Workshop 14

Nouveautés de PostgreSQL 14

14

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 14

Workshop 14

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

REVISION: 14

Table des Matières

1 NOUVEAUTÉS DE POSTGRESQL 14

Photographie d'Antoine Taveneaux, licence CC BY-SA 3.0[1] , obtenue sur wikimedia.org[2] .

Participez à ce workshop !

Pour des précisions, compléments, liens, exemples, et autres corrections et suggestions, soumettez vos *Pull Requests* dans notre dépôt :

https://github.com/dalibo/workshops/tree/master/fr

Licence : PostgreSQL[3]

Ce workshop sera maintenu encore plusieurs mois après la sortie de la version 14.

---

[1] https://creativecommons.org/licenses/by-sa/3.0/deed.en
[2] https://commons.wikimedia.org/wiki/File:Lakshmana_Temple_10.jpg
[3] https://github.com/dalibo/workshops/blob/master/LICENSE.md

1.1 LES NOUVEAUTÉS

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

PostgreSQL 14 est sorti le 30 septembre 2021.

Les points principaux sont décrits dans le « press kit[4] ».

Nous allons décrire ces nouveautés plus en détail.

---

[4] https://www.postgresql.org/about/press/presskit14/

1.2 ADMINISTRATION ET MAINTENANCE

- Sécurité
- Configuration
- Outils clients
- Partitionnement
- Divers

1.2.1 SÉCURITÉ

1.2.1.1 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[5]

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 ;
```

[5] 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
```

1.2.1.2 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.

1.2.2 NOUVEAUTÉS DE CONFIGURATION (GUC)

1.2.2.1 Nouveaux caractères d'échappement pour log_line_prefix

`log_line_prefix` : enrichit le préfixe des lignes de la sortie d'erreur
- `%P` : identifiant du processus principal (*parallel leader*)
  - si l'entrée de journal provient d'un processus auxiliaire (*parallel worker*)
- `%Q` : identifiant de la requête (nouveauté)
  - si le calcul interne de l'identifiant est actif

Le paramètre `log_line_prefix` accepte deux nouveaux caractères d'échappement.

- `%P` pour l'identifiant de processus *leader* dans un groupe parallélisé

Déjà présent depuis la version 13 dans la vue `pg_stat_activity`, l'identifiant de processus *leader pid* est désormais disponible pour faciliter la compréhension des événements dans la sortie d'erreur.

- `%Q` pour l'identifiant de la requête *query id*

Le calcul interne pour l'identifiant de requête est une nouveauté de la version 14 et sera abordé plus loin dans ce workshop.

Prenons une instance dont la configuration est semblable à ce qui suit :

```
compute_query_id = on
log_temp_files = 0
log_min_duration_statement = 0
max_parallel_workers_per_gather = 8
log_line_prefix = '[%P-%p]: id=%Q '
```

On constate dans un extrait des traces, qu'une requête d'agrégat bénéficie de quatre processus auxiliaires (pid 20992 à 20995) et d'un processus principal (pid 20969). Chacun de ses processus partage le même identifiant de requête -8329068551672606797.

```
[20969-20995]: id=-8329068551672606797
    LOG:  temporary file: path "pgsql_tmp20995.0", size 29450240
[20969-20993]: id=-8329068551672606797
    LOG:  temporary file: path "pgsql_tmp20993.0", size 52682752
[20969-20994]: id=-8329068551672606797
    LOG:  temporary file: path "pgsql_tmp20994.0", size 53387264
[20969-20992]: id=-8329068551672606797
    LOG:  temporary file: path "pgsql_tmp20992.0", size 53477376
[-20969]: id=-8329068551672606797
    LOG:  temporary file: path "pgsql_tmp20969.0", size 29384704
```

```
[-20969]: id=-8329068551672606797
    LOG:  duration: 2331.661 ms
    statement:  select trunc(montant/1000) as quintile, avg(montant)
                from ventes group by quintile;
```

1.2.2.2 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
```

1.2.2.3 *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.

1.2.2.4 *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.

1.2.2.5 Changements mineurs de configuration

- VACUUM
  - `vacuum_cost_page_miss` = 2 (autrefois : 10)
- Checkpoint :
  - `checkpoint_completion_target` = 0.9 par défaut
- Nouveaux paramètres :
  - `huge_page_size`
  - `log_recovery_conflict_waits`

La version 14 apporte quelques modifications mineures de configuration :

vacuum_cost_page_miss :

Sa valeur par défaut passe de 10 à 2. Cette modification diminue la pénalité associée à la lecture de page qui ne sont pas dans le cache de l'instance par le *vacuum*. Cela va donc permettre à l'autovacuum (voire le vacuum si `vacuum_cost_delay` est supérieur à zéro) de traiter plus de pages avant de se mettre en pause. Ce changement reflète l'amélioration des performances des serveurs due à l'évolution du stockage et à la quantité de RAM disponible.

checkpoint_completion_target :

La valeur par défaut passe de 0.5 à 0.9. Ce paramétrage était déjà largement adopté par la communauté et permet de lisser les écritures faites lors d'un `CHECKPOINT` sur une durée plus longue (90 % de `checkpoint_timeout`, qui vaut 5 minutes). Cela a pour effet de limiter les pics d'IO suite aux *checkpoints*.

huge_page_size :

Ce paramètre permet de surcharger la configuration système pour la taille des *Huge Pages*. Par défaut, PostgreSQL utilisera la valeur du système d'exploitation.

log_recovery_conflict_waits :

Ce paramètre, une fois activé, permet de tracer toute attente due à un conflit de réplication. Il n'est donc intéressant et pris en compte que sur un serveur secondaire.

1.2.3 OUTILS CLIENTS

1.2.3.1 *pg_dump/pg_restore : Possibilité d'exporter et restaurer des partitions individuellement*

- `pg_dump` au format *custom*, *directory* ou *tar*
  - L'instruction `CREATE TABLE` ne dépend plus du résultat de la commande `ATTACH`
- `pg_restore` et l'option `--table`
  - Restauration possible d'une partition en tant que simple table
  - Plus simple qu'une restauration avec un *fichier liste* (*table of contents*)

La restauration d'une partition en tant que simple table est particulièrement utile lors de la restauration d'une archive ou d'une copie d'environnement avec les données les plus récentes. Bien qu'il était auparavant possible de restaurer la table partitionnée et l'une de ses partitions avec `pg_restore` et l'option `-L / --use-list`, le format des archives *custom*, *directory* et *tar* a évolué pour simplifier cette opération.

Prenons l'exemple d'une table des ventes, partitionnée sur la colonne `date_vente` avec une partition pour chaque année écoulée.

```
           Partitioned table "ventes"
   Column    |  Type   | Collation | Nullable | Default
-------------+---------+-----------+----------+---------
 date_vente | date    |           | not null |
 projet_id  | bigint  |           |          |
 montant    | numeric |           |          |
Partition key: RANGE (date_vente)
Number of partitions: 22 (Use \d+ to list them.)
Partitions: ventes_y2001 FOR VALUES FROM ('2001-01-01') TO ('2002-01-01'),
            ventes_y2002 FOR VALUES FROM ('2002-01-01') TO ('2003-01-01'),
            ...
            ventes_y2021 FOR VALUES FROM ('2021-01-01') TO ('2022-01-01'),
            ventes_y2022 FOR VALUES FROM ('2022-01-01') TO ('2023-01-01')
```

Dans un souci de gestion des données à archiver, la table la plus ancienne `ventes_y2001` est exportée pour libérer l'espace disque. À l'aide de la commande `pg_dump` au format *custom*, nous procédons à la création du fichier d'archive.

```
pg_dump -Fc -d workshop -t ventes_y2001 -f ventes.dump
# pg_dump: dumping contents of table "ventes_y2001"
```

Le fichier dispose de la liste des instructions dans le contenu d'archive (ou *table of contents*), dont la nouvelle ligne `ATTACH` pour la partition exportée.

```
pg_restore -l ventes.dump
```

```
; Selected TOC Entries:
;
210; 1259 18728 TABLE public ventes_y2001 user
2449; 0 0 TABLE ATTACH public ventes_y2001 user
2589; 0 18728 TABLE DATA public ventes_y2001 user
```

La restauration de cette seule partition est possible, même en l'absence de la table partitionnée sur la base cible.

```
pg_restore -v -d staging -t ventes_y2001 ventes.dump
```

```
pg_restore: connecting to database for restore
pg_restore: creating TABLE "ventes_y2001"
pg_restore: processing data for table "ventes_y2001"
```

Dans les versions précédentes, l'erreur suivante empêchait ce type de restauration et la table principale devait être présente pour aboutir au résultat souhaité.

```
pg_restore: connecting to database for restore
pg_restore: creating TABLE "ventes_y2001"
pg_restore: while PROCESSING TOC:
pg_restore: from TOC entry 201; 1259 25987 TABLE ventes_y2001 user
pg_restore: error: could not execute query:
ERROR:  relation "ventes" does not exist
Command was: CREATE TABLE ventes_y2001 (
    date_vente date NOT NULL,
    projet_id bigint,
    montant numeric
);
ALTER TABLE ONLY ventes ATTACH PARTITION ventes_y2001
  FOR VALUES FROM ('2001-01-01') TO ('2002-01-01');

pg_restore: error: could not execute query:
ERROR:  relation "ventes_y2001" does not exist
Command was: ALTER TABLE ventes_y2001 OWNER TO user;

pg_restore: processing data for table "ventes_y2001"
pg_restore: from TOC entry 2286; 0 25987 TABLE DATA ventes_y2001 user
pg_restore: error: could not execute query:
ERROR:  relation "ventes_y2001" does not exist
Command was: COPY ventes_y2001 (date_vente, projet_id, montant) FROM stdin;
pg_restore: warning: errors ignored on restore: 3
```

1.2.3.2 *pg_dump : Nouvelle option pour exporter les extensions*

- `pg_dump --extension=…`
  - spécifier un sous-ensemble d'extensions à exporter
  - exporter les extensions même avec `--schema`

Dans les versions précédentes, les extensions d'une base de données étaient exportées par `pg_dump`, mais il n'était pas possible de préciser les extensions que l'on souhaitait embarquer dans l'export. De plus, le fait de préciser un schéma avec l'option `--schema=MOTIF` ou `-n MOTIF` excluait les extensions de l'export, car elles étaient considérées comme liées à la base de données plutôt qu'au schéma.

Prenons l'exemple de l'extension `pgcrypto` installée sur une base de données `workshop` qui contient le schéma `encrypted_data` que l'on souhaite exporter. Deux exports du schéma `encrypted_data` sont réalisés avec `pg_dump`. L'option `--extension` n'est spécifiée que pour le second.

```
pg_dump --format=custom \
        --dbname=workshop \
        --schema=encrypted_data > /backup/workshop_encrypted_data.dmp

pg_dump --format=custom \
        --dbname=workshop \
        --extension=pgcrypto \
        --schema=encrypted_data > /backup/workshop_encrypted_data_with_ext.dmp
```

La base de données `workshop` est alors supprimée et le premier dump `encrypted_data.dmp` est importé à l'aide de `pg_restore`.

