaction en cours (ou exactement depuis le « *snapshot* » de l'état de la base associé), PostgreSQL considère que les 2 milliards de numéros de transactions supérieurs à celle en cours correspondent à des transactions futures, et les 2 milliards de numéros inférieurs à des transactions dans le passé.

Si on se contentait de boucler, une fois dépassé le numéro de transaction 2^{31}-1 (environ 2 milliards), de nombreux enregistrements deviendraient invisibles, car validés par des transactions à présent situées dans le futur. Il y a donc besoin d'un mécanisme supplémentaire.

1.9.2 Le wraparound (2)

Après 4 milliards de transactions :



PostgreSQL se débrouille afin de ne jamais laisser dans les tables des numéros de transactions visibles situées dans ce qui serait le futur de la transaction actuelle.

En temps normal, les numéros de transaction visibles se situent tous dans des valeurs un peu inférieures à la transaction en cours. Les lignes plus vieilles sont « gelées », c'est-à-dire marquées comme assez anciennes pour qu'on ignore le numéro de transaction. (Cela implique bien sûr de réécrire la ligne sur le disque.) Il n'y a alors plus de risque de présence de numéros de transaction qui serait abusivement dans le futur de la transaction actuelle. Les numéros de transaction peuvent être réutilisés

sans risque de mélange avec ceux issus du cycle précédent.

Ce cycle des numéros de transactions peut se poursuivre indéfiniment.

Ces recyclages sont visibles dans `pg_control_checkpoint()`, comme vu plus haut. La même fonction renvoie `oldest_xid`, qui est le numéro de la transaction (32 bits) la plus ancienne dans l'instance, qui est par définition celui de la base la plus vieille, ou plutôt de la ligne la plus vieille dans la base :

```
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
```

La ligne la plus ancienne dans la base **pgbench** a moins de 150 millions de transactions d'âge, il n'y a pas de confusion possible.

Un peu plus de 50 millions de transactions plus tard, la même requête renvoie :

```
  datname   | datfrozenxid |   age
------------+--------------+----------
 template0  |   1957896953 | 52338522
 template1  |   1959012415 | 51223060
 postgres   |   1959445305 | 50790170
 pgbench    |   1959625471 | 50610004
```

Les lignes les plus anciennes de `pgbench` semblent avoir disparu.

1.9.3 Le wraparound (3)

Concrètement ?

- `VACUUM FREEZE`
  - géré par l'autovacuum
  - au pire, d'office
  - potentiellement beaucoup d'écritures

Concrètement, l'opération de « gel » des lignes qui possèdent des identifiants de transaction suffisamment anciens est appelée `VACUUM FREEZE`.

Ce dernier peut être déclenché manuellement, mais il fait aussi partie des tâches de maintenance habituellement gérées par le démon `autovacuum`, en bonne partie en même temps que les `VACUUM`

habituels. Un `VACUUM FREEZE` n'est pas bloquant, mais les verrous sont parfois plus gênants que lors d'un `VACUUM` simple. Les écritures générées sont très variables, et parfois gênantes.

Si cela ne suffit pas, le moteur déclenche automatiquement un `VACUUM FREEZE` sur une table avec des lignes trop âgées, et ce, même si autovacuum est désactivé.

Ces opérations suffisent dans l'immense majorité des cas. Dans le cas contraire, l'`autovacuum` devient de plus en plus agressif. Si le stock de transactions disponibles descend en-dessous de 40 millions (10 millions avant la version 14), des messages d'avertissements apparaissent dans les traces.

Dans le pire des cas, après bien des messages d'avertissements, le moteur refuse toute nouvelle transaction dès que le stock de transactions disponibles se réduit à 3 millions (1 million avant la version 14 ; valeurs codées en dur). Il faudra alors lancer un `VACUUM FREEZE` manuellement. Ceci ne peut arriver qu'exceptionnellement (par exemple si une transaction préparée a été oubliée depuis 2 milliards de transactions et qu'aucune supervision ne l'a détectée).

`VACUUM FREEZE` sera développé dans le module VACUUM et autovacuum[5]. La documentation officielle[6] contient aussi un paragraphe sur ce sujet.

[5] https://dali.bo/m5_html
[6] https://docs.postgresql.fr/current/maintenance.html

1.10 OPTIMISATIONS DE MVCC

MVCC a été affiné au fil des versions :

- Mise à jour HOT (*Heap-Only Tuples*)
  - si place dans le bloc
  - si aucune colonne indexée modifiée
- *Free Space Map*
- *Visibility Map*

Les améliorations suivantes ont été ajoutées au fil des versions :

- *Heap-Only Tuples* (HOT) s'agit de pouvoir stocker, sous condition, plusieurs versions du même enregistrement dans le même bloc. Ceci permet au fur et à mesure des mises à jour de supprimer automatiquement les anciennes versions, sans besoin de `VACUUM`. Cela permet aussi de ne pas toucher aux index, qui pointent donc grâce à cela sur plusieurs versions du même enregistrement. Les conditions sont les suivantes :
  - Le bloc contient assez de place pour la nouvelle version (les enregistrements ne sont pas chaînés entre plusieurs blocs). Afin que cette première condition ait plus de chance d'être vérifiée, il peut être utile de baisser la valeur du paramètre `fillfactor` pour une table donnée (cf documentation officielle[7]) ;
  - Aucune colonne indexée n'a été modifiée par l'opération.
- Chaque table possède une *Free Space Map* avec une liste des espaces libres de chaque table. Elle est stockée dans les fichiers `*_fsm` associés à chaque table.
- La *Visibility Map* permet de savoir si l'ensemble des enregistrements d'un bloc est visible. En cas de doute, ou d'enregistrement non visible, le bloc n'est pas marqué comme totalement visible. Cela permet à la phase 1 du traitement de `VACUUM` de ne plus parcourir toute la table, mais uniquement les enregistrements pour lesquels la *Visibility Map* est à *faux* (des données sont potentiellement obsolètes dans le bloc). À l'inverse, les parcours d'index seuls utilisent cette *Visibility Map* pour savoir s'il faut aller voir les informations de visibilité dans la table. `VACUUM` repositionne la *Visibility Map* à *vrai* après nettoyage d'un bloc, si tous les enregistrements sont visibles pour toutes les sessions. Enfin, depuis la 9.6, elle repère aussi les bloc entièrement gelés pour accélérer les `VACUUM FREEZE`.

Toutes ces optimisations visent le même but : rendre `VACUUM` le moins pénalisant possible, et simplifier la maintenance.

