PGSession 14

Nouveautés de PostgreSQL 14

DALIBO
L'expertise PostgreSQL

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 14

PGSession 14

TITRE : Nouveautés de PostgreSQL 14
SOUS-TITRE : PGSession 14

REVISION: 1

Table des Matières

Nouveautés de PostgreSQL 14

0.1 LA V14

- Développement depuis le 7 juin 2020
- Sortie le 30 septembre 2021
- version 14.1 sortie le 12 novembre 2021

0.2 LES NOUVEAUTÉS

- Administration et maintenance
- Réplication
- Sharding
- Développement et syntaxe SQL
- Supervision
- Performances

0.3 ADMINISTRATION ET MAINTENANCE

- Sécurité
- Nouveautés de configuration (GUC)
- Partitionnement
- Divers

0.3.1 SÉCURITÉ

0.3.1.1 Authentification SCRAM-SHA-256 par défaut

Défaut à présent : `password_encryption` = `scram-sha-256`

- Utilisation conseillée depuis la version 10 !
- Migration :
  - utilisateur par utilisateur
  - `SET password_encryption TO "scram-sha-256" ;`
  - ré-entrer le mot de passe
  - dans `pg_hba.conf` : `md5` → `scram-sha-256`

L'ancienne méthode de chiffrement MD5 utilisée jusque là par défaut est obsolète. Depuis PostgreSQL 10, on pouvait la remplacer par un nouvel algorithme bien plus sûr : SCRAM-SHA-256.

Il s'agit de l'implémentation du *Salted Challenge Response Authentication Mechanism*, basé sur un schéma de type question-réponse, qui empêche le *sniffing* de mot de passe sur les connexions non fiables.

De plus, un même mot de passe entré deux fois sera stocké différemment, alors qu'un chiffrement en MD5 sera le même pour un même nom d'utilisateur, même dans des instances différentes.

Pour plus d'information à ce sujet, vous pouvez consulter cet article de Depesz[1]

Tous les logiciels clients un peu récents devraient être à présent compatibles. Au besoin, vous pourrez revenir à `md5` pour un utilisateur donné. Pour passer d'un système de chiffrement à l'autre, il suffit de passer le paramètre `password_encryption` de `md5` à `scram-sha-256`, globalement ou dans une session, et de ré-entrer le mot de passe des utilisateurs. La valeur dans `postgresql.conf` n'est donc que la valeur par défaut.

Attention : Ce paramètre dépend en partie de l'installation. Vérifiez que `password_encryption` est bien à `scram-sha-256` dans `postgresql.conf` avant de rentrer des mots de passe.

Par exemple :

```
-- A exécuter en tant que postgres

DROP ROLE pierrot ;
DROP ROLE arlequin ;
```

[1] https://www.depesz.com/2017/04/18/waiting-for-postgresql-10-support-scram-sha-256-authentication-rfc-5802-and-7677/

```
CREATE ROLE pierrot LOGIN ;
CREATE ROLE arlequin LOGIN ;

-- Les 2 utilisent le même mot de passe « colombine »

-- pierrot se connecte avec une vieille application
-- qui a besoin d'un mot de passe MD5

SET password_encryption TO md5 ;
\password pierrot

-- arlequin utilise un client récent
SET password_encryption TO "scram-sha-256" ;

\password arlequin

SELECT rolname, rolpassword
FROM   pg_authid
WHERE  rolname IN ('pierrot', 'arlequin') \gx
```

```
-[ RECORD 1 ]----------------------------------------------------------------------
rolname     | pierrot
rolpassword | md59c20f03b508f8120b2294a8fedd42557
-[ RECORD 2 ]----------------------------------------------------------------------
rolname     | arlequin
rolpassword | SCRAM-SHA-256$4096:tEblPJ9ZoVPEkE/AOyreag==$cb/g6sak7SDEL6gCxRd9GUH …
```

Le type de mot de passe est visible au début de `rolpassword`.

Noter que si Pierrot utilise le même mot de passe sur une autre instance PostgreSQL avec le chiffrement MD5, on retrouvera `md59c20f03b508f8120b2294a8fedd42557`. Cela ouvre la porte à certaines attaques par force brute, et peut donner la preuve que le mot de passe est identique sur différentes installations.

Dans `pg_hba.conf`, pour se connecter, ils auront besoin de ces deux lignes :

```
host    all             pierrot          192.168.88.0/24          md5
host    all             arlequin         192.168.88.0/24          scram-sha-256
```

(Ne pas oublier de recharger la configuration.)

Puis Pierrot met à jour son application. Son administrateur ré-entre alors le même mot de passe avec SCRAM-SHA-256 :

```
-- A exécuter en tant que postgres

SET password_encryption TO "scram-sha-256" ;
```

```
\password pierrot

SELECT rolname, rolpassword
FROM   pg_authid
WHERE  rolname IN ('pierrot', 'arlequin') \gx

-[ RECORD 1 ]----------------------------------------------------------------------
rolname     | arlequin
rolpassword | SCRAM-SHA-256$4096:tEblPJ9ZoVPEkE/AOyreag==$cb/g6sak7SDEL6gCxRd9GUH …
-[ RECORD 2 ]----------------------------------------------------------------------
rolname     | pierrot
rolpassword | SCRAM-SHA-256$4096:fzKspWtDmyFKy3j+ByXvhg==$LfM08hhV3BYgqubxZJ1Vkfh …
```

Pierrot peut se reconnecter tout de suite sans modifier `pg_hba.conf` : en effet, une entrée `md5` autorise une connexion par SCRAM-SHA-256 (l'inverse n'est pas possible).

Par sécurité, après validation de l'accès, il vaut mieux ne plus accepter que SCRAM-SHA-256 dans `pg_hba.conf` :

```
host    all             pierrot         192.168.88.0/24         scram-sha-256
host    all             arlequin        192.168.88.0/24         scram-sha-256
```

0.3.1.2 Nouveaux rôles prédéfinis

- `pg_read_all_data`
- `pg_write_all_data`
- `pg_database_owner` (*template*)

Les rôles `pg_read_all_data`, `pg_write_all_data` et `pg_database_owner` viennent compléter la liste des rôles proposés par PostgreSQL. Les deux premiers de ces rôles permettent d'éviter d'avoir à appliquer des droits de lecture ou d'écriture sur des nouvelles tables à des utilisateurs nominatifs après un déploiement.