```
dropdb workshop

pg_restore --dbname postgres \
           --create \
           /backup/encrypted_data.dmp
```

On constate alors que l'extension est absente, elle n'a donc pas été incluse dans le premier dump. Cela peut gêner lors de la restauration des données.

```
workshop=# \dx

Liste des extensions installées
-[ RECORD 1 ]--------------------------
Nom         | plpgsql
Version     | 1.0
Schéma      | pg_catalog
Description | PL/pgSQL procedural language
```

La base `workshop` est à nouveau supprimée.
Puis `workshop_encrypted_data_with_ext.dmp` est ensuite importé à l'aide de `pg_restore`.

```
dropdb workshop

pg_restore --dbname postgres \
           --create \
           /backup/workshop_encrypted_data_with_ext.dmp
```

En listant les extensions de la base de données, on constate cette fois que l'extension `pgcrypto` a été restaurée dans la base de données `workshop`.

```
workshop=# \dx
Liste des extensions installées
-[ RECORD 1 ]-------------------------------------------------
Nom         | pgcrypto
Version     | 1.9
Schéma      | public
Description | track planning and execution statistics of all SQL
            | statements executed
-[ RECORD 2 ]-------------------------------------------------
Nom         | plpgsql
Version     | 1.0
Schéma      | pg_catalog
Description | PL/pgSQL procedural language
```

Lorsque `--schema` est utilisé, aucune extension n'est donc incluse dans l'export, à moins d'utiliser la nouvelle option `--extension`.

Dans l'export d'une base entière, le comportement par défaut reste d'inclure les extensions.

1.2.4 PARTITIONNEMENT

1.2.4.1 ALTER TABLE ... DETACH PARTITION ... CONCURRENTLY

- Détachement de partition non bloquant
- Fonctionne en mode multi-transactions
- Quelques restrictions :
  - Ne fonctionne pas dans un bloc de transactions
  - Impossible en cas de partition par défaut

Détacher une partition peut maintenant se faire de façon non bloquante grâce à la commande `ALTER TABLE … DETACH PARTITION … CONCURRENTLY`.

Son fonctionnement repose sur l'utilisation de deux transactions :

- La première ne requiert qu'un verrou `SHARE UPDATE EXCLUSIVE` sur la table partitionnée et la partition. Pendant cette phase, la partition est marquée comme en cours de détachement, la transaction est validée et on attend que toutes les transactions qui utilisent la partition se terminent. Cette phase est nécessaire pour s'assurer que tout le monde voit le changement de statut de la partition.
- Pendant la seconde, un verrou `SHARE UPDATE EXCLUSIVE` est placé sur la table partitionnée et un verrou `ACCESS EXCLUSIVE` sur la partition pour terminer le processus de détachement.

Dans le cas d'une annulation ou d'un crash lors de la deuxième transaction, la commande `ALTER TABLE … DETACH PARTITION … FINALIZE` devra être exécutée pour terminer l'opération.

Lors de la séparation d'une partition, une contrainte `CHECK` est créée à l'identique de la contrainte de partitionnement. Ainsi, en cas de ré-attachement de la partition, le système n'aura pas besoin d'effectuer un parcours de la table pour valider la contrainte de partition si cette contrainte existe. Sans elle, la table serait parcourue entièrement pour valider la contrainte de partition tout en nécessitant un verrou de niveau `ACCESS EXCLUSIVE` sur la table parente.

La contrainte peut bien entendu être supprimée après le ré-attachement de la partition afin d'éviter des doublons.

L'exemple suivant porte sur une table partitionnée avec deux partitions :

```
\d+ parent
```

```
                    Table partitionnée « public.parent »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
Clé de partition : RANGE (id)
Partitions: enfant_1 FOR VALUES FROM (0) TO (5000000),
            enfant_2 FOR VALUES FROM (5000000) TO (11000000)

\d enfant_2

                    Table « public.enfant_2 »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
Partition de : parent FOR VALUES FROM (5000000) TO (11000000)
Index :
    "enfant_2_id_idx" btree (id)
```

Nous allons procéder au détachement de la partition `enfant_2` :

```
ALTER TABLE parent DETACH PARTITION enfant_2 CONCURRENTLY ;

-- Une contrainte CHECK a été créée
-- Celle-ci correspond à la contrainte de partition
\d enfant_2

                    Table « public.enfant_2 »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
Index :
    "enfant_2_id_idx" btree (id)
Contraintes de vérification :
    "enfant_2_id_check" CHECK (id IS NOT NULL AND id >= 5000000 AND id < 11000000)
```

Concernant les restrictions :

- Il n'est pas possible d'utiliser `ALTER TABLE … DETACH PARTITION … CONCURRENTLY` dans un bloc de transactions à cause de son mode *multi-transactions*.

```
BEGIN;
ALTER TABLE parent DETACH PARTITION enfant_2 CONCURRENTLY;
-- ERROR:  ALTER TABLE ... DETACH CONCURRENTLY cannot run inside a transaction block
```

- Il est impossible d'utiliser cette commande si une partition par défaut existe car les contraintes associées sont trop importante pour le mode concurrent. En effet, il faut obtenir un verrou de type `EXCLUSIVE LOCK` sur la partition par défaut.

```
ALTER TABLE parent DETACH PARTITION enfant_1 CONCURRENTLY;
-- ERROR:  cannot detach partitions concurrently when a default partition exists
```

1.2.4.2 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.

1.3

1.3.1 DIVERS

1.3.1.1 Compression des TOAST configurable en lz4 ou pglz

- Historiquement : `pglz`
- Nouveau : `lz4`, plus rapide
- Définition :
  - `SET default_toast_compression = …`
  - `ALTER TABLE … SET COMPRESSION …`
- Compatibilité : `pg_dump --no-toast-compression`
- N'affecte pas le fonctionnement de la réplication

Historiquement, le seul algorithme de compression disponible dans PostgreSQL était `pglz`. À présent, il est possible d'utiliser `lz4` et de définir un type de compression jusqu'au niveau d'une colonne.

De manière générale, `lz4` est nettement plus rapide à (dé)compresser, pour un taux de compression légèrement plus faible que l'algorithme historique.

Afin de pouvoir utiliser `lz4`, il faudra veiller à ce que PostgreSQL ait bien été compilé avec l'option `--with-lz4` et que le paquet `liblz4-dev` pour Debian ou `lz4-devel` pour RedHat soit installé. Les paquets précompilés du PGDG incluent cela.

```
# Vérification des options de compilation de PostgreSQL
postgres@pop-os:~$ pg_config | grep 'with-lz4'
CONFIGURE =  [...] '--with-lz4' [...]
```

Plusieurs options sont disponibles pour changer le mode de compression :

- Au niveau de la colonne, lors des opérations de `CREATE TABLE` et `ALTER TABLE`.

```
test=# CREATE TABLE t1 (champ1 text COMPRESSION lz4);

test=# \d+ t1
                                                Table « public.t1 »
 Colonne | Type | Collationnement | NULL-able | Par défaut | Stockage | Compression
---------+------+-----------------+-----------+------------+----------+-------------
 champ1  | text |                 |           |            | extended | lz4

test=# ALTER TABLE t1 ALTER COLUMN champ1 SET COMPRESSION pglz;

test=# \d+ t1
                                                Table « public.t1 »
 Colonne | Type | Collationnement | NULL-able | Par défaut | Stockage | Compression
```

```
---------+------+-----------------+----------+------------+----------+-------------
 champ1  | text |                 |          |            | extended | pglz
```

- Via le paramètre `default_toast_compression` dans le fichier `postgresql.conf`, la valeur par defaut étant `pglz`. Sa modification ne nécessite qu'un simple rechargement de la configuration de l'instance. Ce paramètre étant global à l'instance, il n'est pas prioritaire sur la clause `COMPRESSION` des commandes `CREATE TABLE` et `ALTER TABLE`.

```
test=# SHOW default_toast_compression;
 default_toast_compression
---------------------------
 pglz

test=# SET default_toast_compression TO lz4;

test=# SHOW default_toast_compression;
 default_toast_compression
---------------------------
 lz4
```

La modification du type de compression, qu'elle soit globale ou spécifique à un objet, n'entraînera aucune réécriture, seules les futures données insérées seront concernées. Il est donc tout à fait possible d'avoir des lignes compressées différemment dans une même table.

Pour le voir, une nouvelle fonction est également disponible : `pg_column_compression()` retourne l'algorithme de compression qui a été utilisé lors de l'insertion d'une ligne. Il peut y en avoir plusieurs :

```
test=# SHOW default_toast_compression ;
 default_toast_compression
---------------------------
 pglz
test=# CREATE TABLE t2 (champ2 text);

test=# INSERT INTO t2 VALUES (repeat('123456789', 5000));

test=# SET default_toast_compression TO lz4;

test=# INSERT INTO t2 VALUES (repeat('123456789', 5000));

test=# SELECT pg_column_compression(champ2) FROM t2;
 pg_column_compression
-----------------------
 pglz
 lz4
```

Point particulier concernant les commandes de type `CREATE TABLE AS`, `SELECT INTO` ou encore `INSERT ... SELECT`, les valeurs déjà compressées dans la table source ne seront pas recompressées lors de l'insertion pour des raisons de performance.

```
test=# SELECT pg_column_compression(champ2) FROM t2;
 pg_column_compression
-----------------------
 pglz
 lz4

test=# CREATE TABLE t3 AS SELECT * FROM t2;

test=# SELECT pg_column_compression(champ2) FROM t3;
 pg_column_compression
-----------------------
 pglz
 lz4
```

Concernant la réplication, il est possible de rejouer les WAL qui contiennent des données compressées avec `lz4` sur une instance secondaire via les réplications physique ou logique même si celle-ci ne dispose pas de `lz4`.

Principal inconvénient de la réplication physique, toute tentative de lecture de ces données entraînera une erreur.

La réplication logique n'est pas impactée par ce problème, les données seront compressées en utilisant l'algorithme configuré sur le secondaire. Il faudra cependant faire attention en cas d'utilisation d'algorithmes différents entre primaire et secondaire notamment au niveau de la volumétrie et du temps nécessaire à la compression.

Un exemple simple afin de mettre en évidence la différence entre les deux algorithmes :

```
test=# \d+ compress_pglz
                                              Table « public.compress_pglz »
 Colonne | Type | Collationnement | NULL-able | Par défaut | Stockage | Compression
---------+------+-----------------+-----------+------------+----------+-------------
 champ1  | text |                 |           |            | extended | pglz

test=# \d+ compress_lz4
                                              Table « public.compress_lz4 »
 Colonne | Type | Collationnement | NULL-able | Par défaut | Stockage | Compression
---------+------+-----------------+-----------+------------+----------+-------------
 champ1  | text |                 |           |            | extended | lz4

-- Comparaison à l'insertion des données
test=# INSERT INTO compress_pglz
         SELECT repeat('123456789', 100000) FROM generate_series(1,10000);
```

```
Durée : 36934,700 ms

test=# INSERT INTO compress_lz4
         SELECT repeat('123456789', 100000) FROM generate_series(1,10000);
Durée : 2367,150 ms
```

Le nouvel algorithme est donc beaucoup plus performant.