[7] https://docs.postgresql.fr/current/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS

1.11 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

1.11.1 Le gestionnaire de verrous

PostgreSQL possède un gestionnaire de verrous

- Verrous d'objet
- Niveaux de verrouillage
- Empilement des verrous
- *Deadlock*
- Vue `pg_locks`

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

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler. Si le verrou ne peut être obtenu immédiatement, par défaut PostgreSQL attendra indéfiniment qu'il soit libéré.

Ce verrou en attente peut lui-même imposer une attente à d'autres sessions qui s'intéresseront au même objet. Si ce verrou en attente est bloquant (cas extrême : un `VACUUM FULL` sans `SKIP_LOCKED` lui-même bloqué par une session qui tarde à faire un `COMMIT`), il est possible d'assister à un phénomène d'empilement de verrous en attente.

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.

Il est possible d'accéder aux verrous actuellement utilisés sur une instance par la vue `pg_locks`.

1.11.2 Verrous sur enregistrement

- Le gestionnaire de verrous possèdes des verrous sur enregistrements
  - transitoires
  - le temps de poser le `xmax`
- Utilisation de verrous sur disque
  - pas de risque de pénurie
- Les verrous entre transaction se font sur leurs ID

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 :

- D'abord, chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.
- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. Si ce `xmax` est celui d'une transaction en cours, elle demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction, qui ne lui est naturellement pas accordé. La transaction est donc placée en attente.
- Enfin, quand l'autre 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 évidemment simplifié.

1.11.3 La vue pg_locks

- `pg_locks` :
  - visualisation des verrous en place
  - tous types de verrous sur objets
- Complexe à interpréter :
  - verrous sur enregistrements pas directement visibles

C'est une vue globale à l'instance.

```
# \d pg_locks
                    Vue « pg_catalog.pg_locks »
      Colonne       |   Type                   | Collationnement | NULL-able | …
--------------------+--------------------------+-----------------+-----------+-
 locktype           | text                     |                 |           |
 database           | oid                      |                 |           |
 relation           | oid                      |                 |           |
 page               | integer                  |                 |           |
 tuple              | smallint                 |                 |           |
 virtualxid         | text                     |                 |           |
 transactionid      | xid                      |                 |           |
 classid            | oid                      |                 |           |
 objid              | oid                      |                 |           |
 objsubid           | smallint                 |                 |           |
 virtualtransaction | text                     |                 |           |
 pid                | integer                  |                 |           |
 mode               | text                     |                 |           |
 granted            | boolean                  |                 |           |
 fastpath           | boolean                  |                 |           |
 waitstart          | timestamp with time zone |                 |           |
```

- `locktype` est le type de verrou, les plus fréquents étant `relation` (table ou index), `transactionid` (transaction), `virtualxid` (transaction virtuelle, utilisée tant qu'une transaction n'a pas eu à modifier de données, donc à stocker des identifiants de transaction dans des enregistrements) ;
- `database` est la base dans laquelle ce verrou est pris ;
- `relation` est l'OID de la relation cible si locktype vaut `relation` (ou `page` ou `tuple`) ;
- `page` est le numéro de la page dans une relation (pour un verrou de type `page` ou `tuple`) cible ;
- `tuple` est le numéro de l'enregistrement cible (quand verrou de type `tuple`) ;
- `virtualxid` est le numéro de la transaction virtuelle cible (quand verrou de type `virtualxid`) ;

- `transactionid` est le numéro de la transaction cible ;
- `classid` est le numéro d'OID de la classe de l'objet verrouillé (autre que relation) dans `pg_class`. Indique le catalogue système, donc le type d'objet, concerné. Aussi utilisé pour les advisory locks ;
- `objid` est l'OID de l'objet dans le catalogue système pointé par `classid` ;
- `objsubid` correspond à l'ID de la colonne de l'objet `objid` concerné par le verrou ;
- `virtualtransaction` est le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si `granted` vaut `f`) ;
- `pid` est le PID (l'identifiant de processus système) de la session possédant le verrou ;
- `mode` est le niveau de verrouillage demandé ;
- `granted` signifie si le verrou est acquis ou non (donc en attente) ;
- `fastpath` correspond à une information utilisée surtout pour le débogage (*fastpath* est le mécanisme d'acquisition des verrous les plus faibles) ;
- `waitstart` indique depuis quand le verrou est en attente.

La plupart des verrous sont de type relation, `transactionid` ou `virtualxid`. Une transaction qui démarre prend un verrou virtualxid sur son propre `virtualxid`. Elle acquiert des verrous faibles (`ACCESS SHARE`) sur tous les objets sur lesquels elle fait des `SELECT`, afin de garantir que leur structure n'est pas modifiée sur la durée de la transaction. Dès qu'une modification doit être faite, la transaction acquiert un verrou exclusif sur le numéro de transaction qui vient de lui être affecté. Tout objet modifié (table) sera verrouillé avec `ROW EXCLUSIVE`, afin d'éviter les `CREATE INDEX` non concurrents, et empêcher aussi les verrouillages manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

1.11.4 Verrous - Paramètres

- Nombre :
  - `max_locks_per_transaction` (+ paramètres pour la sérialisation)
- Durée :
  - `lock_timeout` (éviter l'empilement des verrous)
  - `deadlock_timeout` (défaut 1 s)
- Trace :
  - `log_lock_waits`

Nombre de verrous :

`max_locks_per_transaction` sert à dimensionner un espace en mémoire partagée destinée aux verrous sur des objets (notamment les tables). Le nombre de verrous est :

```
max_locks_per_transaction × max_connections
```

ou plutôt, si les transactions préparées sont activées (et `max_prepared_transactions` monté au-delà de 0) :

```
max_locks_per_transaction × (max_connections + max_prepared_transactions)
```

La valeur par défaut de 64 est largement suffisante la plupart du temps. Il peut arriver qu'il faille le monter, par exemple si l'on utilise énormément de partitions, mais le message d'erreur est explicite.

Le nombre maximum de verrous d'une session n'est pas limité à `max_locks_per_transaction`. C'est une valeur moyenne. Une session peut acquérir autant de verrous qu'elle le souhaite pourvu qu'au total la table de hachage interne soit assez grande. Les verrous de lignes sont stockés sur les lignes et donc potentiellement en nombre infini.