- `pg_read_all_data`

Le rôle `pg_read_all_data` permet de donner un droit de lecture sur toutes les tables de tous les schémas et de toutes les bases de données de l'instance PostgreSQL à un rôle spécifique. Ce type de droit est utile lorsque la politique de sécurité mise en place autour de vos instances PostgreSQL implique la création d'un utilisateur spécifique pour la sauvegarde via l'outil `pg_dump`.

Dans l'exemple ci-dessous, seul un utilisateur *superadmin* ou disposant de l'option *admin* sur le rôle `pg_read_all_data` peut octroyer ce nouveau rôle.

```
GRANT pg_read_all_data TO dump_user;
```

Par le passé, une série de commandes était nécessaire pour donner les droits de lecture à un rôle spécifique sur les tables existantes et à venir d'un schéma au sein d'une base de données.

```
GRANT USAGE ON SCHEMA public TO dump_user;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO dump_user;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO dump_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO dump_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON SEQUENCES TO dump_user;
```

Cependant, dès qu'un nouveau schéma était créé dans la base, l'export par `pg_dump` échouait avec le message `ERROR:  permission denied for schema <name>`. Il fallait alors réaffecter les droits précédents sur le nouveau schéma pour corriger le problème.

- `pg_write_all_data`

Le rôle `pg_write_all_data` permet de donner un droit d'écriture sur toutes les tables de tous les schémas de l'instance PostgreSQL à un rôle spécifique. Ce rôle peut être utile lors de traitement d'import de type ETL, où les données existantes ne doivent pas être lues pour des raisons de sécurité.

- `pg_database_owner`

Le rôle `pg_database_owner`, contrairement à `pg_read_all_data` et `pg_write_all_data`, n'a pas de droits par défaut. Il représente le propriétaire d'une base de données, afin de faciliter l'application de droits d'une base de données *template*, prête à être déployée. À la création d'une nouvelle base à partir de ce *template*, les droits qui lui ont été donnés s'appliqueront au propriétaire de cette base de données.

Le rôle `pg_database_owner` ne peut pas être octroyé directement à un autre rôle, comme le montre le message ci-dessous. PostgreSQL considère qu'il ne peut y avoir qu'un seul propriétaire par base de données.

```
GRANT pg_database_owner TO atelier;
-- ERROR:  role "pg_database_owner" cannot have explicit members
```

Lorsqu'un changement de propriétaire survient dans la base, les droits sur les objets appartenant au rôle `pg_database_owner` sont alors transmis à ce nouveau rôle. Le précédent propriétaire n'aura plus accès au contenu des tables ou des vues.

```
CREATE TABLE tab (id int);
ALTER TABLE tab OWNER TO pg_database_owner;
```

```
-- avec un compte superutilisateur
ALTER DATABASE test OWNER TO role1;

SET role = role1;
INSERT INTO tab VALUES (1), (2), (3);
-- INSERT 0 3

-- avec un compte superutilisateur
ALTER DATABASE test OWNER TO role2;

SET role = role1;
INSERT INTO tab VALUES (4), (5), (6);
-- ERROR:  permission denied for table tab
```

Pour conclure, les rôles `pg_write_all_data`, `pg_read_all_data` et `pg_database_owner` peuvent se voir donner des droits sur d'autres objets de la base de données au même titre que tout autre rôle.

0.3.2 NOUVEAUTÉS DE CONFIGURATION (GUC)

0.3.2.1 Temps d'attente maximal pour une session inactive

- Nouveau paramètre `idle_session_timeout`
- Temps d'attente avant d'interrompre une session inactive
  - Désactivé par défaut (valeur `0`)
  - Comportement voisin de `idle_in_transaction_session_timeout`
  - Paramètre de session, ou globalement pour l'instance

Le paramètre `idle_session_timeout` définit la durée maximale sans activité entre deux requêtes lorsque l'utilisateur n'est pas dans une transaction. Son comportement est similaire à celui du paramètre `idle_in_transaction_session_timeout` introduit dans PostgreSQL 9.6, qui ne concerne que les sessions en statut `idle in transaction`.

Ce paramètre a pour conséquence d'interrompre toute session inactive depuis plus longtemps que la durée indiquée par ce paramètre. Cela permet de limiter la consommation de ressources des sessions inactives (mémoire notamment) et de diminuer le coût de maintenance des sessions connectées à l'instance en limitant leur nombre.

Si cette valeur est indiquée sans unité, elle est comprise comme un nombre en millisecondes. La valeur par défaut de `0` désactive cette fonctionnalité. Le changement de la valeur du paramètre `idle_session_timeout` ne requiert pas de démarrage ou de droit particulier.

```
SET idle_session_timeout TO '5s';
-- Attendre 5 secondes.

SELECT 1;

FATAL:  terminating connection due to idle-session timeout
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
```

Un message apparaît dans les journaux d'activité :

```
FATAL:  terminating connection due to idle-session timeout
```

0.3.2.2 Modification à chaud de la restore_command

- Le paramètre `restore_command` ne nécessite plus de redémarrage
- Applicable pour les instances secondaires

La modification du paramètres `restore_command` ne nécessite plus de redémarrage pour que l'instance secondaire prenne en compte sa nouvelle valeur. Un simple rechargement suffit.

Cette amélioration permet de ne plus redémarrer un réplica lorsque la provenance des archives de journaux de transaction est modifiée. Les sessions en cours sont donc maintenues sans risque lors de la manipulation.

0.3.2.3 Détection des déconnexions pendant l'exécution d'une requête

- Nouveau paramètre `client_connection_check_interval`
- Détermine le délai entre deux contrôles de connexion
  - Désactivé par défaut (valeur `0`)
  - Utile pour les très longues requêtes
  - Repose sur des appels système non standards (non définis par POSIX)
    - donc Linux uniquement

Le paramètre `client_connection_check_interval` indique le délai avant de contrôler la connexion avec le client distant. En l'absence de ces contrôles intermédiaires, le serveur ne détecte la perte de connexion que lorsqu'il interagit avec le *socket* de la session (attente, envoyer ou recevoir des données).

Sans unité, il s'agit d'une durée exprimée en milliseconde. La valeur par défaut est de `0`, ce qui désactive ce comportement. Si le client ne répond pas lors de l'exécution d'une requête (très) longue, l'instance peut à présent interrompre la requête afin de ne pas consommer inutilement les ressources du serveur.