```
# SELECT
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid::regclass::text AS table_toast,
    reltuples AS nb_lignes_estimees,
    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_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname LIKE 'compress%' \gx

-[ RECORD 1 ]------+-------------------------
table              | public.compress_lz4
table_toast        | pg_toast.pg_toast_357496
nb_lignes_estimees | 10000
  Heap             | 512 kB
  Toast            | 39 MB
  Toast (PK)       | 456 kB
Total              | 40 MB
-[ RECORD 2 ]------+-------------------------
table              | public.compress_pglz
table_toast        | pg_toast.pg_toast_357491
nb_lignes_estimees | 10000
  Heap             | 512 kB
  Toast            | 117 MB
  Toast (PK)       | 1328 kB
Total              | 119 MB
```

Dans ce cas précis, `lz4` est plus efficace à la compression. Ce n'est pas le cas général, comme constaté dans cet article de Fujitsu[6] : `lz4` est généralement un peu moins efficace en compression.

Afin d'éviter les problèmes de compatibilité avec des versions plus anciennes, l'option `--no-toast-compression` a été ajoutée à `pg_dump`. Elle permet de ne pas exporter les méthodes de compression définies avec `CREATE TABLE` et `ALTER TABLE`.

Pour les données déjà insérées et compressées, s'il y a un besoin de change-

[6] https://www.postgresql.fastware.com/blog/what-is-the-new-lz4-toast-compression-in-postgresql-14

ment ou d'unification des algorithmes employés, il faudra le forcer par une procédure d'export/import. Avec cette méthode, les lignes seront réinsérées en utilisant la clause `COMPRESSION` des colonnes concernées ou à défaut le paramètre `default_toast_compression`.

1.3.1.2 Nouvelle option pour VACUUM : PROCESS_TOAST

- Traite les tables de débordement TOAST lors d'un `VACUUM` manuel
- Activé par défaut

```
VACUUM (PROCESS_TOAST false) blog;
```

`VACUUM` dispose désormais de l'option `PROCESS_TOAST` qui permet de lui spécifier s'il doit traiter ou non les tables TOAST. C'est un booléen et il est positionné à `true` par défaut.

À `false`, ce paramètre pourra être particulièrement utile pour accélérer un `VACUUM` si le taux de fragmentation (*bloat*) ou l'âge des transactions diffère grandement entre la table principale et la table TOAST, pour ne pas perdre de temps sur cette dernière. Les TOAST sont toujours concernées par un `VACUUM FULL`.

Dans cet exemple, on dispose d'une table `blog` avec une table TOAST associée :

```
test=# SELECT relname, reltoastrelid::regclass AS reltoastname
         FROM pg_class WHERE relname = 'blog';

test=# \d blog

 relname |      reltoastname
---------+-------------------------
 blog    | pg_toast.pg_toast_16565

                    Table « public.blog »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 title   | text    |                 |           |
 content | text    |                 |           |
```

Après lancement d'un VACUUM sans l'option `PROCESS_TOAST`, l'horodatage du traitement, à travers la vue `pg_stat_all_tables`, montre que la table `blog` et la table TOAST associée ont bien été traitées par le VACUUM.

```
test=# VACUUM blog;
test=# SELECT relname, last_vacuum FROM pg_stat_all_tables
        WHERE relname IN ('blog', 'pg_toast_16565');
```

```
    relname     |          last_vacuum
----------------+-------------------------------
 blog           | 2021-08-16 12:03:43.994759+02
 pg_toast_16565 | 2021-08-16 12:03:43.994995+02
```

Lors d'un lancement d'un VACUUM avec l'option PROCESS_TOAST, seule la table principale est traitée par le VACUUM :

```
test=# VACUUM (PROCESS_TOAST false) blog;
test=# SELECT relname, last_vacuum FROM pg_stat_all_tables
        WHERE relname IN ('blog', 'pg_toast_16565');

    relname     |          last_vacuum
----------------+-------------------------------
 blog           | 2021-08-16 12:06:04.745281+02
 pg_toast_16565 | 2021-08-16 12:03:43.994995+02
```

Cette fonctionnalité est également disponible avec la commande `vacuumdb --no-process-toast`.

1.3.1.3 Nouvelle option pour REINDEX : TABLESPACE

- Ajout de l'option TABLESPACE pour la commande REINDEX
- Possibilité de déplacer des index vers un autre tablespace tout en les reconstruisant
- Avec ou sans la clause CONCURRENTLY
- Restrictions :
  - sur les tables et index partitionnés
  - sur les tables TOAST
  - sur le catalogue système

La commande REINDEX dispose avec cette nouvelle version de l'option TABLESPACE qui donne la possibilité de déplacer des index dans un autre tablespace durant leur reconstruction. Son utilisation avec la clause CONCURRENTLY est supportée.

```
-- On dispose d'une table t1 avec un index idx_col1 dans le tablespace pg_default.
test=# \d t1
                    Table « public.t1 »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 col1    | integer |                 |           |
Index :
    "idx_col1" btree (col1)

-- Réindexation de la table t1 et déplacement de l'index idx_col1 dans le tablespace tbs.
```

```
test=# REINDEX (TABLESPACE tbs) TABLE t1 ;

-- L'index a bien été déplacé.
test=# SELECT c.relname, t.spcname FROM pg_class c
         JOIN pg_tablespace t ON (c.reltablespace = t.oid)
        WHERE c.relname = 'index_col1';

  relname   | spcname
------------+---------
 index_col1 | tbs
```

Quelques restrictions s'appliquent :

- Lors de l'utilisation de l'option `TABLESPACE` sur des tables et index partitionnés, seuls les index des partitions seront déplacés vers le nouveau tablespace. Aucune modification du tablespace ne sera effectuée dans `pg_class.reltablespace`, il faudra pour cela utiliser la commande `ALTER TABLE SET TABLESPACE`. Afin de déplacer l'index parent, il faudra passer par la commande `ALTER INDEX SET TABLESPACE`.

```
-- On dispose de la table partionnée suivante.
test=# SELECT * FROM pg_partition_tree('parent');
  relid   | parentrelid | isleaf | level
----------+-------------+--------+-------
 parent   |             | f      |     0
 enfant_1 | parent      | t      |     1
 enfant_2 | parent      | t      |     1

-- Avec un index dans la table parent et dans chaque partition.
test=# SELECT * FROM pg_partition_tree('parent_index');
      relid      | parentrelid  | isleaf | level
-----------------+--------------+--------+-------
 parent_index    |              | f      |     0
 enfant_1_id_idx | parent_index | t      |     1
 enfant_2_id_idx | parent_index | t      |     1

-- Tous les index sont dans le tablespace pg_default.
test=# SELECT c.relname, CASE
                WHEN c.reltablespace = 0 THEN td.spcname
                ELSE tr.spcname
              END spcname
         FROM pg_partition_tree('parent_index') p
         JOIN pg_class c ON (c.oid = p.relid)
         JOIN pg_database d ON (d.datname = current_database())
         JOIN pg_tablespace td ON (d.dattablespace = td.oid)
    LEFT JOIN pg_tablespace tr ON (c.reltablespace = tr.oid);

     relname     |  spcname
```

```
-----------------+------------
 parent_index    | pg_default
 enfant_1_id_idx | pg_default
 enfant_2_id_idx | pg_default

-- Reindexation de la table parent avec l'option TABLESPACE.
test=# REINDEX (TABLESPACE tbs) TABLE parent;

-- Seuls les index des partitions ont été déplacés.
test=# SELECT c.relname, CASE
                WHEN c.reltablespace = 0 THEN td.spcname
                ELSE tr.spcname
              END spcname
         FROM pg_partition_tree('parent_index') p
         JOIN pg_class c ON (c.oid = p.relid)
         JOIN pg_database d ON (d.datname = current_database())
         JOIN pg_tablespace td ON (d.dattablespace = td.oid)
    LEFT JOIN pg_tablespace tr ON (c.reltablespace = tr.oid);

     relname     |  spcname
-----------------+------------
 parent_index    | pg_default
 enfant_1_id_idx | tbs
 enfant_2_id_idx | tbs
```

- Les index des tables TOAST sont conservés dans leur tablespace d'origine. Ils seront déplacés avec la table TOAST si la table utilisateur rattachée est déplacée.

```
-- On dispose d'une table blog avec une table TOAST.
test=# \d blog
                    Table « public.blog »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 title   | text    |                 |           |
 content | text    |                 |           |
Index :
    "blog_title_idx" btree (title)

test=# \d+ pg_toast.pg_toast_16417
Table TOAST « pg_toast.pg_toast_16417 »
  Colonne   |  Type   | Stockage
------------+---------+----------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Table propriétaire : « public.blog »
```

```
Index :
    "pg_toast_16417_index" PRIMARY KEY, btree (chunk_id, chunk_seq)

-- réindexation de la table blog
test=# REINDEX (TABLESPACE tbs) TABLE blog;

-- Seul l'index de la table blog à été déplacé.
-- Celui de la table TOAST a uniquement été reconstruit.
test=# SELECT c.relname, t.spcname FROM pg_class c
         JOIN pg_tablespace t ON (c.reltablespace = t.oid)
        WHERE t.spcname = 'tbs';

     relname     | spcname
-----------------+---------
blog_title_idx   | tbs

-- Test de déplacement d'un index directement sur une table TOAST.
test=# reindex (tablespace tbs) table pg_toast.pg_toast_16417;
ERROR:  cannot move system relation "pg_toast_16417_index"
```

- L'option est interdite sur le catalogue système. Lors de l'utilisation des commandes `REINDEX SCHEMA`, `DATABASE` ou `TABLE` les objets systèmes ne seront pas concernés par le déplacement si l'option `TABLESPACE` est utilisée.