Pour la sérialisation, les verrous de prédicat possèdent des paramètres spécifiques. Pour économiser la mémoire, les verrous peuvent être regroupés par bloc ou relation (voir `pg_locks` pour le niveau de verrouillage). Les paramètres respectifs sont :

- `max_pred_locks_per_transaction` (64 par défaut) ;
- `max_pred_locks_per_page` (par défaut 2, donc 2 lignes verrouillées entraînent le verrouillage de tout le bloc, du moins pour la sérialisation) ;
- `max_pred_locks_per_relation` (voir la documentation[8] pour les détails).

Durées maximales de verrou :

Si une session attend un verrou depuis plus longtemps que `lock_timeout`, la requête est annulée. Il est courant de poser cela avant un ordre assez intrusif, même bref, sur une base utilisée. Par exemple, il faut éviter qu'un `VACUUM FULL`, s'il est bloqué par une transaction un peu longue, ne bloque lui-même toutes les transactions suivantes (phénomène d'empilement des verrous) :

```
postgres=# SET lock_timeout TO '3s' ;
SET
postgres=# VACUUM FULL t_grosse_table ;
ERROR:  canceling statement due to lock timeout
```

Il faudra bien sûr retenter le `VACUUM FULL` plus tard, mais la production n'est pas bloquée plus de 3 secondes.

PostgreSQL recherche périodiquement les *deadlocks* entre transactions en cours. La périodicité par défaut est de 1 s (paramètre `deadlock_timeout`), ce qui est largement suffisant la plupart du temps : les *deadlocks* sont assez rares, alors que la vérification est quelque chose de coûteux. L'une des transactions est alors arrêtée et annulée, pour que les autres puissent continuer :

```
postgres=*# DELETE FROM t_centmille_int WHERE i < 50000;
```

[8] https://docs.postgresql.fr/current/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-RELATION

```
ERROR:  deadlock detected
DÉTAIL : Process 453259 waits for ShareLock on transaction 3010357;
blocked by process 453125.
Process 453125 waits for ShareLock on transaction 3010360;
blocked by process 453259.
ASTUCE : See server log for query details.
CONTEXTE : while deleting tuple (0,1) in relation "t_centmille_int"
```

Trace des verrous :

Pour tracer les attentes de verrous un peu longue, il est fortement conseillé de passer `log_lock_waits` à `on` (le défaut est `off`).

Le seuil est également défini par `deadlock_timeout` (1 s par défaut) Ainsi, une session toujours en attente de verrou au-delà de cette durée apparaîtra dans les traces :

```
LOG:  process 457051 still waiting for ShareLock on transaction 35373775
      after 1000.121 ms
DETAIL:  Process holding the lock: 457061. Wait queue: 457051.
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
```

S'il ne s'agit pas d'un *deadlock*, la transaction continuera, et le moment où elle obtiendra son verrou sera également tracé :

```
LOG:  process 457051 acquired ShareLock on transaction 35373775 after
      18131.402 ms
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
LOG:  duration: 18203.059 ms  statement: DELETE FROM t_centmille_int ;
```

1.12 MÉCANISME TOAST

TOAST : *The Oversized-Attribute Storage Technique*
Que faire si une ligne dépasse d'un bloc ?

Une ligne ne peut déborder d'un bloc, et un bloc fait 8 ko (par défaut). Cela ne suffit pas pour certains champs beaucoup plus longs, comme certains textes, mais aussi des types composés (`json` , `jsonb` , `hstore`), ou binaires (`bytea`), et même `numeric` . PostgreSQL sait compresser alors les champs, mais ça ne suffit pas forcément non plus.

1.12.1 Principe du TOAST

- Table de débordement `pg_toast_XXX`
  - masquée, transparente
- Jusqu'à 1 Go par champ (déconseillé)
  - texte, JSON, binaire…
  - compression optionnelle
- Politiques par champ
  - `PLAIN` / `MAIN` / `EXTERNAL` ou `EXTENDED`

Principe du TOAST :

Le mécanisme TOAST consiste à déporter le contenu de certains champs d'un enregistrement vers une autre table associée à la table principale, gérée de manière transparente pour l'utilisateur. Ce mécanisme permet d'éviter qu'un enregistrement ne dépasse la taille d'un bloc.

Le mécanisme TOAST a d'autres intérêts :

- la partie principale d'une table ayant des champs très longs est moins grosse, alors que les « gros champs » ont moins de chance d'être accédés systématiquement par le code applicatif ;
- ces champs peuvent être compressés de façon transparente, avec souvent de gros gains en place ;
- si un `UPDATE` ne modifie pas un de ces champs « toastés », la table TOAST n'est pas mise à jour : le pointeur vers l'enregistrement de cette table est juste « cloné » dans la nouvelle version de l'enregistrement.

Politiques de stockage :

Chaque champ possède une propriété de stockage :

```
CREATE TABLE unetable (i int, t text, b bytea, j jsonb);
```

```
# \d+ unetable
                    Table « public.unetable »
 Colonne |  Type   | Col... | NULL-able | Par défaut | Stockage | …
---------+---------+--------+-----------+------------+----------+--
 i       | integer |        |           |            | plain    |
 t       | text    |        |           |            | extended |
 b       | bytea   |        |           |            | extended |
 j2      | jsonb   |        |           |            | extended |
Méthode d'accès : heap
```

Les différentes politiques de stockage sont :

- `PLAIN` permettant de stocker uniquement dans la table, sans compression (champs numériques ou dates notamment) ;
- `MAIN` permettant de stocker dans la table tant que possible, éventuellement compressé (politique rarement utilisée) ;
- `EXTERNAL` permettant de stocker éventuellement dans la table TOAST, sans compression ;
- `EXTENDED` permettant de stocker éventuellement dans la table TOAST, éventuellement compressé (cas général des champs texte ou binaire).

Il est rare d'avoir à modifier ce paramétrage, mais cela arrive. Par exemple, certains longs champs (souvent binaires, par exemple du JPEG, des PDF…) se compressent si mal qu'il ne vaut pas la peine de gaspiller du CPU dans cette tâche. Dans le cas extrême où le champ compressé est plus grand que l'original, PostgreSQL revient à la valeur originale, mais du CPU a été consommé inutilement. Il peut alors être intéressant de passer de `EXTENDED` à `EXTERNAL`, pour un gain de temps parfois non négligeable :

```
ALTER TABLE t1 ALTER COLUMN champ SET STORAGE EXTERNAL ;
```

Lors d'un changement de mode de stockage, les données existantes ne sont pas modifiées. Il faut modifier les champs d'une manière ou d'une autre pour forcer le changement.