Actuellement, le comportement du paramètre `client_connection_check_interval` repose sur une extension non standard du système d'appel au kernel. Cela implique que seuls les systèmes d'exploitation basés sur Linux peuvent en bénéficier. Dans un avenir hypothétique, les développeurs pourront réécrire le code pour reposer sur un nouveau système de *heartbeat* ou équivalent pour supporter plus de systèmes.

0.3.3 PARTIONNEMENT

0.3.3.1 Nouveautés sur REINDEX et reindexdb

- `REINDEX` est maintenant disponible pour les tables et index partitionnés
- Supporte la clause `CONCURRENTLY`
- Fonctionne en mode multi-transactions

Jusqu'à la version 13, la commande `REINDEX` ne pouvait pas être utilisée sur les tables et index partionnés. Il fallait réindexer les partitions une par une.

```
REINDEX INDEX parent_index;
-- ERROR:  REINDEX is not yet implemented for partitioned indexes

REINDEX TABLE parent;
-- WARNING:  REINDEX of partitioned tables is not yet implemented, skipping "parent"
-- REINDEX
```

Avec la version 14, il est maintenant possible de passer une table ou un index partitionné comme argument aux commandes `REINDEX INDEX` ou `REINDEX TABLE`. L'ensemble des partitions sont parcourues afin de réindexer tous les éléments concernés. Seuls ceux disposant d'un stockage physique sont visés (on écarte donc les tables et index parents).

Prenons la table partitionnée `parent` et son index `parent_index`. Il est possible de déterminer la fragmentation de l'index à l'aide de l'extension `pgstattuple` :

```
CREATE EXTENSION pgstattuple;
SELECT avg_leaf_density, leaf_fragmentation FROM pgstatindex('enfant_1_id_idx');

 avg_leaf_density | leaf_fragmentation
------------------+--------------------
            74.18 |                 50

SELECT avg_leaf_density, leaf_fragmentation FROM pgstatindex('enfant_2_id_idx');

 avg_leaf_density | leaf_fragmentation
------------------+--------------------
            74.17 |                 50
```

Tous les index peuvent être reconstruits avec une unique commande :

```
REINDEX INDEX parent_index;

SELECT avg_leaf_density, leaf_fragmentation FROM pgstatindex('enfant_1_id_idx');

 avg_leaf_density | leaf_fragmentation
------------------+--------------------
            90.23 |                  0
```

```
SELECT avg_leaf_density, leaf_fragmentation FROM pgstatindex('enfant_2_id_idx');
```

```
 avg_leaf_density | leaf_fragmentation
------------------+--------------------
            90.23 |                  0
```

Côté fonctionnement, celui-ci est *multi transactions*. C'est-à-dire que chaque partition est traitée séquentiellement dans une transaction spécifique. Cela a pour avantage de minimiser le nombre d'index invalides en cas d'annulation ou d'échec avec la commande `REINDEX CONCURRENTLY`. Cependant, cela empêche son fonctionnement dans un bloc de transaction.

```
BEGIN;
REINDEX INDEX parent_index;
-- ERROR:  REINDEX INDEX cannot run inside a transaction block
-- CONTEXT: while reindexing partitioned index "public.parent_index"
```

La vue `pg_stat_progress_create_index` peut être utilisée pour suivre la réindexation, mais les colonnes `partitions_total` et `partitions_done` resteront à 0 durant la durée de l'opération. Il est néanmoins possible de voir les `REINDEX` passer les uns après les autres dans cette vue.

0.3.4 DIVERS

0.3.4.1 Compression des toast configurable en LZ4 et PGLZ

- Il est maintenant possible de compresser les données `TOAST` avec `LZ4`
- Plusieurs niveaux de définition (global ou par colonne)
- LZ4 est sensiblement plus rapide
- PGLZ a un meilleur taux de compression
- Nouvelle option `--no-toast-compression` pour `pg_dump`

0.4 RÉPLICATION

- Réplication physique
- Réplication logique

0.4.1 RÉPLICATION PHYSIQUE

0.4.1.1 *Amélioration de `pg_rewind`*

- La source d'un rewind peut être une instance secondaire
- Permet de limiter l'impact des lectures sur la nouvelle instance primaire

---

0.4.1.2 *Nouveautés de libpq pour simplifier HA et répartition de charge*

- Nouvelles options pour le paramètre `target_session_attrs`
  - `read_only`
  - `primary`
  - `standby`
  - `prefer-standby`

0.4.2 RÉPLICATION LOGIQUE

- Nouveau mode *streaming in-progress* pour la réplication logique
  - à activer
- Informations supplémentaires pour les messages d'erreur de type `columns are missing`
- Ajout de la syntaxe `ALTER SUBSCRIPTION… ADD/DROP PUBLICATION…`

Streaming in-progress

Lorsque l'on utilise la réplication logique, le processus *walsender* va procéder au décodage logique et réordonner les modifications depuis les fichiers WAL avant de les envoyer à l'abonné. Cette opération est faite en mémoire mais en cas de dépassement du seuil indiqué par le paramètre `logical_decoding_work_mem`, ces données sont écrites sur disque.

Ce comportement à deux inconvénients :

- il peut provoquer l'apparition d'une volumétrie non négligeable dans le répertoire `pg_replslot` et jouer sur les I/O ;
- il n'envoie les données à l'abonné qu'au `COMMIT` de la transaction, ce qui peut engendrer un fort retard dans la réplication. Dans le cas de grosses transactions, le réseau et l'abonné peuvent également être mis à rude épreuve car toutes les données seront envoyées en même temps.

Avec cette nouvelle version, il est maintenant possible d'avoir un comportement différent. Lorsque la mémoire utilisée pour décoder les changements depuis les WAL atteint le seuil de `logical_decoding_work_mem`, plutôt que d'écrire les données sur disque, la transaction consommant le plus de mémoire de décodage va être sélectionnée et diffusée en continu et ce même si elle n'a pas encore reçu de `COMMIT`.

Il va donc être possible de réduire la consommation I/O et également la latence entre le publieur et l'abonné.

Ce nouveau comportement n'est pas activé par défaut ; il faut ajouter l'option `streaming = on` à l'abonné :

```
CREATE SUBSCRIPTION sub_stream
  CONNECTION 'connection string'
  PUBLICATION pub WITH (streaming = on);

ALTER SUBSCRIPTION sub_stream SET (streaming = on);
```

Certains cas nécessiteront toujours des écritures sur disque. Par exemple dans le cas où le seuil mémoire de décodage est atteint, mais qu'un tuple n'est pas complètement décodé.