```
-- Test d'un déplacement d'index sur une table système.
test=# REINDEX (TABLESPACE tbs) TABLE pg_aggregate;
ERROR:  cannot move system relation "pg_aggregate_fnoid_index"

-- Test d'une réindexation de BDD avec déplacement des index.
test=# REINDEX (TABLESPACE tbs) DATABASE test;
WARNING:  cannot move system relations, skipping all
REINDEX
```

Cette fonctionnalité est également disponible avec la commande `reindexdb --tablespace`.

1.3.1.4 Nouvelle fonction pour attendre lorsque l'on arrête un backend

`pg_terminate_backend ( pid integer, timeout bigint DEFAULT 0 )`

- Possibilité d'attendre l'arrêt du backend
- Nouveau paramètre `timeout`

Il est désormais possible d'attendre l'arrêt du backend ciblé par l'exécution de `pg_terminate_backend()` pendant un temps configuré avec le nouveau paramètre `timeout` de cette fonction :

```
# \df pg_terminate_backend

List of functions
-[ RECORD 1 ]-------+-------------------------------------
Schema              | pg_catalog
Name                | pg_terminate_backend
Result data type    | boolean
Argument data types | pid integer, timeout bigint DEFAULT 0
Type                | fun
```

Ce paramètre est exprimé en millisecondes et est configuré à 0 par défaut, ce qui signifie que l'on n'attend pas.

En le configurant à une valeur positive, on attendra que le backend soit arrêté ou que le timeout configuré soit atteint. Si le timeout est atteint, un message d'avertissement sera affiché à l'écran et la fonction renverra `false` :

```
# SELECT pg_terminate_backend(358855, 200);

WARNING:  backend with PID 358818 did not terminate within 200 milliseconds
-[ RECORD 1 ]--------+--
pg_terminate_backend | f
```

1.4 RÉPLICATION ET SHARDING

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

1.4.1 RÉPLICATION PHYSIQUE

1.4.1.1 *Autorise pg_rewind à utiliser un secondaire comme source*

- La source d'un *rewind* peut être une instance secondaire

`pg_rewind` permet de synchroniser le répertoire de données d'une instance avec un autre répertoire de données de la même instance. Il est notamment utilisé lorsqu'une instance d'un dispositif en réplication à divergé de l'instance primaire. Cela peut arriver à l'ancienne primaire lors d'un *failover*. `pg_rewind` permet alors de raccrocher l'ancienne instance primaire sans avoir besoin de restaurer une sauvegarde ou de cloner l'instance primaire avec `pg_basebackup`.

Lorsque l'on dispose de plus de deux serveurs dans une architecture répliquée, il est désormais possible d'utiliser une instance secondaire comme source du *rewind*. Cela permet de limiter l'impact des lectures sur la nouvelle instance primaire.

Précédemment, PostgreSQL utilisait une table temporaire pour stocker certaines information le temps du *rewind*. Une réécriture du code a rendu cette étape inutile. C'est cette modification qui permet l'utilisation des instances secondaires comme source d'un *rewind*.

1.4.1.2 *Nouveau paramètre de connexion dans libpq*

- Nouvelles options pour le paramètre `target_session_attrs`
  - `read-only`, `primary`, `standby`, et `prefer-standby`

Une chaîne de connexion `libpq` peut contenir plusieurs serveurs. Le paramètre `target_session_attrs` permet de préciser quel type de serveur l'on veut[7] .

En plus des options `any` (qui reste celle par défaut) et `read-write` (choisir un serveur ouvert en écriture), `target_session_attrs` supporte désormais les options suivantes :

- `read-only`, le serveur ne doit accepter que les sessions en lecture seule (mode *hot standby* ou `default_transaction_read_only` à `on`) ;
- `primary`, le serveur ne doit pas être en mode *hot standby* ;
- `standby`, le serveur doit être en mode *hot standby* ;
- `prefer-standby`, dans un premier temps, essayer de trouver une instance secondaire, sinon utilise le mode `any`.

[7] https://docs.postgresql.fr/14/libpq-connect.html#LIBPQ-MULTIPLE-HOSTS

Des exemples de chaînes de connexion avec paramètre sont :

```
'postgresql://host1:123,host2:456/somedb?target_session_attrs=any'
'postgresql://host1:123,host2:456/somedb?target_session_attrs=read-write'
'host=serveur1,serveur2,serveur3 port=5432,5433,5434 target_session_attrs=read-only'
```

Avec ces nouvelles options, aucune communication réseau supplémentaire ne sera nécessaire pour obtenir l'état de la session ou du serveur. Les variables GUC fournies sont suffisantes. Dans les versions plus anciennes, une requête `SHOW` ou `SELECT` devait être émise afin de détecter si la session était en lecture seule ou si l'instance était en mode *hot standby*.

1.4.2 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;
```

1.4.3 FOREIGN DATA WRAPPER ET SHARDING

Deux évolutions majeures sont apparues dans la gestion des tables distantes à travers l'API *Foreign Data Wrapper*, portées dans l'extension `postgres_fdw`. Nous verrons que l'architecture distribuée, dites *sharding*, devient alors possible.

1.4.3.1 Support du TRUNCATE sur les tables distantes

- Nouvelle routine dans l'API Foreign Data Wrapper pour la commande `TRUNCATE`
- Supportée pour les serveurs distants PostgreSQL avec l'extension `postgres_fdw`
- Valable pour les partitions distantes d'une table partitionnée
- Option `truncatable` activée par défaut

La commande `TRUNCATE` dispose à présent d'un `callback` dans l'API *Foreign Data Wrapper*. L'extension `postgres_fdw` propose une implémentation pour les serveurs distants PostgreSQL avec l'ensemble des options existantes pour cette commande :

- `CASCADE` : supprime automatiquement les lignes des tables disposant d'une contrainte de clé étrangère sur la table concernée ;
- `RESTRICT` : refuse le vidage de la table si l'une de ses colonnes est impliquée dans une contrainte de clé étrangère (comportement par défaut) ;
- `RESTART IDENTITY` : redémarre les séquences rattachés aux colonnes d'identité de la table tronquée ;
- `CONTINUE IDENTITY` : ne change pas la valeur des séquences (comportement par défaut).

L'usage du `TRUNCATE` apporte un gain de performance par rapport à la commande `DELETE`, qui était jusqu'à présent la seule alternative pour vider les tables distantes.

Si dans une table partitionnée il existe des partitions distantes, la commande `TRUNCATE` est également propagée vers les différents serveurs distants.

Ce nouveau comportement peut être désactivé par l'option `truncatable` au niveau de la table ou du serveur distant.

```
ALTER SERVER srv1 OPTIONS (ADD truncatable 'false');
ALTER FOREIGN TABLE tbl1 OPTIONS (ADD truncatable 'false');

TRUNCATE tbl1;

ERROR:  foreign table "tbl1" does not allow truncates
```

1.4.3.2 *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);
```

1.5 DÉVELOPPEMENT ET SYNTAXE SQL

1.5.1 FONCTION STRING_TO_TABLE

- Nouvelle fonction pour subdiviser une chaîne de caractère et renvoyer le résultat dans une table :

```
SELECT string_to_table('une chaine à ignorer', ' ', 'ignorer');

string_to_table
-----------------
 une
 chaine
 à
 ¤
```

- Alternative plus performante à `regexp_split_to_table()` et `unnest(string_to_array())`.

Une nouvelle fonction a été créée pour subdiviser une chaîne de caractères et renvoyer le résultat dans une table :

```
\df string_to_table
                                List of functions
   Schema   |      Name       | Result data type | Argument data types | Type
------------+-----------------+------------------+---------------------+------
 pg_catalog | string_to_table | SETOF text       | text, text          | func
 pg_catalog | string_to_table | SETOF text       | text, text, text    | func
(2 rows)
```

Exemple d'utilisation :

```
\pset null '¤'
SELECT string_to_table('une chaine à ignorer', ' ', 'ignorer');

 string_to_table
-----------------
 une
 chaine
 à
 ¤
(4 rows)
```

Dans les versions précédentes, ce genre d'opération était déjà possible avec les fonctions `unnest(string_to_array())` et `regexp_split_to_table()`. Cette nouvelle fonction a l'avantage d'être plus performante car plus spécialisée.

1.5.2 NOUVELLE SYNTAXE OR REPLACE POUR LA MODIFICATION D'UN TRIGGER

```
CREATE OR REPLACE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

- Ne fonctionne pas pour les `CONSTRAINT TRIGGER`
- Ne pas lancer dans une transaction qui a modifié la table du trigger

La syntaxe `OR REPLACE` est désormais disponible dans l'ordre de création des triggers. C'est une extension du standard SQL qui permet de mettre à jour la définition d'un trigger sans devoir le supprimer au préalable.

Cette fonctionnalité n'est pas disponible pour les triggers de type `CONSTRAINT TRIGGER`[8] et provoque le message d'erreur suivant.

```
ERROR:  CREATE OR REPLACE CONSTRAINT TRIGGER is not supported
```

De plus, si des instructions ont mis à jour la table sur laquelle le trigger est placé, il est déconseillé d'exécuter le `CREATE OR REPLACE` dans la même transaction. En effet, le résultat pourrait être différent de ce que vous anticipez.

1.5.3 SUPPORT DES PARAMÈTRES OUT DANS LES PROCÉDURES

```
CREATE PROCEDURE assign(IN a int, OUT b int)
```

- paramètre initialisé à NULL en début de procédure

La version 11 a introduit les procédures dans PostgreSQL. Jusqu'à maintenant, le mode des paramètres pouvait être : `IN`, `INOUT` ou `VARIADIC`. Il est désormais possible de déclarer des paramètres avec le mode `OUT`.

Exemple :

```
CREATE PROCEDURE assign(IN a int, OUT b int)
  LANGUAGE plpgsql
AS $$
BEGIN
  -- assigner une valeur à b si a = 10
  IF a = 10 THEN
    b := a;
  END IF;
END;
$$;
-- CREATE PROCEDURE
```

[8] https://docs.postgresql.fr/14/sql-createtrigger.html

Comme le montre l'exemple ci-dessous, la variable spécifiée comme paramètre de sortie est initialisée à NULL en début de procédure.

```
DO $$
DECLARE _a int; _b int;
BEGIN
  _a := 10;
  CALL assign(_a, _b);
  RAISE NOTICE '_a: %, _b: %', _a, _b;

  _a := 100;
  CALL assign(_a, _b);
  RAISE NOTICE '_a: %, _b: %', _a, _b;
END
$$;
-- NOTICE:  _a: 10, _b: 10
-- NOTICE:  _a: 100, _b: <NULL>
-- DO
```

1.5.4 PL/PGSQL : ASSIGNATION POUR LES TYPES COMPLEXES

- Évolution du *parser* de requêtes
- Supporte l'assignation de valeurs pour les types complexes en PL/pgSQL

```
a[2:3] := array[3,4];     -- slice de tableaux  int[]
a[1].i := 2;              -- champ de record
h['a'] := 'b';            -- hstore
```

- et plus performants !

Le langage PL/pgSQL bénéficie d'une petite évolution dans les règles d'assignation de valeurs pour les types complexes, tels que les lignes (*records*) ou les tableaux (*arrays*). Le *parser* du langage est à présent capable de reconnaître des expressions pour les assigner avec l'opérateur := dans la variable de destination, sans contournement du langage.

Ci-dessous, une liste non-exhaustive des possibilités :

```
-- assigner les valeurs d'une portion d'un tableau
-- où "a" est de type int[]
a[2:3] := array[3,4];

-- assigner la valeur d'un champ de record personnalisé
-- où "a" est de type complex[]
CREATE TYPE complex AS (r float8, i float8);
a[1].r := 1;
a[1].i := 2;
```

```
-- assigner la valeur d'une clé hstore
-- où "h" est de type hstore
CREATE EXTENSION hstore;
h['a'] := 'b';
```

D'autres bénéfices sont obtenus avec cette évolution dans l'analyse de la syntaxe de ces types d'assignations. Tom Lane, à l'origine de ce patch, annonce[9] un gain de performance substantiel ainsi qu'une meilleure lisibilité des erreurs pouvant survenir à l'assignation.