Les tables pg_toast_XXX :

À chaque table utilisateur se voit associée une table TOAST, et ce dès sa création si la table possède un champ « toastable ». Les enregistrements y sont stockés en morceaux (*chunks*) d'un peu moins de 2 ko. Tous les champs « toastés » d'une table se retrouvent dans la même table `pg_toast_XXX`, dans un espace de nommage séparé nommé `pg_toast`.

Pour l'utilisateur, les tables TOAST sont totalement transparentes. Un développeur doit juste savoir qu'il n'a pas besoin de déporter des champs texte (ou JSON, ou binaires…) imposants dans une table séparée pour des raisons de volumétrie de la table principale : PostgreSQL le fait déjà, et de manière efficace ! Il est donc souvent inutile de se donner la peine de compresser les données au niveau applicatif juste pour réduire le stockage.

Le découpage et la compression restent des opérations coûteuses. Il reste déconseillé de stocker des données binaires de grande taille dans une base de données !

La présence de ces tables n'apparaît guère que dans `pg_class`, par exemple ainsi :

```
SELECT * FROM pg_class c
WHERE c.relname = 'longs_textes'
OR c.oid = (SELECT reltoastrelid FROM pg_class
            WHERE relname = 'longs_textes');

-[ RECORD 1 ]-------+---------------
oid                 | 16614
relname             | longs_textes
relnamespace        | 2200
reltype             | 16616
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16614
reltablespace       | 0
relpages            | 35
reltuples           | 2421
relallvisible       | 35
reltoastrelid       | 16617
…
-[ RECORD 2 ]-------+---------------
oid                 | 16617
relname             | pg_toast_16614
relnamespace        | 99
reltype             | 16618
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16617
reltablespace       | 0
relpages            | 73161
reltuples           | 293188
relallvisible       | 73161
reltoastrelid       | 0
…
```

La partie TOAST est une table à part entière, avec une clé primaire. On ne peut ni ne doit y toucher !

```
\d+ pg_toast.pg_toast_16614

Table TOAST « pg_toast.pg_toast_16614 »
  Colonne   |  Type   | Stockage
------------+---------+----------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain

Table propriétaire : « public.textes_comp »
```

```
Index :
    "pg_toast_16614_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Méthode d'accès : heap
```

L'index est toujours utilisé pour accéder aux *chunks*.

La volumétrie des différents éléments (partie principale, TOAST, index éventuels) peut se calculer grâce à cette requête dérivée du wiki[9] :

```
SELECT
    oid AS table_oid,
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid,
    reltoastrelid::regclass::text AS table_toast,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
    pg_size_pretty(pg_relation_size(c.oid)) AS "  Heap",
    pg_size_pretty(pg_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(reltoastrelid)) AS  "  Toast (PK)",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname = 'longs_textes'
\gx
```

```
-[ RECORD 1 ]------+------------------------
table_oid          | 16614
table              | public.longs_textes
reltoastrelid      | 16617
table_toast        | pg_toast.pg_toast_16614
nb_lignes_estimees | 2421
 Table (dont TOAST) | 578 MB
  Heap              | 280 kB
  Toast             | 572 MB
  Toast (PK)        | 6448 kB
 Index              | 560 kB
Total              | 579 MB
```

La taille des index sur les champs susceptibles d'être toastés est comptabilisée avec tous les index de la table (la clé primaire de la table TOAST est à part).

Les tables TOAST restent forcément dans le même tablespace que la table principale. Leur maintenance (notamment le nettoyage par `autovacuum`) s'effectue en même temps que la table principale, comme le montre un `VACUUM VERBOSE`.

Détails du mécanisme TOAST :

Les détails techniques du mécanisme TOAST sont dans la documentation officielle[10]. En résumé, le mécanisme TOAST est déclenché sur un enregistrement quand la taille d'un enregistrement dépasse 2 ko. Les champs « toastables » peuvent alors être compressés pour que la taille de l'enregistrement redescende en-dessous de 2 ko. Si cela ne suffit pas, des champs sont alors découpés et déportés

[9] https://wiki.postgresql.org/wiki/Disk_Usage
[10] https://doc.postgresql.fr/current/storage-toast.html

vers la table TOAST. Dans ces champs de la table principale, l'enregistrement ne contient plus qu'un pointeur vers la table TOAST associée.

Un champ MAIN peut tout de même être stocké dans la table TOAST, si l'enregistrement dépasse 2 ko : mieux vaut « toaster » que d'empêcher l'insertion.

Cette valeur de 2 ko convient généralement. Au besoin, on peut l'augmenter en utilisant le paramètre de stockage `toast_tuple_target` ainsi :

```
ALTER TABLE t1 SET (toast_tuple_target = 3000);
```

mais cela est rarement utile.

1.12.2 TOAST et compression

- `pglz` (`zlib`) : défauts
- `lz4` à préférer
  - généralement plus rapide
  - compression équivalente (à vérifier)
- Mise en place :

```
default_toast_compression = lz4
```

ou :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

Depuis la version 14, il est possible de modifier l'algorithme de compression. Ceci est défini par le nouveau paramètre `default_toast_compression` dont la valeur par défaut est :

```
SHOW default_toast_compression ;

 default_toast_compression
---------------------------
 pglz
```

c'est-à-dire que PostgreSQL utilise la zlib, seule compression disponible jusqu'en version 13 incluse.

À partir de la version 14, un nouvel algorithme `lz4` est disponible (et intégré dans les paquets distribués par le PGDG).

De manière générale, l'algorithme `lz4` ne compresse pas mieux les données courantes que `pglz`, mais cela dépend des usages. Surtout, `lz4` est **beaucoup** plus rapide à compresser, et parfois à décompresser.
Nous conseillons d'utiliser `lz4` pour la compression de vos TOAST, même si, en toute rigueur, l'arbitrage entre consommations CPU en écriture ou lecture et place disque ne peut se faire qu'en testant soigneusement avec les données réelles.