Messages d'erreur plus précis

Le message d'erreur affiché dans les traces lorsqu'il manque certaines colonnes à une table présente sur un abonné, a été amélioré. Il indique maintenant la liste des colonnes manquantes et non plus simplement le message `is missing some replicated columns`.

```
-- En version 13
ERROR:  logical replication target relation "public.t"
        is missing some replicated columns

-- En version 14
ERROR:  logical replication target relation "public.t"
        is missing replicated column: "champ"
```

ALTER SUBSCRIPTION... ADD/DROP PUBLICATION...

Jusqu'alors, dans le cas d'une mise à jour de publication dans une souscription, il était nécessaire d'utiliser la commande `ALTER SUBSCRIPTION… SET PUBLICATION…` et de connaître la liste des publications sous peine d'en perdre.

Avec la version 14, il est désormais possible d'utiliser la syntaxe `ALTER SUBSCRIPTION… ADD/DROP PUBLICATION…` pour manipuler plus facilement les publications.

```
-- on dispose d'une souscription avec 2 publications
\dRs
          Liste des souscriptions
 Nom | Propriétaire | Activé | Publication
-----+--------------+--------+-------------
 sub | postgres     | t      | {pub,pub2}

-- en version 13 et inférieures, pour ajouter une nouvelle publication, il était
-- nécessaire de connaître les autres publications pour actualiser la souscription
ws14=# ALTER SUBSCRIPTION sub SET PUBLICATION pub,pub2,pub3;

-- en version 14, les clauses ADD et DROP simplifient ces modifications
ws14=# ALTER SUBSCRIPTION sub ADD PUBLICATION pub3;
```

0.5 SHARDING

- *Foreign Data Wrapper* et Sharding

0.5.1 FOREIGN DATA WRAPPER ET SHARDING

- Évolutions pour les *Foreign Data Wrapper*
- Vers une architecture distribuée (*sharding*)

0.5.1.1 Lecture asynchrone des tables distantes

- Nouveau nœud d'exécution `Async Foreign Scan`
- `CREATE SERVER … OPTIONS (host …, port …, async_capable on)` (pas par défaut !)
- Lecture parallélisée pour les partitions distantes

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append
   ->  Async Foreign Scan on public.async_p1 t1_1
         Output: t1_1.a, t1_1.b, t1_1.c
         Remote SQL: SELECT a, b, c FROM public.base_tbl1 WHERE (((b % 100) = 0))
   ->  Async Foreign Scan on public.async_p2 t1_2
         Output: t1_2.a, t1_2.b, t1_2.c
         Remote SQL: SELECT a, b, c FROM public.base_tbl2 WHERE (((b % 100) = 0))
```

Les tables distantes fournies par l'extension `postgres_fdw` bénéficient du nouveau nœud d'exécution `Async Foreign Scan` lorsqu'elles proviennent de plusieurs serveurs distincts. Il s'agit d'une évolution du nœud existant `Foreign Scan` pour favoriser la lecture parallélisée de plusieurs tables distantes, notamment au sein d'une table partitionnée.

L'option `async_capable` doit être activée au niveau de l'objet serveur ou de la table distante, selon la granularité voulue. L'option n'est pas active par défaut.

Les tables parcourues en asynchrone apparaissent dans un nouveau nœud `Async` :

```
EXPLAIN (verbose, costs off) SELECT * FROM t1 WHERE b % 100 = 0;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append
   ->  Async Foreign Scan on public.async_p1 t1_1
         Output: t1_1.a, t1_1.b, t1_1.c
         Remote SQL: SELECT a, b, c FROM public.base_tbl1 WHERE (((b % 100) = 0))
```

```
->  Async Foreign Scan on public.async_p2 t1_2
      Output: t1_2.a, t1_2.b, t1_2.c
      Remote SQL: SELECT a, b, c FROM public.base_tbl2 WHERE (((b % 100) = 0))
```

L'intérêt est évidemment de faire fonctionner simultanément plusieurs serveurs distants, ce qui peut amener de gros gains de performance. C'est un grand pas dans l'intégration d'un *sharding* natif dans PostgreSQL.

En ce qui concerne la syntaxe, les ordres d'activation et de désactivation de l'option, sur le serveur ou la table sont par exemple :

```
CREATE SERVER distant3
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'machine3', dbname 'bi', port 5432, async_capable 'on') ;

ALTER SERVER distant1 OPTIONS (ADD async_capable 'on');

CREATE FOREIGN TABLE donnees1
PARTITION OF …
OPTIONS (async_capable 'on') ;

ALTER FOREIGN TABLE donnees1  OPTIONS (DROP async_capable);
```

0.6 DÉVELOPPEMENT ET SYNTAXE SQL

- Manipulation du type JSONB
- Nouvelle fonction `date_bin`

0.6.1 MANIPULATION DU TYPE JSONB

- Nouvelle syntaxe d'accès aux éléments d'une colonne `jsonb`
- Expressions avec indice, de style tableau

```
SELECT ('{"a": 1}'::jsonb)['a'];
SELECT * FROM table_name WHERE jsonb_field['key'] = '"value"';
UPDATE table_name SET jsonb_field['key'] = '1';
```

Cette nouvelle version de PostgreSQL apporte une nouvelle syntaxe pour extraire ou modifier les éléments d'une colonne `jsonb`. À l'instar des opérateurs `->` et `->>`, il est à présent possible de manipuler les éléments à la manière d'un tableau avec l'indiçage (*subscripting*).

Les deux requêtes suivantes sont similaires :

```
SELECT id, product->'name' AS product, product->'price' AS price
  FROM products WHERE product->>'brand' = 'AniOne';

SELECT id, product['name'] AS product, product['price'] AS price
  FROM products WHERE product['brand'] = '"AniOne"';
```

```
 id  |            product             | price
-----+--------------------------------+-------
 100 | "Arbre à chat tonneau Aurelio" | 189
 101 | "Griffoir tonneau Tobi"        | 169
```

Cependant, l'opérateur `->>` permettant d'extraire la valeur d'un élément textuel n'a pas d'équivalent et il est nécessaire d'ajouter les guillemets pour réaliser des comparaisons, par exemple.

L'extraction de valeurs imbriquées est également possible avec cette syntaxe. La mise à jour d'un élément est aussi supportée comme le montre l'exemple suivant :