Exemple en version 13 et inférieures :

```
DO $$ DECLARE x int := 1/0; BEGIN END $$ ;
-- ERROR:  division by zero
-- CONTEXT:  SQL statement "SELECT 1/0"
```

Exemple en version 14 :

```
DO $$ DECLARE x int := 1/0; BEGIN END $$ ;
-- ERROR:  division by zero
-- CONTEXT:  SQL expression "1/0"
```

1.5.5 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
```

[9] https://www.postgresql.org/message-id/flat/4165684.1607707277@sss.pgh.pa.us

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;
```

1.5.6 NOUVEAUX TYPES MULTIRANGE ET NOUVELLES FONCTIONS D'AGRÉGATS

- Nouveaux types `multirange`
  - permettent de créer des ensembles d'intervalles disjoints
  - fonctionnalités similaires aux types d'intervalles simples
- Nouvelles fonctions pour agréger des intervalles :
  - `range_agg()`
  - `range_intersect_agg()`
- Indexable avec `btree`, `gist` et `hash`

PostgreSQL dispose de types intervalles de valeurs depuis la version 9.2 de PostgreSQL. Ils permettent de stocker et manipuler des intervalles pour des données de type : `int4`, `int8`, `numeric`, `timestamp`, `timestamp with timezone` et `date`. Différents traitements peuvent être effectués sur les données, comme :

- différence, intersection, union d'intervalles ;
- comparaison d'intervalles ;
- tests sur les bornes, l'inclusion etc.

Exemple :

```
SELECT x,
       lower(x) as "borne inf",
       upper(x) as "borne sup",
       x @> 2 as "contient 2",
       x @> 4 as "contient 4",
       x * '[1,2]'::int4range AS "intersection avec [1,2]"
  FROM (VALUES ('[1,4)'::int4range) ) AS F(x) \gx
```

```
-[ RECORD 1 ]-----------+------
x                       | [1,4)
borne inf               | 1
borne sup               | 4
contient 2              | t
contient 4              | f
intersection avec [1,2] | [1,3)
```

La version 14 voit une nouvelle avancée sur ce thème : les types `multirange`. Ces nouveaux types étendent les types d'intervalles existant pour stocker plusieurs intervalles disjoints ensemble.

Exemple :

```
SELECT '{ [1,2), (2,3]}'::int4multirange \gx
```

```
-[ RECORD 1 ]--+--------
int4multirange | {[1,2),[3,4)}
```

```
SELECT '{[1,5], [2,6]}'::int4multirange \gx
```

```
-[ RECORD 1 ]--+--------
int4multirange | {[1,7)}
```

Il est possible d'effectuer des opérations similaires à celles permises sur les intervalles simples sur ces nouveaux types :

```
SELECT x,
       lower(x) as "borne inf",
       upper(x) as "borne sup",
       x @> 2 as "contient 2",
       x @> 4 as "contient 4",
       x * '{[1,2],[6,7]}'::int4multirange
         AS "intersection avec {[1,2], [6,7]}"
  FROM (VALUES ('{[1,4), [5,8)}'::int4multirange) )
    AS F(x) \gx
```

```
-[ RECORD 1 ]--------------------+--------------
x                                | {[1,4),[5,8)}
borne inf                        | 1
borne sup                        | 8
contient 2                       | t
contient 4                       | f
intersection avec {[1,2], [6,7]} | {[1,3),[6,8)}
```

Ils permettent également de produire des résultats qui n'étaient pas possibles avec des intervalles simples. Comme pour cette soustraction d'intervalles :

```
SELECT '[1,5]'::int4range - '[2,3)'::int4range AS RESULT;
-- ERROR:  result of range difference would not be contiguous

SELECT '{[1,5]}'::int4multirange - '{[2,3)}'::int4multirange AS result;
```

```
    result
---------------
 {[1,2),[3,6)}
```

De nouvelles fonctions sont également disponibles pour agréger les intervalles. Il s'agit de `range_agg()` et `range_intersect_agg()`.

Voici un exemple d'utilisation :

```
CREATE TABLE planning (
  classe text,
  salle text,
  plage_horaire tsrange,
  matiere text
);

INSERT INTO planning(classe, salle, plage_horaire, matiere)
VALUES
  (1, 'Salle 1', '[2021-07-19  9:00, 2021-07-19 10:00)'::tsrange, 'math'),
  (1, NULL,      '[2021-07-19 10:00, 2021-07-19 10:15)'::tsrange, 'recreation'),
  (1, 'Salle 2', '[2021-07-19 10:15, 2021-07-19 12:15)'::tsrange, 'français'),
  (1, NULL,      '[2021-07-19 12:15, 2021-07-19 14:15)'::tsrange, 'repas / recreation'),
  (1, 'Salle 2', '[2021-07-19 14:15, 2021-07-19 15:15)'::tsrange, 'anglais'),
  (2, 'Salle 1', '[2021-07-19  8:00, 2021-07-19 10:00)'::tsrange, 'physique'),
  (2, NULL,      '[2021-07-19 10:00, 2021-07-19 10:15)'::tsrange, 'recreation'),
  (2, 'Salle 1', '[2021-07-19 10:15, 2021-07-19 12:45)'::tsrange, 'technologie'),
  (2, NULL,      '[2021-07-19 12:45, 2021-07-19 14:15)'::tsrange, 'repas / recreation'),
  (2, 'Salle 1', '[2021-07-19 14:15, 2021-07-19 16:15)'::tsrange, 'math'),
  (3, 'Salle 2', '[2021-07-19 14:15, 2021-07-19 15:15)'::tsrange, 'allemand')
;
```

Planning par classe et salle :

```
SELECT classe, salle, range_agg(plage_horaire) AS plages_horaires
  FROM planning
 WHERE salle IS NOT NULL
 GROUP BY classe, salle
 ORDER BY classe, salle;
```

```
 classe |  salle  |                plages_horaires
--------+---------+-------------------------------------------------
 1      | Salle 1 | {["2021-07-19 09:00:00","2021-07-19 10:00:00")}
 1      | Salle 2 | {["2021-07-19 10:15:00","2021-07-19 12:15:00"),
        |         |  ["2021-07-19 14:15:00","2021-07-19 15:15:00")}
 2      | Salle 1 | {["2021-07-19 08:00:00","2021-07-19 10:00:00"),
        |         |  ["2021-07-19 10:15:00","2021-07-19 12:45:00"),
        |         |  ["2021-07-19 14:15:00","2021-07-19 16:15:00")}
 3      | Salle 2 | {["2021-07-19 14:15:00","2021-07-19 15:15:00")}
(4 rows)
```

Collisions dans l'utilisation des salles :

```
WITH planning_par_classe_et_salle (classe, salle, plages_horaires) AS (
  SELECT classe, salle, range_agg(plage_horaire) AS plages_horaires
    FROM planning
   WHERE salle IS NOT NULL
```

```
   GROUP BY classe, salle
)
SELECT salle,
       range_intersect_agg(plages_horaires) as plages_horaires,
       array_agg(classe) as classes
  FROM planning_par_classe_et_salle
 GROUP BY salle HAVING count(*) > 1
 ORDER BY salle;

  salle  |                 plages_horaires                 | classes
---------+-------------------------------------------------+---------
 Salle 1 | {["2021-07-19 09:00:00","2021-07-19 10:00:00")} | {1,2}
 Salle 2 | {["2021-07-19 14:15:00","2021-07-19 15:15:00")} | {3,1}
(2 rows)
```

Voici la liste des index qui supportent ces nouveaux types et les opérations indexables :

```
SELECT a.amname, of.opfname, t1.typname as lefttype,
       t2.typname as righttyp, o.oprname, o.oprcode
  FROM pg_amop ao
 INNER JOIN pg_am a ON ao.amopmethod = a.oid
 INNER JOIN pg_opfamily of ON ao.amopfamily = of.oid
 INNER JOIN pg_type t1 ON ao.amoplefttype = t1.oid
 INNER JOIN pg_type t2 ON ao.amoplefttype = t2.oid
 INNER JOIN pg_operator o ON ao.amopopr = o.oid
 WHERE of.opfname LIKE '%multirange%';

amname |    opfname      |   lefttype    |   righttyp    |oprname|          oprcode
-------+-----------------+---------------+---------------+-------+---------------------------------
gist   | multirange_ops | anymultirange | anymultirange | <<    |multirange_before_multirange
gist   | multirange_ops | anymultirange | anymultirange | <<    |multirange_before_range
gist   | multirange_ops | anymultirange | anymultirange | &<    |multirange_overleft_multirange
gist   | multirange_ops | anymultirange | anymultirange | &<    |multirange_overleft_range
gist   | multirange_ops | anymultirange | anymultirange | &&    |multirange_overlaps_multirange
gist   | multirange_ops | anymultirange | anymultirange | &&    |multirange_overlaps_range
gist   | multirange_ops | anymultirange | anymultirange | &>    |multirange_overright_multirange
gist   | multirange_ops | anymultirange | anymultirange | &>    |multirange_overright_range
gist   | multirange_ops | anymultirange | anymultirange | >>    |multirange_after_multirange
gist   | multirange_ops | anymultirange | anymultirange | >>    |multirange_after_range
gist   | multirange_ops | anymultirange | anymultirange | -|-   |multirange_adjacent_multirange
gist   | multirange_ops | anymultirange | anymultirange | -|-   |multirange_adjacent_range
gist   | multirange_ops | anymultirange | anymultirange | @>    |multirange_contains_multirange
gist   | multirange_ops | anymultirange | anymultirange | @>    |multirange_contains_range
gist   | multirange_ops | anymultirange | anymultirange | <@    |multirange_contained_by_multirang
gist   | multirange_ops | anymultirange | anymultirange | <@    |multirange_contained_by_range
gist   | multirange_ops | anymultirange | anymultirange | @>    |multirange_contains_elem
gist   | multirange_ops | anymultirange | anymultirange | =     |multirange_eq
btree  | multirange_ops | anymultirange | anymultirange | <     |multirange_lt
```

```
btree  | multirange_ops | anymultirange | anymultirange | <=    |multirange_le
btree  | multirange_ops | anymultirange | anymultirange | =     |multirange_eq
btree  | multirange_ops | anymultirange | anymultirange | >=    |multirange_ge
btree  | multirange_ops | anymultirange | anymultirange | >     |multirange_gt
hash   | multirange_ops | anymultirange | anymultirange | =     |multirange_eq
```

La lecture du catalogue nous montre que les opérations simples (exp : =, >, <) peuvent être indexées avec un `btree`. En revanche, pour les opérations plus complexes (appartenance, proximité ...), il faut utiliser un index `gist`.

La documentation détaille l'ensemble des opérateurs[10] , fonctions[11] , et agrégats[12] disponibles pour ce nouveau type[13] .

1.5.7 GROUP BY DISTINCT

- Dédoublonnage des résultats d'agrégations multiples produit par un `GROUP BY`
- Utile avec `ROLLUP` ou `CUBE`

Lorsque l'on combine plusieurs méthodes d'agrégation comme `ROLLUP` ou `CUBE`, il est fréquent de se retrouver avec des doublons. Le standard SQL prévoit de dédupliquer le résultat de ce genre de requête avec la syntaxe `GROUP BY DISTINCT`. Elle a été implémentée dans PostgreSQL.