Pour changer l'algorithme, vous pouvez :

- soit changer la valeur par défaut dans `postgresql.conf` :

```
default_toast_compression = lz4
```

- soit déclarer la méthode de compression à la création de la table :

```
CREATE TABLE t1 (
  c1 bigint GENERATED ALWAYS AS identity,
  c2 text   COMPRESSION lz4
  ) ;
```

- soit après création :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

`lz4` accélère aussi les restaurations logiques comportant beaucoup de données toastées et compressées. Si `lz4` n'a pas été activé par défaut, il peut être utilisé dès le chargement :

```
$ PGOPTIONS='-c default_toast_compression=lz4'  pg_restore …
```

Des lignes d'une même table peuvent être compressées de manières différentes. En effet, l'utilisation de `SET COMPRESSION` sur une colonne préexistante ne recompresse pas les données de la table TOAST associée. De plus, des données toastées lues par une requête, puis réinsérées sans être modifiées, sont recopiées vers les champs cibles telles quelles, sans étapes de décompression/recompression, et ce même si la compression de la cible est différente, même s'il s'agit d'une autre table. Même un `VACUUM FULL` sur la table principale réécrit la table TOAST, en recopiant simplement les champs compressés tels quels.

Pour forcer la recompression de toutes les données d'une colonne, il faut modifier leur contenu. Et en fait, c'est peu intéressant car l'écart de volumétrie est généralement faible. Noter qu'il existe une fonction `pg_column_compression (nom_champ)`[11] pour consulter la compression d'un champ sur chaque ligne concernée.

Pour aller plus loin :

- Blog Dalibo : Les mains dans le cambouis #2 - Le mécanisme de TOAST[12] (2024)
- Blog Fujitsu PostgreSQL : What is the new LZ4 TOAST compression in PostgreSQL 14, and how fast is it?[13] (2021), avec notamment un benchmark montrant l'intérêt de la compression `lz4`.

[11] https://docs.postgresql.fr/current/functions-admin.html#FUNCTIONS-ADMIN-DBOBJECT
[12] https://blog.dalibo.com/2024/02/27/toast.html
[13] https://www.postgresql.fastware.com/blog/what-is-the-new-lz4-toast-compression-in-postgresql-14

1.13 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
- Cela impose par contre une mécanique un peu complexe, dont les parties visibles sont la commande `VACUUM` et le processus d'arrière-plan autovacuum.

1.13.1 Questions

N'hésitez pas, c'est le moment !

1.14 QUIZ

https://dali.bo/m4_quiz

1.15 TRAVAUX PRATIQUES

1.15.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ

But : Découvrir les niveaux d'isolation

Créer une nouvelle base de données nommée **b2**.

Dans la base de données **b2**, créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type `text`.

Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

Ouvrir une transaction.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

Revenir à la première session et lire de nouveau toute la table `t1`.

Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

1.15.2 Niveau d'isolation SERIALIZABLE (Optionnel)

But : Découvrir le niveau d'isolation *serializable*

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

Au sein de trois transactions successives :

- insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**
- chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`

- la sécurité fonctionne-t-elle ?

Pour le client **2**, ouvrir deux transactions en parallèle :

- dans chacune, procéder à retrait de 500 € ;
- dans chacune, appeler `CALL verifie_solde_positif (2)` ;
- dans chacune, valider la transaction ;
- la règle du solde positif est-elle respectée ?

- Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables : (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
- Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :

```
SELECT locktype, mode, pid, granted FROM pg_locks
WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes')
 ↪  ;
```

1.15.3 Effets de MVCC

But : Voir l'effet du MVCC dans les lignes

Créer une nouvelle table `t2` avec deux colonnes : un entier et un texte.

Insérer 5 lignes dans la table `t2`, de `(1, 'un')` à `(5, 'cinq')`.

Lire les données de la table `t2`.

Commencer une transaction. Mettre en majuscules le texte de la troisième ligne de la table `t2`.

Lire les données de la table `t2`. Que faut-il remarquer ?

Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

Afficher `xmin` et `xmax` lors de la lecture des données de la table `t2`, dans chaque session.

Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`, dans chaque session.

Valider la transaction.

Installer l'extension `pageinspect`.

La documentation[a] indique comment décoder le contenu du bloc 0 de la table `t2` :

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

Que faut-il remarquer ?

[a] https://docs.postgresql.fr/current/pageinspect.html

- Lancer `VACUUM` sur `t2`.
- Relancer la requête avec `pageinspect`.
- Comment est réorganisé le bloc ?

Pourquoi l'autovacuum n'a-t-il pas nettoyé encore la table ?

1.15.4 Verrous

But : Trouver des verrous

Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

Récupérer la liste des verrous en attente pour la requête bloquée.

Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

Retrouver les informations sur la session bloquante.

Retrouver cette information avec la fonction `pg_blocking_pids`.

Détruire la session bloquant le `DROP TABLE`.

Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

1.16 TRAVAUX PRATIQUES (SOLUTIONS)

1.16.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ

> Créer une nouvelle base de données nommée **b2**.

```
$ createdb b2
```

> Dans la base de données **b2**, créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type `text`.

```
CREATE TABLE t1 (c1 integer, c2 text);
```

```
CREATE TABLE
```

> Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t1 (c1, c2) VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Ouvrir une transaction.

```
BEGIN;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

```
UPDATE t1 SET c2 = upper(c2) WHERE c1 = 3;
```

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau toute la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

Les modifications réalisées par la deuxième transaction sont immédiatement visibles par la première transaction. C'est le cas des transactions en niveau d'isolation READ COMMITED.

> Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

```
ROLLBACK;
```

```
ROLLBACK;
```

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

> Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

```
UPDATE t1 SET c2 = upper(c2) WHERE c1 = 4;
```

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

En niveau d'isolation `REPEATABLE READ`, la transaction est certaine de ne pas voir les modifications réalisées par d'autres transactions (à partir de la première lecture de la table).

1.16.2 Niveau d'isolation SERIALIZABLE (Optionnel)

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

> Au sein de trois transactions successives :
>
> - insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**
> - chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`
> - la sécurité fonctionne-t-elle ?

Ce client a bien 600 € :

```
SELECT * FROM mouvements_comptes WHERE client = 1 ;
```

```
 client | mouvement
--------+-----------
      1 |       100
      1 |       200
      1 |       300
```

Première transaction :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
CALL verifie_solde_positif (1) ;

NOTICE:  Client 1 - Solde positif : 300
CALL

COMMIT ;
```

Lors d'une seconde transaction : les mêmes ordres renvoient :

```
NOTICE:  Client 1 - Solde positif : 0
```

Avec le troisième débit :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
CALL verifie_solde_positif (1) ;

ERROR:  Client 1 - Solde négatif : -300 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

La transaction est annulée : il est interdit de retirer plus d'argent qu'il n'y en a.

> Pour le client **2**, ouvrir deux transactions en parallèle :
>
> - dans chacune, procéder à retrait de 500 € ;
> - dans chacune, appeler `CALL verifie_solde_positif (2)` ;
> - dans chacune, valider la transaction ;
> - la règle du solde positif est-elle respectée ?