```
UPDATE products SET product['dimension']['L'] = '50' WHERE id = 100;
```

0.6.2 NOUVELLE FONCTION DATE_BIN

- Nouvelle fonction pour répartir des timestamps dans des intervalles (buckets)

```
SELECT date_bin('1 hour 30 minutes', t, '2021-06-01 00:00:00'::timestamptz),
       id_sonde, avg(mesure)
  FROM sonde GROUP BY 1, 2 ORDER BY 1 ASC;
       date_bin        | id_sonde |          avg
-----------------------+----------+-----------------------
 2021-06-01 00:00:00+02 |        1 |     2.9318518518518519
 2021-06-01 01:30:00+02 |        1 |     8.6712962962962963
 2021-06-01 03:00:00+02 |        1 |    14.1218518518518519
 2021-06-01 04:30:00+02 |        1 |    19.0009259259259259
```

0.7 SUPERVISION

- Nouvelle vue `pg_stat_wal`
- Nouveautées dans `pg_stat_statements`
- Ajout de statistiques sur les sessions dans `pg_stat_database`
- Identifiant pour les requêtes normalisées
- Nouveauté dans `pg_locks`

0.7.1 NOUVELLE VUE `PG_STAT_WAL`

- Permet de surveiller l'activité des WAL
- Nouveau paramètre : `track_wal_io_timing`

La nouvelle vue système `pg_stat_wal` permet d'obtenir des statistiques sur l'activité des WAL. Elle est composée des champs suivants :

- `wal_records` : Nombre total d'enregistrement WAL
- `wal_fpi` : Nombre total d'enregistrement *full page images*, ces écritures de page complètes sont déclenchées lors de la première modification d'une page après un `CHECKPOINT` si le paramètre `full_page_writes` est configuré à `on` ;
- `wal_bytes` : Quantité totale de WAL générés en octets ;
- `wal_buffers_full` : Nombre de fois où des enregistrements WAL ont été écrits sur disque car les *WAL buffers* était pleins ;
- `wal_write` : Nombre de fois ou les données du *WAL buffers* ont été écrit sur disque via une requête `XLogWrite` ;
- `wal_sync` : Nombre de fois ou les données du *WAL buffers* ont été synchronisées sur disque via une requête `issue_xlog_fsync` ;
- `wal_write_time` : Temps total passé à écrire les données du *WAL buffers* sur disque via une requête `XLogWrite` ;
- `wal_sync_time` : Temps total passé à synchroniser les données du *WAL buffers* sur disque via une requête `issue_xlog_fsync` ;
- `stats_reset` : Date de la dernière remise à zéro des statistiques.

Les statistiques de cette vue peuvent être remises à zéro grâce à l'appel de la fonction `pg_stat_reset_shared()` avec le paramètre `wal`.

Cette vue est couplée à un nouveau paramètre : `track_wal_io_timing`. Il permet d'activer ou non le chronométrage des appels d'entrées/sortie pour les WAL. Par défaut celui-ci est à `off`. Comme pour le paramètre `track_io_timing`, l'activation de ce nouveau paramètre peut entraîner une surcharge importante en raison d'appels répétés au système d'exploitation. Une mesure de ce surcoût pourra être réalisée avec l'outil

pg_test_timing. Seul un super utilisateur peut modifier ce paramètre.

L'activation de track_wal_io_timing est nécessaire afin d'obtenir des données pour les colonnes wal_write_time et wal_sync_time de la vue pg_stat_wal.

Ces nouvelles statistiques vont permettre d'avoir de nouvelles métriques pour la métrologie et la supervision. Elles permettront également d'ajuster la taille de paramètres comme wal_buffers (grâce à wal_buffers_full) ou d'évaluer l'impact de checkpoint trop fréquents sur le système (wal_fpi & wal_records).

0.7.2 NOUVEAUTÉES DANS PG_STAT_STATEMENTS

- Traçage des accès faits via CREATE TABLE AS, SELECT INTO, CREATE MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW et FETCH
- Nouvelle vue pg_stat_statements_info
- Nouvelle colonne toplevel dans la vue pg_stat_statements

Statistiques plus complètes

pg_stat_statements est désormais capable de comptabiliser les lignes lues ou affectées par les commandes CREATE TABLE AS, SELECT INTO, CREATE MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW et FETCH.

Le script SQL suivant permet d'illustrer cette nouvelle fonctionnalité. Il effectue plusieurs de ces opérations après avoir réinitialisé les statistiques de pg_stat_statements.

```
SELECT pg_stat_statements_reset();

CREATE TABLE pg_class_1 AS SELECT * FROM pg_class;
SELECT * INTO pg_class_2 FROM pg_class;
CREATE MATERIALIZED VIEW pg_class_3 AS SELECT * FROM pg_class;
REFRESH MATERIALIZED VIEW pg_class_3;
```

On retrouve bien le nombre de lignes affectées par les requêtes, dans le champ rows de la vue pg_stat_statements.

```
SELECT query, rows FROM pg_stat_statements;
```

```
                              query                               | rows
------------------------------------------------------------------+------
 select * into pg_class_2 FROM pg_class                           |  401
 select pg_stat_statements_reset()                                |    1
 refresh materialized view pg_class_3                             |  410
 create materialized view pg_class_3 as select * from pg_class    |  404
 create table pg_class_1 as select * from pg_class                |  398
```

Le même scénario de test réalisé en version 13 ne donne pas ces informations.

```
SELECT query, rows FROM pg_stat_statements;
```

```
                              query                               | rows
------------------------------------------------------------------+------
 select * into pg_class_2 FROM pg_class                           |    0
 refresh materialized view pg_class_3                             |    0
 select pg_stat_statements_reset()                                |    1
 create table pg_class_1 as select * from pg_class                |    0
 create materialized view pg_class_3 as select * from pg_class    |    0
```

La vue pg_stat_statements_info

Une nouvelle vue pg_stat_statements_info est ajoutée pour tracer les statistiques du module lui-même.