Voici un exemple :

```
cat <<_EOF_ | psql
CREATE TABLE entreprise(nom text, departement int, ville text, creation date, montant int);
COPY entreprise FROM STDIN WITH DELIMITER ',' CSV;
entreprise1,44,Nantes,20210506,1000
entreprise2,44,Nantes,20200506,200
entreprise3,29,Brest,20200605,3000
entreprise4,29,Brest,20200406,4000
\.
_EOF_
```

En exécutant cette requête, on voit que certaines lignes sont en double :

```
SELECT row_number() OVER(), departement, ville,
       extract(YEAR FROM  creation) as year,
       avg(montant)::int as montant
  FROM entreprise
```

[10] https://docs.postgresql.fr/14/functions-range.html#RANGE-OPERATORS-TABLE
[11] https://docs.postgresql.fr/14/functions-range.html#RANGE-FUNCTIONS-TABLE
[12] https://docs.postgresql.fr/14/functions-aggregate.html
[13] https://docs.postgresql.fr/14/rangetypes.html#RANGETYPES-BUILTIN

```
 GROUP BY rollup(departement, ville),
          rollup(departement, year);

 row_number | departement |  ville  | year | montant
------------+-------------+---------+------+---------
          1 |           ¤ | ¤       |    ¤ |    2050
          2 |          44 | Nantes  | 2021 |    1000
          3 |          44 | Nantes  | 2020 |     200
          4 |          29 | Brest   | 2020 |    3500
          5 |          29 | Brest   |    ¤ |    3500
          6 |          44 | Nantes  |    ¤ |     600
          7 |          29 | Brest   |    ¤ |    3500 << DOUBLON DE 5
          8 |          44 | Nantes  |    ¤ |     600 << DOUBLON DE 6
          9 |          44 | ¤       |    ¤ |     600
         10 |          29 | ¤       |    ¤ |    3500
         11 |          44 | ¤       |    ¤ |     600 << DOUBLON DE 9
         12 |          29 | ¤       |    ¤ |    3500 << DOUBLON DE 10
         13 |          44 | ¤       |    ¤ |     600 << DOUBLON DE 9
         14 |          29 | ¤       |    ¤ |    3500 << DOUBLON DE 10
         15 |          29 | ¤       | 2020 |    3500
         16 |          44 | ¤       | 2020 |     200
         17 |          44 | ¤       | 2021 |    1000
         18 |          29 | ¤       | 2020 |    3500 << DOUBLON DE 15
         19 |          44 | ¤       | 2020 |     200 << DOUBLON DE 16
         20 |          44 | ¤       | 2021 |    1000 << DOUBLON DE 17
(20 rows)
```

L'utilisation de `GROUP BY DISTINCT` permet de régler ce problème sans étape supplémentaire :

```
SELECT departement, ville, extract(YEAR FROM  creation) as year,
       avg(montant)::int as montant
  FROM entreprise
 GROUP BY DISTINCT rollup(departement, ville),
                   rollup(departement, year);

 departement |  ville  | year | montant
-------------+---------+------+---------
           ¤ | ¤       |    ¤ |    2050
          44 |  Nantes | 2021 |    1000
          44 |  Nantes | 2020 |     200
          29 |  Brest  | 2020 |    3500
          29 |  Brest  |    ¤ |    3500
          44 |  Nantes |    ¤ |     600
          44 | ¤       |    ¤ |     600
          29 | ¤       |    ¤ |    3500
          29 | ¤       | 2020 |    3500
          44 | ¤       | 2020 |     200
```

```
          44 | ¤        | 2021 |    1000
(11 rows)
```

1.5.8 CORPS DE ROUTINES RESPECTANT LE STANDARD SQL

- Nouvelles syntaxes :
  - `RETURN`
  - `BEGIN ATOMIC .. END;`
- Limitées au langage SQL
- Impossible d'utiliser des paramètres polymorphiques (`anyelement`, etc.)
- Dépendances avec les objets utilisés (`DROP CASCADE`)

La version 14 de PostgreSQL permet de créer des procédures et fonctions avec un corps qui respecte le standart SQL. Cette nouvelle fonctionnalité se limite aux fonctions écrites avec le langage SQL.

Deux syntaxes sont disponibles :

```
-- RETURN
CREATE FUNCTION add(a integer, b integer) RETURNS integer
  LANGUAGE SQL
  RETURN a + b;

-- BEGIN ATOMIC .. END
CREATE PROCEDURE insert_data(a integer, b integer)
  LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
END;
```

Ce type de déclaration ne permet pas d'utiliser les types polymorphiques[14] :

```
CREATE OR REPLACE FUNCTION display_type(a anyelement) RETURNS text
LANGUAGE SQL
RETURN 'The input type is ' || pg_typeof(a);
```

```
ERROR:  SQL function with unquoted function body cannot have polymorphic arguments
```

Pour cela, il faut continuer d'utiliser l'ancienne syntaxe avec l'encadrement du corps de la fonction par des doubles `$$` :

```
CREATE OR REPLACE FUNCTION display_type(a anyelement) RETURNS text
LANGuAGE SQL
AS $$ SELECT 'The input type is ' || pg_typeof(a); $$
-- CREATE FUNCTION
```

[14] https://docs.postgresql.fr/14/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC

Cette différence de comportement s'explique par le fait que la nouvelle syntaxe est analysée (*parsed*) lors de la définition de la routine, alors que l'ancienne l'était à chaque exécution.

La nouvelle approche permet de définir les dépendances entre la routine et les objets qu'elle utilise. Il en résulte une suppression des routines lors de l'exécution d'un `DROP CASCADE` sur les objets en question.

Exemple :

```
CREATE TABLE tbl1(i int);
CREATE TABLE tbl2(i int);

-- Procédure avec la nouvelle syntaxe
CREATE OR REPLACE PROCEDURE insert_data_new(a integer, b integer)
  LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl1 VALUES (a);
  INSERT INTO tbl2 VALUES (a);
END;

-- Procédure avec l'ancienne syntaxe
CREATE OR REPLACE PROCEDURE insert_data_old(a integer, b integer)
  LANGUAGE SQL
AS $$
  INSERT INTO tbl1 VALUES (a);
  INSERT INTO tbl2 VALUES (a);
$$;
```

Lors de la création, seule la procédure utilisant la nouvelle méthode est supprimée.

```
DROP TABLE tbl1, tbl2 CASCADE;
-- NOTICE:  drop cascades to function insert_data_new(integer,integer)
-- DROP TABLE
```

Les deux méthodes renvoient une erreur si on utilise des objets qui n'existent pas lors de la création de la routine :

```
ERROR:  relation "tbl1" does not exist
LINE 4:   INSERT INTO tbl1 VALUES (a);
```

1.5.9 NOUVELLES CLAUSES SEARCH ET CYCLE

- Génération d'une colonne de tri pour les requêtes récursives :

```
[ SEARCH { BREADTH | DEPTH } FIRST BY column_name [, ...]
    SET search_seq_col_name ]
```

- Protection contre les cycles :

```
[ CYCLE column_name [, ...]
    SET cycle_mark_col_name
    [ TO cycle_mark_value DEFAULT cycle_mark_default ]
    USING cycle_path_col_name ]
```

PostgreSQL permet de créer des requêtes récursives grâce à la clause `WITH RECURSIVE`. Ce genre de requêtes permet de remonter une arborescence d'enregistrements liés par des colonnes de type `id` et `parent_id`.

Dans ce genre de requête, il est courant de vouloir :

- ordonner les données en fonction de leur profondeur ;
- afficher le chemin parcouru ou la profondeur de l'enregistrement dans l'arborescence ;
- détecter l'apparition d'un cycle, une séquence d'enregistrement provoquant une boucle.

La norme SQL prévoit différentes syntaxes pour réaliser ce genre de tâches. Elles sont désormais implémentées dans PostgreSQL.

Création d'un jeu d'essais :

```
CREATE TABLE tree(id int, parent_id int, name text);
ALTER TABLE tree ADD PRIMARY KEY (id);
INSERT INTO tree(id, parent_id, name)
VALUES (1, NULL, 'Albert'),
       (2, 1, 'Bob'),
       (3, 1, 'Barbara'),
       (4, 1, 'Britney'),
       (5, 3, 'Clara'),
       (6, 3, 'Clement'),
       (7, 2, 'Craig'),
       (8, 5, 'Debby'),
       (9, 5, 'Dave'),
       (10, 9, 'Edwin');
```

Il est fréquent de vouloir récupérer la profondeur d'un enregistrement dans l'arbre que l'on reconstitue afin d'ordonner les données. Voici un exemple qui récupère la ou les personnes avec la plus grande profondeur dans l'arborescence.

```
--- ajout d'un champ profondeur (depth)
WITH RECURSIVE mtree(id, name, depth) AS (
   -- initialisation de la profondeur à 0 pour le point de départ
```

```
   SELECT id, name, 0
     FROM tree
    WHERE id = 1

   UNION ALL

   -- Incrémenter la profondeur de 1
   SELECT t.id, t.name, m.depth + 1
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
)
SELECT * FROM mtree ORDER BY depth DESC LIMIT 1;
```

```
 id | name  | depth
----+-------+-------
 10 | Edwin |     4
(1 row)
```

En version 14, la syntaxe suivante permet de récupérer des informations similaires :

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( query )
  [ SEARCH BREADTH FIRST BY column_name [, ...] SET search_seq_col_name ];

query: ( select | values | insert | update | delete )
```

Exemple :

```
WITH RECURSIVE mtree(id, name) AS (
   SELECT id, name
     FROM tree
    WHERE id = 1

   UNION ALL

   SELECT t.id, t.name
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
) SEARCH BREADTH FIRST BY name SET morder
SELECT * FROM mtree ORDER BY morder DESC;
```

```
 id |  name   |   morder
----+---------+-------------
 10 | Edwin   | (4,Edwin)
  8 | Debby   | (3,Debby)
  9 | Dave    | (3,Dave)
  7 | Craig   | (2,Craig)
  6 | Clement | (2,Clement)
  5 | Clara   | (2,Clara)
  4 | Britney | (1,Britney)
```

```
  2 | Bob     | (1,Bob)
  3 | Barbara | (1,Barbara)
  1 | Albert  | (0,Albert)
(10 rows)
```

En appliquant la clause `LIMIT 1`. On obtient donc le même résultat que précédemment.

Ce genre de requête a pour inconvénient de risquer de boucler si un cycle est introduit dans le jeu de données. Il faut donc se prémunir contre ce genre de problème.