Chaque transaction va donc se dérouler dans une session différente.

Première transaction :

```
BEGIN ; --session 1
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;

NOTICE:  Client 2 - Solde positif : 100
```

On ne commite pas encore.

Dans la deuxième session, il se passe exactement la même chose :

```
BEGIN ; --session 2
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;
```

```
NOTICE:  Client 2 - Solde positif : 100
```

En effet, cette deuxième session ne voit pas encore le débit de la première session.

Les deux tests étant concluants, les deux sessions committent :

```
COMMIT ; --session 1
```

```
COMMIT
```

```
COMMIT ; --session 2
```

```
COMMIT
```

Au final, le solde est négatif, ce qui est pourtant strictement interdit !

```
CALL verifie_solde_positif (2) ;
```

```
ERROR:  Client 2 - Solde négatif : -400 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

Les deux sessions en parallèle sont donc un moyen de contourner la sécurité, qui porte sur le résultat d'un ensemble de lignes, et non juste sur la ligne concernée.

- Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables : (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
- Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :

```
SELECT locktype, mode, pid, granted FROM pg_locks
WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes')
 ↪  ;
```

Première session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;
```

```
NOTICE:  Client 3 - Solde positif : 100
```

On ne committe pas encore.

Deuxième session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;
```

```
NOTICE:  Client 3 - Solde positif : 100
```

Les verrous en place sont les suivants :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;
```

```
 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28304 | t
 relation | RowExclusiveLock | 28304 | t
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

`SIReadLock` est un verrou lié à la sérialisation : noter qu'il porte sur des lignes, portées par les deux sessions. `AccessShareLock` empêche surtout de supprimer la table. `RowExclusiveLock` est un verrou de ligne.

Validation de la première session :

```
COMMIT ;
```

```
COMMIT
```

Dans les verrous, il subsiste toujours les verrous `SIReadLock` de la session de PID 28304, qui a pourtant committé :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;
```

```
 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

Tentative de validation de la seconde session :

```
COMMIT ;
```

```
ERROR:  could not serialize access due to read/write dependencies among transactions
DÉTAIL : Reason code: Canceled on identification as a pivot, during commit attempt.
ASTUCE : The transaction might succeed if retried.
```

La transaction est annulée pour erreur de sérialisation. En effet, le calcul effectué pendant la seconde transaction n'est plus valable puisque la première a modifié les lignes qu'elle a lues.

La transaction annulée doit être rejouée de zéro, et elle tombera alors bien en erreur.

1.16.3 Effets de MVCC

> Créer une nouvelle table `t2` avec deux colonnes : un entier et un texte.

```
CREATE TABLE t2 (i int, t text) ;
```

```
CREATE TABLE
```

> Insérer 5 lignes dans la table `t2`, de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t2(i, t)
VALUES
(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Lire les données de la table `t2`.

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Commencer une transaction. Mettre en majuscules le texte de la troisième ligne de la table `t2`.

```
BEGIN ;
UPDATE t2 SET t = upper(t) WHERE i = 3;
```

```
UPDATE 1
```

> Lire les données de la table `t2`. Que faut-il remarquer ?

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

La ligne mise à jour n'apparaît plus, ce qui est normal. Elle apparaît en fin de table. En effet, quand un `UPDATE` est exécuté, la ligne courante est considérée comme morte et une nouvelle ligne est ajoutée, avec les valeurs modifiées. Comme nous n'avons pas demandé de récupérer les résultats dans un certain ordre (pas d'`ORDER BY`), les lignes sont simplement affichées dans leur ordre de stockage dans les blocs de la table.

> Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

L'ordre des lignes en retour n'est pas le même. Les autres sessions voient toujours l'ancienne version de la ligne 3, puisque la transaction n'a pas encore été validée.

> Afficher `xmin` et `xmax` lors de la lecture des données de la table `t2`, dans chaque session.

Voici ce que renvoie la session qui a fait la modification :

```
SELECT xmin, xmax, * FROM t2;
```

```
 xmin | xmax |  i |   t
------+------+----+--------
  753 |    0 | 1 | un
  753 |    0 | 2 | deux
  753 |    0 | 4 | quatre
  753 |    0 | 5 | cinq
  754 |    0 | 3 | TROIS
```

Et voici ce que renvoie l'autre session :

```
SELECT xmin, xmax, * FROM t2;
```

```
 xmin | xmax |  i |   t
------+------+----+--------
  753 |    0 | 1 | un
  753 |    0 | 2 | deux
  753 |  754 | 3 | trois
  753 |    0 | 4 | quatre
  753 |    0 | 5 | cinq
```

La transaction 754 est celle qui a réalisé la modification. La colonne `xmin` de la nouvelle version de ligne contient ce numéro. De même pour la colonne `xmax` de l'ancienne version de ligne. PostgreSQL se base sur cette information pour savoir si telle transaction peut lire telle ou telle ligne.

> Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`, dans chaque session.

Voici ce que renvoie la session qui a fait la modification :

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

```
 ctid  | xmin | xmax | i |   t
-------+------+------+---+--------
 (0,1) |  753 |    0 | 1 | un
 (0,2) |  753 |    0 | 2 | deux
 (0,4) |  753 |    0 | 4 | quatre
 (0,5) |  753 |    0 | 5 | cinq
 (0,6) |  754 |    0 | 3 | TROIS
```

Et voici ce que renvoie l'autre session :

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

```
 ctid  | xmin | xmax |  i |   t
-------+------+------+----+--------
 (0,1) |  753 |    0 |  1 | un
 (0,2) |  753 |    0 |  2 | deux
 (0,3) |  753 |  754 |  3 | trois
 (0,4) |  753 |    0 |  4 | quatre
 (0,5) |  753 |    0 |  5 | cinq
```

La colonne `ctid` contient une paire d'entiers. Le premier indique le numéro de bloc, le second le numéro de l'enregistrement dans le bloc. Autrement dit, elle précise la position de l'enregistrement sur le fichier de la table.

En récupérant cette colonne, nous voyons que la première session voit la nouvelle position (enregistrement 6 du bloc 0) car elle est dans la transaction 754. Mais pour la deuxième session, cette nouvelle transaction n'est pas validée, donc l'information d'effacement de la ligne 3 n'est pas prise en compte, et on la voit toujours.

> Valider la transaction.

```
COMMIT;
```

```
COMMIT
```

> Installer l'extension `pageinspect`.