```
\d pg_stat_statements_info;
```

```
                View "public.pg_stat_statements_info"
   Column    |           Type           | Collation | Nullable | Default
-------------+--------------------------+-----------+----------+---------
 dealloc     | bigint                   |           |          |
 stats_reset | timestamp with time zone |           |          |
```

La colonne stats_reset rapporte la date de la dernière réinitialisation des statistiques par la fonction pg_stat_statements_reset().

La colonne dealloc décompte les événements de purge qui sont déclenchés lorsque le nombre de requêtes distinctes dépasse le seuil défini par le paramètre pg_stat_statements.max. Elle sera particulièrement utile pour configurer ce paramètre. En effet, si pg_stat_statements.max est trop bas, des purges trop fréquentes peuvent avoir un impact négatif sur les performances.

Sur une instance en version 14 avec pg_stat_statement.max configuré à une valeur basse de 100, des requêtes distinctes sont exécutées via un script après une réinitialisation des statistiques de pg_stat_statements, afin de provoquer un dépassement volontaire du seuil :

```
psql -d ws14 -c "select pg_stat_statements_reset();"

for rel_id in {0..200} ; do
    psql -d ws14 -c "create table pg_rel_${rel_id} (id int)";
    psql -d ws14 -c "drop table pg_rel_${rel_id}";
done
```

La vue pg_stat_statements a bien conservé un nombre de requêtes inférieur à pg_stat_statement.max, bien que 400 requêtes distinctes aient été exécutées :

```
SELECT count(*) FROM pg_stat_statements;
```

```
 count
-------
    93
```

Le nombre d'exécution de la purge de `pg_stat_statements` est désormais tracé dans la vue `pg_stat_statements_info`. Elle a été déclenchée 31 fois pendant les créations et suppressions de tables :

```
SELECT * FROM pg_stat_statements_info;

 dealloc |          stats_reset
---------+------------------------------
      31 | 2021-09-02 13:30:26.497678+02
```

Ces informations peuvent également être obtenues via la fonction du même nom :

```
SELECT pg_stat_statements_info();
       pg_stat_statements_info

--------------------------------------
 (31,"2021-09-02 13:35:22.457383+02")
```

La nouvelle colonne `toplevel`

Une nouvelle colonne `toplevel` apparaît dans la vue `pg_stat_statements`. Elle est de type booléen et précise si la requête est directement exécutée ou bien exécutée au sein d'une fonction. Le traçage des exécutions dans les fonctions n'est possible que si le paramètre `pg_stat_statements.track` est à `all`.

Sur une instance en version 14 avec `pg_stat_statement.track` configuré à `all`, une fonction simple contenant une seule requête SQL est créée. Elle permet de retrouver le nom d'une relation à partir de son `oid`.

```
CREATE OR REPLACE FUNCTION f_rel_name(oid int) RETURNS varchar(32) AS
$$
    SELECT relname FROM pg_class WHERE oid=$1;
$$
LANGUAGE SQL;
```

Après avoir réinitialisé les statistiques de `pg_stat_statements`, le nom d'une table est récupérée depuis son `oid` en utilisant une requête SQL directement, puis via la fonction `f_rel_name` :

```
SELECT pg_stat_statements_reset();
SELECT relname FROM pg_class WHERE oid=26140 ;
SELECT f_rel_name(26140);
```

La vue `pg_stat_statements` est consultée directement après :

```
SELECT query, toplevel FROM pg_stat_statements
 WHERE query NOT LIKE '%pg_stat_statements%'
 ORDER BY query;

                  query                   | toplevel
------------------------------------------+----------
 select f_rel_name($1)                    | t
 select relname from pg_class where oid=$1 | f
 select relname from pg_class where oid=$1 | t
```

On retrouve bien l'appel de la fonction, ainsi que les deux exécutions de la requête sur `pg_class`, celle faite directement, et celle faite au sein de la fonction `f_rel_name`. La requête dont `toplevel` vaut `false` correspond à l'exécution dans la fonction. Il n'était pas possible dans une version antérieure de distinguer aussi nettement les deux contextes d'exécution.

0.7.3 AJOUT DE STATISTIQUES SUR LES SESSIONS DANS `PG_STAT_DATABASE`

- Ajout des colonnes suivantes à la vue système `pg_stat_database` :
  - `session_time`
  - `active_time`
  - `idle_in_transaction_time`
  - `sessions`
  - `sessions_abandoned`
  - `sessions_fatal`
  - `sessions_killed`

La vue `pg_stat_database` dispose à présent de nouveaux compteurs orientés sessions et temps de session :

- `session_time` : temps passé par les sessions sur cette base de données. Ce compteur n'est mis à jour que lorsque l'état d'une session change ;
- `active_time` : temps passé à exécuter des requêtes SQL sur cette base de données. Correspond aux états `active` et `fastpath function call` dans `pg_stat_activity` ;
- `idle_in_transaction_time` : temps passé à l'état `idle` au sein d'une transaction sur cette base de données. Correspond aux états `idle in transaction` et `idle in transaction (aborted)` dans `pg_stat_activity`. Rappelons que cet état, s'il est prolongé, gêne l'autovacuum ;
- `sessions` : nombre total de sessions ayant établi une connexion à cette base de données ;

- `sessions_abandoned` : nombre de sessions interrompues à cause d'une perte de connexion avec le client ;
- `sessions_fatal` : nombre de sessions interrompues par des erreurs fatales ;
- `sessions_killed` : nombre de sessions interrompues par des demandes administrateur.

Ces nouvelles statistiques d'activité permettront d'avoir un aperçu de l'activité des sessions sur une base de données. C'est un réel plus lors de la réalisation d'un audit car elles permettront de repérer facilement des problèmes de connexion (`sessions_abandoned`), d'éventuels passages de l'*OOM killer* (`sessions_fatal`) ou des problèmes de stabilité (`sessions_fatal`). Cela permettra également d'évaluer plus facilement la pertinence de la mise en place d'un pooler de connexion (`*_time`).

La présence de ces métriques dans l'instance simplifiera également leur obtention pour les outils de supervision et métrologie. En effet, certaines n'étaient accessibles que par analyse des traces (`session time`, `sessions`) ou tout simplement impossibles à obtenir.

0.7.4 IDENTIFIANT POUR LES REQUÊTES NORMALISÉES

- Le *query id* est disponible globalement
  - valeur hachée sur 64 bits d'une requête normalisée
  - introduit avec `pg_stat_statements` en version 9.4
  - `pg_stat_activity`, `log_line_prefix`, `EXPLAIN VERBOSE`
- nouveau paramètre `compute_query_id` (`auto` par défaut)