```
UPDATE tree SET parent_id = 10 WHERE id = 1;
-- UPDATE 1

-- ajout de deux champs:
-- * un booleen qui permet de détecter les cycles (is_cycle)
-- * un tableau qui contient le chemin parcouru (path)
WITH RECURSIVE mtree(id, name, depth, is_cycle, path) AS (
   -- initialisations
   SELECT id, name, 0,
          false,          -- initialement, on ne boucle pas
          ARRAY[ROW(id)]  -- le premier élément du chemin
     FROM tree
    WHERE id = 1

   UNION ALL

   SELECT t.id, t.name, m.depth + 1,
          ROW(t.id) = ANY(m.path), -- déja traitré ?
          m.path || ROW(t.id)      -- ajouter le tuple au chemin
   FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id

   -- stopper l'itération si on détecte un cycle
   WHERE NOT m.is_cycle
)
SELECT * FROM mtree ORDER BY depth DESC LIMIT 1;

 id |  name  | depth | is_cycle |            path
----+--------+-------+----------+----------------------------
  1 | Albert |     5 | t        | {(1),(3),(5),(9),(10),(1)}
(1 row)
```

Le même résultat peut être obtenu en utilisant la clause CYCLE :

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( query )
  [ CYCLE column_name [, ...] SET cycle_mark_col_name
                              [ TO cycle_mark_value DEFAULT cycle_mark_default ]
                              USING cycle_path_col_name ]
```

```
query: ( select | values | insert | update | delete )
```

Voici un exemple :

```
WITH RECURSIVE mtree(id, name) AS (
   SELECT id, name
     FROM tree
    WHERE id = 1
   UNION ALL
   SELECT t.id, t.name
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
) SEARCH BREADTH FIRST BY name SET morder
  CYCLE id SET is_cycle USING path
SELECT * FROM mtree ORDER BY morder DESC LIMIT 1;
```

```
 id |  name  |   morder   | is_cycle |            path
----+--------+------------+----------+----------------------------
  1 | Albert | (5,Albert) | t        | {(1),(3),(5),(9),(10),(1)}
(1 row)
```

Il est également possible de construire un tableau avec le contenu de la table et de trier à partir de ce contenu grâce à la syntaxe suivante :

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( query )
  [ SEARCH DEPTH FIRST BY column_name [, ...] SET search_seq_col_name ];

query: ( select | values | insert | update | delete )
```

Comme vous pouvez le voir dans l'exemple ci-dessous, il est possible d'utiliser la clause CYCLE avec cette syntaxe aussi :

```
WITH RECURSIVE mtree(id, name) AS (
   SELECT id, name
     FROM tree
    WHERE id = 1

   UNION ALL

   SELECT t.id, t.name
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
) SEARCH DEPTH FIRST BY name SET morder
  CYCLE id SET is_cycle USING path
SELECT * FROM mtree WHERE not is_cycle ORDER BY morder DESC;
```

```
 id |  name   |                   morder                    | is_cycle |          path
----+---------+---------------------------------------------+----------+------------------------
  4 | Britney | {(Albert),(Britney)}                        | f        | {(1),(4)}
  7 | Craig   | {(Albert),(Bob),(Craig)}                    | f        | {(1),(2),(7)}
```

```
  2 | Bob     | {(Albert),(Bob)}                           | f        | {(1),(2)}
  6 | Clement | {(Albert),(Barbara),(Clement)}             | f        | {(1),(3),(6)}
  8 | Debby   | {(Albert),(Barbara),(Clara),(Debby)}       | f        | {(1),(3),(5),(8)}
 10 | Edwin   | {(Albert),(Barbara),(Clara),(Dave),(Edwin)} | f        | {(1),(3),(5),(9),(10)}
  9 | Dave    | {(Albert),(Barbara),(Clara),(Dave)}        | f        | {(1),(3),(5),(9)}
  5 | Clara   | {(Albert),(Barbara),(Clara)}               | f        | {(1),(3),(5)}
  3 | Barbara | {(Albert),(Barbara)}                       | f        | {(1),(3)}
  1 | Albert  | {(Albert)}                                 | f        | {(1)}
(10 rows)
```

L'implémentation actuelle[15] ne permet pas d'interagir avec les valeurs ramenées par les clauses `{ BREADTH | DEPTH } FIRST` car leur fonction est de produire une colonne qui facilite le tri des résultats.

```
WITH RECURSIVE mtree(id, name) AS (
   SELECT id, name
     FROM tree
    WHERE id = 1
   UNION ALL
   SELECT t.id, t.name
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
) SEARCH BREADTH FIRST BY name SET morder
  CYCLE id SET is_cycle USING path
SELECT id, name, (morder).* FROM mtree ORDER BY morder DESC LIMIT 1;
```

```
ERROR:  CTE mtree does not have attribute 3
```

Il est cependant possible d'y accéder en transformant le champ en objet JSON.

```
 WITH RECURSIVE mtree(id, name) AS (
   SELECT id, name
     FROM tree
    WHERE id = 1
   UNION ALL
   SELECT t.id, t.name
     FROM tree AS t
          INNER JOIN mtree AS m ON t.parent_id = m.id
) SEARCH BREADTH FIRST BY name SET morder
  CYCLE id SET is_cycle USING path
SELECT id, name, row_to_json(morder) -> '*DEPTH*' AS depth
  FROM mtree ORDER BY morder DESC LIMIT 1;
```

```
 id |  name  | depth
----+--------+-------
  1 | Albert | 5
(1 row)
```

[15] https://www.postgresql.org/message-id/4a068167-37ed-3d6c-5ec5-c9b03cae84e6%40enterprisedb.com

1.5.10 NOUVELLE FONCTION DATE_BIN

- Nouvelle fonction pour répartir des timestamps dans des intervalles (*buckets*)
- `date_bin`
  - `interval` : taille des *buckets* (unités `month` et `year` interdites)
  - `timestamptz` : valeur en entrée à traiter
  - `timestamptz` : timestamp correspondant au début du premier *bucket*

La nouvelle fonction `date_bin` permet de placer un timestamp fourni en entrée (second paramètre) dans un intervalle aussi appellée *bucket*.

Documentation : FUNCTIONS-DATETIME-BIN[16]

Les valeurs produites correspondent au timestamp en début de l'intervalle et peuvent par exemple être utilisées pour calculer des statistiques en regroupant les données par plages de 15 minutes.

La valeur mise en second paramètre de la fonction est placée dans un *bucket* en se basant sur :

- Un timestamp de début (troisième paramètre).
- Une taille définie sous forme d'intervalle (premier paramètre).
  L'unité utilisée pour définir la taille du *bucket* peut être définie en secondes, minutes, heures, jours ou semaines.

La fonction existe pour des timestamp avec et sans fuseau horaire :

```
\df date_bin

List of functions
-[ RECORD 1 ]-------+---------------------------------------------------------------------
Schema              | pg_catalog
Name                | date_bin
Result data type    | timestamp with time zone
Argument data types | interval, timestamp with time zone, timestamp with time zone
Type                | func
-[ RECORD 2 ]-------+---------------------------------------------------------------------
Schema              | pg_catalog
Name                | date_bin
Result data type    | timestamp without time zone
Argument data types | interval, timestamp without time zone, timestamp without time zone
Type                | func
```

Voici un exemple de cette fonction en action :

[16] https://docs.postgresql.fr/14/functions-datetime.html#FUNCTIONS-DATETIME-BIN

```
-- Génération des données
CREATE TABLE sonde(t timestamp with time zone, id_sonde int, mesure int);
INSERT INTO sonde(t, id_sonde, mesure)
SELECT '2021-06-01 00:00:00'::timestamp with time zone + INTERVAL '1s' * x,
       1,
       sin(x*3.14/86401)*30
  FROM generate_series(0, 60*60*24) AS F(x);

-- création de buckets de 1h30 commençant à minuit le premier juin
SELECT date_bin('1 hour 30 minutes', t, '2021-06-01 00:00:00'::timestamp with time zone),
       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
 2021-06-01 06:00:00+02 |        1 |    23.1514814814814815
 2021-06-01 07:30:00+02 |        1 |    26.3951851851851852
 2021-06-01 09:00:00+02 |        1 |    28.6138888888888889
 2021-06-01 10:30:00+02 |        1 |    29.9274074074074074
 2021-06-01 12:00:00+02 |        1 |    29.9359259259259259
 2021-06-01 13:30:00+02 |        1 |    28.6224074074074074
 2021-06-01 15:00:00+02 |        1 |    26.4207407407407407
 2021-06-01 16:30:00+02 |        1 |    23.1851851851851852
 2021-06-01 18:00:00+02 |        1 |    19.0346296296296296
 2021-06-01 19:30:00+02 |        1 |    14.1690740740740741
 2021-06-01 21:00:00+02 |        1 |     8.7175925925925926
 2021-06-01 22:30:00+02 |        1 |     2.9829629629629630
 2021-06-02 00:00:00+02 |        1 | 0.00000000000000000000
(17 rows)
```

La date de début utilisée pour la création des *buckets* ne doit pas nécessairement coïncider avec le timestamp le plus ancien présent dans la table :

```
SELECT date_bin('1 hour 30 minutes', t, '2021-06-01 00:11:00'::timestamp with time zone),
       id_sonde,
       avg(mesure)
FROM sonde GROUP BY 1, 2 ORDER BY 1 ASC;
```

```
        date_bin        | id_sonde |          avg
------------------------+----------+-----------------------
 2021-05-31 22:41:00+02 |        1 | 0.30454545454545454545
 2021-06-01 00:11:00+02 |        1 |     3.6372222222222222
 2021-06-01 01:41:00+02 |        1 |     9.3907407407407407
 2021-06-01 03:11:00+02 |        1 |    14.7375925925925926
```

```
 2021-06-01 04:41:00+02 |         1 |    19.5405555555555556
 2021-06-01 06:11:00+02 |         1 |    23.5896296296296296
 2021-06-01 07:41:00+02 |         1 |    26.7618518518518519
 2021-06-01 09:11:00+02 |         1 |    28.7857407407407407
 2021-06-01 10:41:00+02 |         1 |    30.0000000000000000
 2021-06-01 12:11:00+02 |         1 |    29.8137037037037037
 2021-06-01 13:41:00+02 |         1 |    28.4772222222222222
 2021-06-01 15:11:00+02 |         1 |    26.0770370370370370
 2021-06-01 16:41:00+02 |         1 |    22.6962962962962963
 2021-06-01 18:11:00+02 |         1 |    18.4644444444444444
 2021-06-01 19:41:00+02 |         1 |    13.5207407407407407
 2021-06-01 21:11:00+02 |         1 |     8.0494444444444444
 2021-06-01 22:41:00+02 |         1 |     2.6230753005695001
(17 rows)
```

Comme dit précédemment, il n'est pas possible d'utiliser une taille de *bucket* définie en mois ou années. Il est cependant possible de spécifier des tailles de *bucket* supérieures ou égales à un mois avec les autres unités :

```
SELECT date_bin('1 year', '2021-06-01 10:05:10', '2021-06-01');
-- ERROR:  timestamps cannot be binned into intervals containing months or years

SELECT date_bin('1 month', '2021-06-01 10:05:10', '2021-06-01');
-- ERROR:  timestamps cannot be binned into intervals containing months or years

SELECT date_bin('12 weeks', '2021-06-01 10:05:10', '2021-06-01');

        date_bin
------------------------
 2021-06-01 00:00:00+02
(1 row)

SELECT date_bin('365 days', '2021-06-01 10:05:10', '2021-06-01');

        date_bin
------------------------
 2021-06-01 00:00:00+02
(1 row)
```

La fonction `date_bin` a un effet similaire à `date_trunc` lorsqu'elle est utilisée avec les intervalles `1 hour` et `1 minute`.

Documentation : #FUNCTIONS-DATETIME-TRUNC[17]

```
SELECT date_bin('1 hour', '2021-06-01 10:05:10'::timestamp, '2021-06-01'),
       date_trunc('hour', '2021-06-01 10:05:10'::timestamp);

      date_bin       |     date_trunc
---------------------+---------------------
```

[17] https://docs.postgresql.fr/14/functions-datetime.html#FUNCTIONS-DATETIME-TRUNC