```
CREATE EXTENSION pageinspect ;
```

```
CREATE EXTENSION
```

> La documentation[a] indique comment décoder le contenu du bloc 0 de la table `t2` :
>
> ```
> SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
> ```
>
> Que faut-il remarquer ?

[a] https://docs.postgresql.fr/current/pageinspect.html

Cette table est assez petite pour tenir dans le bloc 0 toute entière. `pageinspect` nous fournit le détail de ce qu'il y a dans les lignes (la sortie est coupée en deux pour l'affichage) :

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

```
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid |
----+--------+----------+--------+--------+--------+----------+--------+-
  1 |   8160 |        1 |     31 |    753 |      0 |        0 | (0,1)  |
  2 |   8120 |        1 |     33 |    753 |      0 |        0 | (0,2)  |
  3 |   8080 |        1 |     34 |    753 |    754 |        0 | (0,6)  |
  4 |   8040 |        1 |     35 |    753 |      0 |        0 | (0,4)  |
  5 |   8000 |        1 |     33 |    753 |      0 |        0 | (0,5)  |
  6 |   7960 |        1 |     34 |    754 |      0 |        0 | (0,6)  |
 lp | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |          t_data
----+-------------+------------+--------+--------+-------+--------------------------
  1 |           2 |       2306 |     24 |        |       | \x0100000007756e
  2 |           2 |       2306 |     24 |        |       | \x020000000b64657578
  3 |       16386 |        258 |     24 |        |       | \x030000000d74726f6973
  4 |           2 |       2306 |     24 |        |       | \x040000000f717561747265
  5 |           2 |       2306 |     24 |        |       | \x050000000b63696e71
  6 |       32770 |      10242 |     24 |        |       | \x030000000d54524f4953
```

Tous les champs ne sont pas immédiatement compréhensibles, mais on peut lire facilement ceci :

- Les six lignes sont bien présentes, dont les deux versions de la ligne 3 ;
- Le `t_ctid` de l'ancienne ligne ne contient plus `(0,3)` mais l'adresse de la nouvelle ligne (soit `(0,6)`).

Mais encore :

- Les lignes sont stockées à rebours depuis la fin du bloc : la première a un *offset* de 8160 octets depuis le début, la dernière est à seulement 7960 octets du début ;
- la longueur de la ligne est indiquée par le champ `lp_len` : la ligne 4 est la plus longue ;
- `t_infomask2` est un champ de bits : la valeur 16386 pour l'ancienne version nous indique que le changement a eu lieu en utilisant la technologie HOT (la nouvelle version de la ligne est maintenue dans le même bloc et un chaînage depuis l'ancienne est effectué) ;
- le champ `t_data` contient les valeurs de la ligne : nous devinons `i` au début (01 à 05), et la fin correspond aux chaînes de caractères, précédée d'un octet lié à la taille.

La signification de tous les champs n'est pas dans la documentation mais se trouve dans le code de PostgreSQL[14].

- Lancer `VACUUM` sur `t2`.
- Relancer la requête avec `pageinspect`.
- Comment est réorganisé le bloc ?

```
VACUUM (VERBOSE) t2 ;

INFO:  vacuuming "postgres.public.t2"
…
tuples: 1 removed, 5 remain, 0 are dead but not yet removable
…
VACUUM
```

Une seule ligne a bien été nettoyée. La requête suivante nous montre qu'elle a bien disparu :

[14] https://doxygen.postgresql.org/itemid_8h_source.html

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

```
 lp | lp_off | lp_flags | … | t_ctid |… |          t_data
----+--------+----------+---+--------+--+-------------------------
  1 |   8160 |        1 |   | (0,1)  |  | \x0100000007756e
  2 |   8120 |        1 |   | (0,2)  |  | \x020000000b64657578
  3 |      6 |        2 |   |        |  |
  4 |   8080 |        1 |   | (0,4)  |  | \x040000000f717561747265
  5 |   8040 |        1 |   | (0,5)  |  | \x050000000b63696e71
  6 |   8000 |        1 |   | (0,6)  |  | \x030000000d54524f4953
```

La 3è ligne ici a été remplacée par un simple pointeur sur la nouvelle version de la ligne dans le bloc (mise à jour HOT).

On peut aussi remarquer que les lignes non modifiées ont été réorganisées dans le bloc : là où se trouvait l'ancienne version de la 3è ligne (à 8080 octets du début de bloc) se trouve à présent la 4è. Le `VACUUM` opère ainsi une défragmentation des blocs qu'il nettoie.

> Pourquoi l'autovacuum n'a-t-il pas nettoyé encore la table ?

Le vacuum ne se déclenche qu'à partir d'un certain nombre de lignes modifiées ou effacées (50 + 20% de la table par défaut). On est encore très loin de ce seuil avec cette très petite table.

1.16.4 Verrous

> Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

```
BEGIN;
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | TROIS
  4 | QUATRE
  5 | CINQ
```

> Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

```
DROP TABLE t1;
```

La suppression semble bloquée.

> Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

En tant qu'utilisateur système `postgres` :

```
$ ps -o pid,cmd fx
  PID CMD
 2657 -bash
 2693  \_ psql
 2431 -bash
 2622  \_ psql
 2728      \_ ps -o pid,cmd fx
 2415 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
 2417  \_ postgres: logger
 2419  \_ postgres: checkpointer
 2420  \_ postgres: background writer
 2421  \_ postgres: walwriter
 2422  \_ postgres: autovacuum launcher
 2424  \_ postgres: logical replication launcher
 2718  \_ postgres: postgres b2 [local] DROP TABLE waiting
 2719  \_ postgres: postgres b2 [local] idle in transaction
```

La ligne intéressante est la ligne du `DROP TABLE`. Elle contient le mot clé `waiting`. Ce dernier indique que l'exécution de la requête est en attente d'un verrou sur un objet.

> Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

```
\x
```

```
Expanded display is on.
```

```
SELECT * FROM pg_stat_activity
  WHERE application_name='psql' AND wait_event IS NOT NULL;
```

```
-[ RECORD 1 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2718
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:45.38342+00
xact_start       | 2018-11-02 15:57:32.82511+00
query_start      | 2018-11-02 15:57:32.82511+00
state_change     | 2018-11-02 15:57:32.825112+00
wait_event_type  | Lock
wait_event       | relation
state            | active
backend_xid      | 575
backend_xmin     | 575
query_id         |
query            | drop table t1 ;
backend_type     | client backend
-[ RECORD 2 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