```
       query_id       |                            query
----------------------+-------------------------------------------------------
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 85694;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 51222;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 14006;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 48639;
```

L'identifiant de requête est un *hash* unique pour les requêtes dites normalisées, qui présentent la même forme sans considération des expressions variables. Cet identifiant, ou *query id*, a été introduit avec la contribution `pg_stat_statements` afin de regrouper des statistiques d'exécution d'une requête distincte pour chaque base et chaque utilisateur.

La méthode pour générer cet identifiant a été élargie globalement dans le code de PostgreSQL, rendant possible son exposition en dehors de `pg_stat_statements`. Les quelques composants de supervision en ayant bénéficié sont :

- la vue `pg_stat_activity` dispose à présent de sa colonne `query_id` ;

- le paramètre `log_line_prefix` peut afficher l'identifiant avec le nouveau caractère d'échappement `%Q` ;
- le mode `VERBOSE` de la commande `EXPLAIN`.

```
SET compute_query_id = on;
EXPLAIN (verbose, costs off)
 SELECT abalance FROM pgbench_accounts WHERE aid = 28742;

                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on public.pgbench_accounts
   Output: abalance
   Index Cond: (pgbench_accounts.aid = 28742)
 Query Identifier: 2691537454541915536
```

Dans l'exemple ci-dessus, le paramètre `compute_query_id` doit être activé pour déclencher la recherche de l'identifiant rattachée à une requête. Par défaut, ce paramètre vaut `auto`, c'est-à-dire qu'en l'absence d'un module externe comme l'extension `pg_stat_statements`, l'identifiant ne sera pas disponible.

```
CREATE EXTENSION pg_stat_statements;
SHOW compute_query_id ;

 compute_query_id
------------------
 auto

SELECT query_id, query FROM pg_stat_activity
 WHERE state = 'active';

      query_id       |                          query
---------------------+-------------------------------------------------------
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 85694;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 51222;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 14006;
 2691537454541915536 | SELECT abalance FROM pgbench_accounts WHERE aid = 48639;

SELECT query, calls, mean_exec_time FROM pg_stat_statements
 WHERE queryid = 2691537454541915536 \gx

-[ RECORD 1 ]--+-----------------------------------------------------
query          | SELECT abalance FROM pgbench_accounts WHERE aid = $1
calls          | 3786805
mean_exec_time | 0.009108110672981447
```

0.7.5 NOUVEAUTÉ DANS PG_LOCKS

- Ajout de la colonne `waitstart`
  - Heure à laquelle l'attente d'un verrou a commencé

```
       mode          | granted |      waitstart
---------------------+---------+---------------------
 AccessExclusiveLock | t       |
 AccessShareLock     | f       | 2021-08-26 15:54:53
```

La vue système `pg_locks` présente une nouvelle colonne `waitstart`. Elle indique l'heure à laquelle le processus serveur a commencé l'attente d'un verrou ou alors `null` si le verrou est détenu. Afin d'éviter tout surcoût, la mise à jour de cette colonne est faite sans poser de verrou, il est donc possible que la valeur de `waitstart` soit à `null` pendant une très courte période après le début d'une attente et ce même si la colonne `granted` est à `false`.

```
-- Une transaction pose un verrou
SELECT pg_backend_pid();
--  pg_backend_pid
-- ----------------
--           27829
BEGIN;
LOCK TABLE test_copy ;

-- Une autre transaction réalise une requête sur la même table
SELECT pg_backend_pid();
--  pg_backend_pid
-- ----------------
--           27680
SELECT * FROM test_copy ;

-- Via la vue pg_locks on peut donc voir qui bloque
-- le processus 27680 et également depuis quand
SELECT pid, mode, granted, waitstart
  FROM pg_locks WHERE pid in (27829,27680);
```