Récupérer la liste des verrous en attente pour la requête bloquée.

```
SELECT * FROM pg_locks WHERE pid = 2718 AND NOT granted;
```

```
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
```

```
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

> Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

```
SELECT relname FROM pg_class WHERE oid=16394;
```

```
-[ RECORD 1 ]
relname | t1
```

Noter que l'objet n'est visible dans `pg_class` que si l'on est dans la même base de données que lui. D'autre part, la colonne `oid` des tables systèmes n'est pas visible par défaut dans les versions antérieures à la 12, il faut demander explicitement son affichage pour la voir.

> Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

```
SELECT * FROM pg_locks WHERE relation = 16394;
```

```
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 4/10
pid                | 2719
mode               | AccessShareLock
granted            | t
fastpath           | f
waitstart          |
-[ RECORD 2 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
```

```
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

Le processus de PID 2718 (le `DROP TABLE`) demande un verrou exclusif sur `t1`, mais ce verrou n'est pas encore accordé (`granted` est à `false`). La session `idle in transaction` a acquis un verrou `Access Share`, normalement peu gênant, qui n'entre en conflit qu'avec les verrous exclusifs.

> Retrouver les informations sur la session bloquante.

On retrouve les informations déjà affichées :

```
SELECT * FROM pg_stat_activity WHERE pid = 2719;
```

```
-[ RECORD 1 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

> Retrouver cette information avec la fonction `pg_blocking_pids`.

Il existe une fonction pratique indiquant quelles sessions bloquent une autre. En l'occurence, notre `DROP TABLE t1` est bloqué par :

```
SELECT pg_blocking_pids(2718);
```

```
-[ RECORD 1 ]----+-------
pg_blocking_pids | {2719}
```

Potentiellement, la session pourrait attendre la levée de plusieurs verrous de différentes sessions.

> Détruire la session bloquant le `DROP TABLE`.

À partir de là, il est possible d'annuler l'exécution de l'ordre bloqué, le `DROP TABLE`, avec la fonction `pg_cancel_backend()`. Si l'on veut détruire le processus bloquant, il faudra plutôt utiliser la fonction `pg_terminate_backend()` :

```
SELECT pg_terminate_backend (2719) ;
```

Dans ce dernier cas, vérifiez que la table a été supprimée, et que la session en statut `idle in transaction` affiche un message indiquant la perte de la connexion.

> Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

```
LOCK TABLE t1;
```

> Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

Le code source de la vue `pg_show_locks` est le suivant :

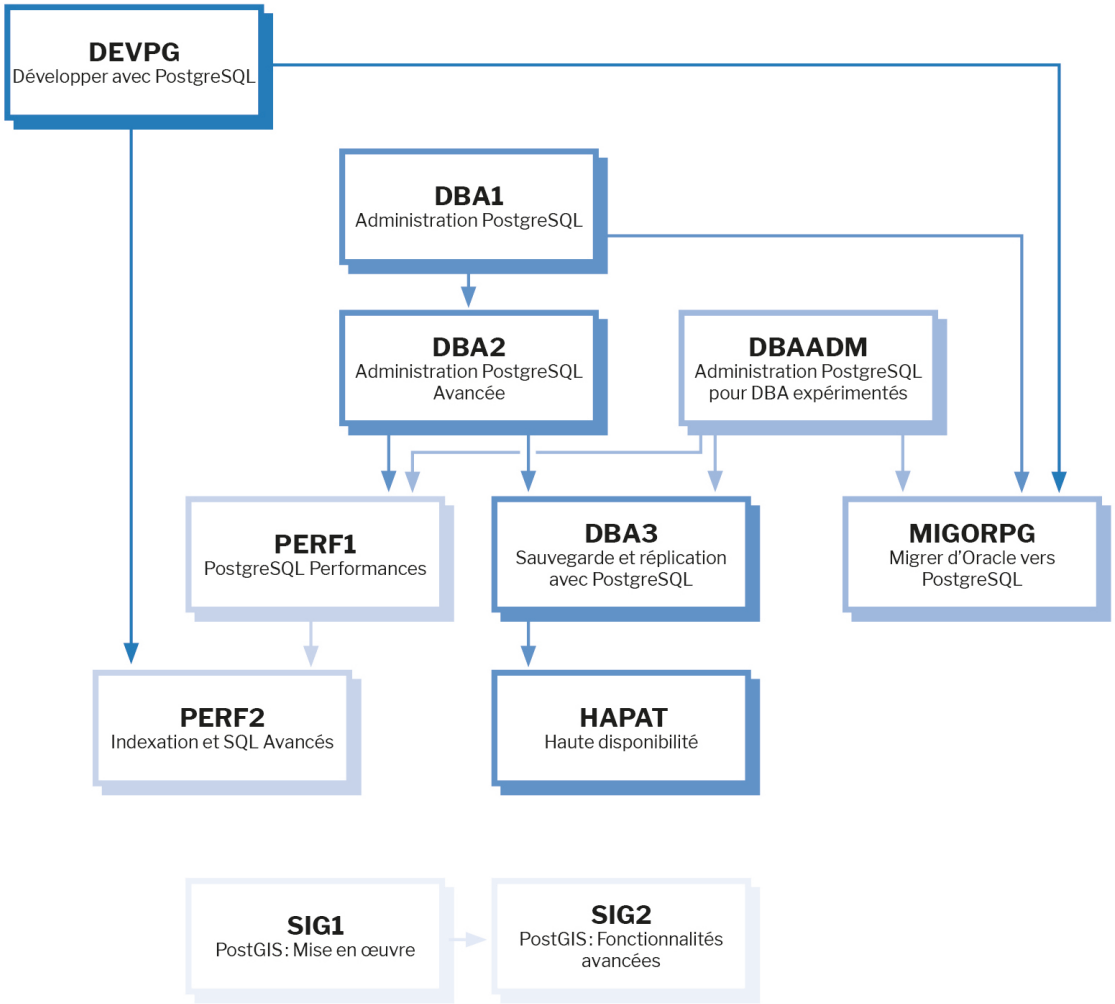
```
CREATE VIEW pg_show_locks as
SELECT
        a.pid,
        usename,
        (now() - query_start) as age,
        c.relname,
        l.mode,
        l.granted
FROM
        pg_stat_activity a
        LEFT OUTER JOIN pg_locks l
                ON (a.pid = l.pid)
        LEFT OUTER JOIN pg_class c
                ON (l.relation = c.oid)
WHERE
        c.relname !~ '^pg_'
ORDER BY
        pid;
```

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.