```
  pid  |        mode         | granted |           waitstart
-------+---------------------+---------+-------------------------------
 27829 | AccessExclusiveLock | t       |
 27680 | AccessShareLock     | f       | 2021-08-26 15:54:53.280405+02
```

0.8 PERFORMANCES

- Nettoyage des index B-tree
- Connexions simultanées en lecture seule

0.8.1 NETTOYAGE DES INDEX B-TREE

- Nettoyage des index B-tree « par le haut »
  - limite la fragmentation lorsque des lignes sont fréquemment modifiées

Lorsqu'une ligne est mise à jour par un ordre `UPDATE`, PostgreSQL garde l'ancienne version de la ligne dans la table jusqu'à ce qu'elle ne soit plus nécessaire à aucune transaction. L'adresse physique de chaque version est différente. Il faut donc ajouter cette nouvelle version à tous les index (y compris ceux pour lesquels la donnée n'a pas changé), afin de s'assurer qu'elle soit visible lors des parcours d'index. Ce processus est très pénalisant pour les performances et peut provoquer de la fragmentation.

La notion de *Heap Only Tuple* (HOT) a été mis en place pour palier ce problème. Lorsqu'une mise à jour ne touche aucune colonne indexée et que la nouvelle version de ligne peut être stockée dans la même page que les autres versions, PostgreSQL peut éviter la mise à jour des index.

Il y a cependant beaucoup de cas où il n'est pas possible d'éviter la mise à jour de colonnes indexées. Dans certains profils d'activité avec beaucoup de mise à jour, cela peut mener à la création de beaucoup d'enregistrements d'index correspondant à des versions différentes d'une même ligne dans la table, mais pour lequel l'enregistrement dans l'index est identique.

PostgreSQL 14 introduit un nouveau mécanisme pour limiter fortement la fragmentation due à des changements de versions fréquents d'une ligne de la table sans changement des données dans l'index. Lorsque ce genre de modifications se produit, l'exécuteur marque les tuples avec le hint *logically unchanged index*. Par la suite, lorsqu'une page menace de se diviser (*page split*), PostgreSQL déclenche un nettoyage des doublons de ce genre correspondant à des lignes mortes. Ce nettoyage est décrit comme *bottom up* (du bas vers le haut) car c'est la requête qui le déclenche lorsque la page va se remplir. Il se distingue du nettoyage qualifié de *top down* (de haut en bas) effectué par l'autovacuum.

Un autre mécanisme se déclenche en prévention d'une division de page : la suppression des entrées d'index marquées comme mortes lors d'*index scan* précédents (avec le flag `LP_DEAD`). Cette dernière est qualifiée de *simple index tuple deletion* (suppression simple de tuple d'index).

Si les nettoyages *top down* et *simple* ne suffisent pas, la déduplication tente de faire de la place dans la page. En dernier recours, la page se divise en deux (*page split*) ce qui fait grossir l'index.

Pour le tester, on peut comparer la taille des index sur une base `pgbench` après 1,5 millions de transactions en version 13 et 14. Rappelons que `pgbench` consiste essentiellement à mettre à jour les lignes d'une base, sans en ajouter ou supprimer. Les index par défaut étant des clés primaires ou étrangères, on ajoute aussi un index sur des valeurs qui changent réellement. Pour un test aussi court, on désactive l'autovacuum :

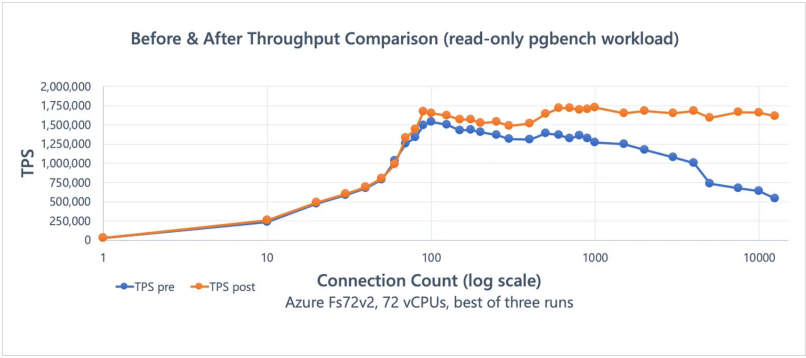
```
createdb bench
pgbench -i -s 100 bench --unlogged-tables
psql -X -d bench -c 'CREATE INDEX ON pgbench_accounts (abalance) ' -c '\di+'
psql -X -d bench -c 'ALTER TABLE pgbench_accounts SET (autovacuum_enabled = off)'
psql -X -d bench -c 'ALTER TABLE pgbench_history SET (autovacuum_enabled = off)'
pgbench -n -c 50 -t30000  bench -r -P10
psql -c '\di+' bench
```

À l'issue du pgbench, on constate que : * l'index `pgbench_accounts_abalance_idx` a une fragmentation quasi identique en version 13 et 14. Ce résultat est attendu car la colonne `abalance` est celle qui est mise à jour ; * l'index `pgbench_accounts_pkey` n'est pas fragmenté en version 14, contrairement à en la version 13. Là encore le résultat est attendu, la clé primaire n'est jamais modifiée, c'est le genre d'index que cible cette optimisation ; * les index `pgbench_branches_pkey` et `pgbench_tellers_pkey` sont très petits et la fragmentation n'est pas significative.

Name	Taille avant	Taille après (v13)	Taille après (v14)
pgbench_accounts_abalance_idx	66 Mo	80 Mo	79 MB
pgbench_accounts_pkey	214 Mo	333 Mo	214 MB
pgbench_branches_pkey	16 ko	56 ko	40 kB
pgbench_tellers_pkey	40 ko	224 ko	144 kB

Dans la réalité, l'autovacuum fonctionnera et nettoiera une partie des lignes au fil de l'eau, mais il peut être gêné par les autres transactions en cours. PostgreSQL 14 permettra donc d'éviter quelques `REINDEX`.

0.8.2 CONNEXIONS SIMULTANÉES EN LECTURE SEULE



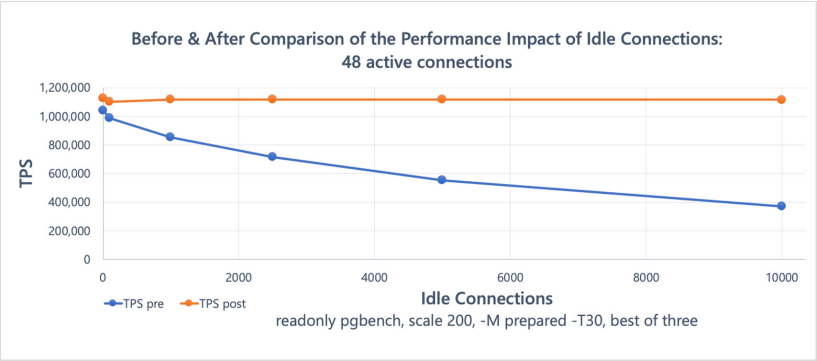
(Les graphiques sont empruntés à l'article de blog d'Andres Freund cité plus bas.)

Pour rappel, le mécanisme MVCC (*MultiVersion Concurrency Control*) de PostgreSQL facilite 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*).

Une série d'optimisations ont été apportées dans cette version 14 sur la gestion des *snapshots* induits par ce mécanisme lorsqu'un très grand nombre de connexions est atteint. Dans un mail destiné aux développeurs[2] , Andres Freund explique qu'une transaction consomme beaucoup de ressources pour actualiser l'état *xmin* de sa propre session, et que la méthode `GetSnapshotData()`, requise pour obtenir les informations sur les transactions du système, nécessitait de consulter l'état de chacune d'entre elles dans les zones mémoires de tous les CPU du serveur.

Dans un article à ce sujet[3] , l'auteur du patch indique que les bénéfices sont également remarquables lorsqu'un grand nombre de sessions inactives (*idle*) sont connectées à l'instance. Dans le *benchmark* suivant, on peut constater que les performances (TPS : *Transactions Per Second*) restent stables pour 48 sessions actives à mesure que le nombre de sessions inactives augmentent.

[2] https://www.postgresql.org/message-id/flat/20200301083601.ews6hz5dduc3w2se@alap3.anarazel.de
[3] https://www.citusdata.com/blog/2020/10/25/improving-postgres-connection-scalability-snapshots/



La solution consiste à changer la méthode `GetSnapshotData()` afin que seules les informations *xmin* des transactions en écriture soient accessibles depuis un cache partagé. Dans une architecture où les lectures sont majoritaires, cette astuce permet de reconstituer les instantanés bien plus rapidement, augmentant considérablement la quantité d'opérations par transaction.

PostgreSQL 14 est donc un gros progrès pour les instances supportant de très nombreuses connexions simultanées, actives ou non.

0.9 QUESTIONS

- *Merci de votre écoute !*