```
 2021-06-01 10:00:00 | 2021-06-01 10:00:00
(1 row)
```

```
SELECT date_bin('1 minute', '2021-06-01 10:05:10'::timestamp, '2021-06-01'),
       date_trunc('minute', '2021-06-01 10:05:10'::timestamp);
```

```
      date_bin       |     date_trunc
---------------------+---------------------
 2021-06-01 10:05:00 | 2021-06-01 10:05:00
(1 row)
```

1.5.11 POSSIBILITÉ D'ATTACHER UN ALIAS À UN JOIN .. USING

- Permet de référencer les colonnes de jointures
- Syntaxe : `SELECT ... FROM t1 JOIN t2 USING (a, b, c) AS x`

Il est désormais possible d'utiliser un alias sur une jointure effectuée avec la syntaxe `JOIN .. USING`. Ce dernier peut être utilisé pour référencer les colonnes de jointures.

C'est une fonctionnalité du standard SQL. Elle s'ajoute à la liste des fonctionnalitées supportées[18] .

Exemple :

```
CREATE TABLE region (
  region_id int,
  region_name text
);

CREATE TABLE ville (
  ville_id int,
  region_id int,
  ville_name text
);
```

```
SELECT c.*
  FROM ville a INNER JOIN region b USING (region_id) AS c \gdesc
```

```
  Column   |  Type
-----------+---------
 region_id | integer
```

---

[18] https://docs.postgresql.fr/14/features.html

1.6 SUPERVISION

1.6.1 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).

1.6.2 NOUVELLE VUE PG_STAT_PROGRESS_COPY

- Possibilité de suivre l'avancement d'un COPY avec la vue pg_stat_progress_copy

Il est maintenant possible de surveiller la progression d'une instruction COPY grâce à la vue système pg_stat_progress_copy. Celle-ci retourne une ligne par *backend* lançant un COPY.

```
CREATE TABLE test_copy (i int);
INSERT INTO test_copy SELECT generate_series (1,10000000);
COPY test_copy TO '/tmp/test_copy';

SELECT * FROM pg_stat_progress_copy \gx
```

```
-[ RECORD 1 ]----+---------
pid              | 39148
datid            | 16384
datname          | test
relid            | 36500
command          | COPY TO
type             | FILE
bytes_processed  | 43454464
bytes_total      | 0
tuples_processed | 5570696
tuples_excluded  | 0
```

Parmi ces informations, on retrouve le type de COPY exécuté (command), le type d'entrée/sortie utilisé (type), ainsi que le nombre d'octets déjà traités (bytes_processed) et le nombre de lignes déjà insérées (tuples_processed).

Pour le champ tuples_excluded, il n'est renseigné qu'en cas d'utilisation d'une clause WHERE et remonte le nombre de lignes exclues par cette même clause.

```
COPY test_copy FROM '/tmp/test_copy' WHERE i > 1000;

SELECT * FROM pg_stat_progress_copy \gx
```

```
-[ RECORD 1 ]----+----------
pid              | 39148
datid            | 16384
datname          | test
relid            | 36500
command          | COPY FROM
type             | FILE
```

```
bytes_processed  | 17563648
bytes_total      | 78888897
tuples_processed | 2329752
tuples_excluded  | 1000
```

Le champ `bytes_total` correspond à la taille en octets de la source de données. Il n'est renseigné que dans le cadre d'un `COPY FROM` et si la source de données est située sur le même serveur que l'instance PostgreSQL. Il ne sera pas renseigné si le champ `type` est à `PIPE`, ce qui équivaut à `COPY FROM … STDIN` ou à une commande psql `\copy`.

```
\copy test_copy FROM '/tmp/test_copy' WHERE i > 1000;

SELECT * FROM pg_stat_progress_copy \gx

-[ RECORD 1 ]----+----------
pid              | 39148
datid            | 16384
datname          | test
relid            | 36500
command          | COPY FROM
type             | PIPE
bytes_processed  | 17150600
bytes_total      | 0
tuples_processed | 2281713
tuples_excluded  | 1000
```

Pour la même raison, `bytes_total` n'est pas renseigné lors d'une restauration de sauvegarde logique avec `pg_restore` :

```
SELECT pid, p.datname, relid::regclass, command, type, a.application_name,
bytes_processed, bytes_total, tuples_processed, tuples_excluded,
a.query, a.query_start
FROM pg_stat_progress_copy p INNER JOIN pg_stat_activity a USING(pid) \gx

-[ RECORD 1 ]----+-----------------------------------------------------
pid              | 223301
datname          | scratch
relid            | textes
command          | COPY FROM
type             | PIPE
application_name | pg_restore
bytes_processed  | 111194112
bytes_total      | 0
tuples_processed | 839619
tuples_excluded  | 0
query            | COPY public.textes (livre, ligne, contenu) FROM stdin;+
                 |
query_start      | 2021-11-23 11:11:15.685557+01
```

De même lors d'une sauvegarde logique (ici avec `pg_dumpall`) :

```
-[ RECORD 1 ]----+-------------------------------------------------------------
pid              | 223652
datname          | magasin
relid            | 256305
command          | COPY TO
type             | PIPE
application_name | pg_dump
bytes_processed  | 22817772
bytes_total      | 0
tuples_processed | 402261
tuples_excluded  | 0
query            | COPY magasin.lots (numero_lot, transporteur_id, numero_suivi,
                   date_depot, date_expedition, date_reception) TO stdout;
query_start      | 2021-11-23 11:14:09.755587+01
```

1.6.3 NOUVELLE VUE PG_STAT_REPLICATION_SLOTS

- Donne des statistiques sur l'utilisation des *slots* de réplication **logique**
- Ajout de la fonction `pg_stat_reset_replication_slot`

Il est maintenant possible d'obtenir des statistiques d'utilisation des *slots* de réplication logique (et non physique) via la vue système `pg_stat_replication_slots`. La supervision de la réplication logique était jusque là assez délicate.

Voici la description des colonnes de cette vue :

- `slot_name` : nom du *slot* de réplication ;
- `spill_txns` : nombre de transactions écritens sur disque lorsque la mémoire utilisée pour décoder les changements depuis les WAL a dépassé la valeur du paramètre `logical_decoding_work_mem`. Ce compteur est incrémenté par les transactions de haut niveau et par les sous-transactions ;
- `spill_count` : nombre de fois où des transactions ont été écrites sur disque lors du décodage des changements depuis les WAL. Ce compteur est incrémenté à chaque fois qu'une transaction est écrite sur disque. Une même transaction peut être écrite plusieurs fois ;
- `spill_bytes` : quantité de données écrite sur disque lors du décodage des changements depuis les WAL. Ce compteur et ceux liés aux écritures sur disque peuvent servir pour mesurer les I/O générés pendant le décodage logique et permettre d'optimiser le paramètre `logical_decoding_work_mem` ;
- `stream_txns` : nombre de transactions en cours envoyées directement au plugin de décodage logique lorsque la mémoire utilisée pour le décodage des change-

ments depuis les WAL a dépassé le paramètre `logical_decoding_work_mem`. Le flux de réplication ne fonctionne que pour les transactions de haut niveau (les sous-transactions ne sont pas envoyées indépendamment), ainsi le compteur n'est pas incrémenté pour les sous-transactions ;

- `stream_count` : nombre de fois où des transactions en cours ont été envoyées au plugin de décodage logique lors du décodage des changements depuis les WAL. Ce compteur est incrémenté chaque fois qu'une transaction est envoyée. La même transaction peut être envoyée plusieurs fois ;
- `stream_bytes` : quantité de données envoyée par flux au plugin de décodage logique lors du décodage des changements depuis les WAL. Ce compteur et ceux liés aux plugin de décodage logique peuvent servir pour optimiser le paramètre `logical_decoding_work_mem` ;
- `total_txns` : nombre de transactions décodées et envoyées au plugin de décodage logique. Ne comprend que les transactions de haut niveau (pas de sous-transaction). Cela inclut les transactions écrites sur disques et envoyées par flux ;
- `total_bytes` : quantité de données décodée et envoyée au plugin de décodage logique lors du décodage des changements depuis les WAL. Cela inclut les transactions envoyées par flux et écrites sur disques ;
- `stats_reset` : date de remise à zéro des statistiques.

Concernant les colonnes `stream_txns`, `stream_count` et `stream_count`, celles-ci ne seront renseignées qu'en cas d'utilisation du mode *streaming in-progress* (autre nouveauté de PostgreSQL 14). Il faudra pour cela ajouter la clause `(streaming=on)` lors de la création de la souscription.

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

Une nouvelle fonction est également disponible : `pg_stat_reset_replication_slot`. Elle permet la remise à zéro des statisques de la vue `pg_stat_replication_slots` et peut prendre comme paramètre `NULL` ou le nom d'un *slot* de réplication. Dans le cas de `NULL`, toutes les statistiques seront remises à zéro. Si un nom de *slot* est précisé, seules les statistiques du *slot* en question seront réinitialisées.

```
-- RAZ pour un slot précis
SELECT pg_stat_reset_replication_slot(slot_name);

-- RAZ pour tous les slots
SELECT pg_stat_reset_replication_slot(NULL);
```

1.6.4 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.

1.6.5 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.

1.6.6 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
```

1.6.7 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
```

1.7 PERFORMANCES

1.7.1 AMÉLIORATIONS DE L'INDEXATION GIST / SPGIST

- Amélioration des performances de certains index GiST
  - plus rapides, plus petits
  - type : `point`
- Support des index SPGiST couvrants

Améliorations lors de la création d'index GiST

La création de certains index GiST est rendue plus rapide par l'exécution d'un pré-tri. Un effet secondaire de cette amélioration est que la taille des index bénéficiant de cette optimisation est plus petite. Cela va permettre de diminuer la durée des opérations de maintenances sur ces index GiST (`INDEX`, `REINDEX`) et limiter l'espace utilisé.

```
-- PostgreSQL 14
\timing on
CREATE TABLE gist_fastbuild
    AS SELECT point(random(),random()) as pt
         FROM  generate_series(1,10000000,1);
COPY gist_fastbuild TO '/tmp/gist_fastbuild.copy';

CREATE INDEX ON gist_fastbuild USING gist (pt);
-- Time: 15837.450 ms (00:15.837)
```

```
=# \di+ gist_fastbuild_pt_idx
                              List of relations
         Name          | Type  |     Table      | Access method |  Size
-----------------------+-------+----------------+---------------+--------
 gist_fastbuild_pt_idx | index | gist_fastbuild | gist          | 474 MB
```

```
EXPLAIN (ANALYZE, BUFFERS)
 SELECT pt FROM gist_fastbuild
  WHERE pt <@ box(point(.5,.5), point(.75,.75));

                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using gist_fastbuild_pt_idx on gist_fastbuild
  (cost=0.42..419.42 rows=10000 width=16)
  (actual time=0.497..130.077 rows=624484 loops=1)
   Index Cond: (pt <@ '(0.75,0.75),(0.5,0.5)'::box)
   Heap Fetches: 0
   Buffers: shared hit=301793
 Planning Time: 4.406 ms
 Execution Time: 149.662 ms
```

```
-- PostgreSQL 13
\timing on
CREATE TABLE gist_fastbuild(pt point);
COPY gist_fastbuild FROM '/tmp/gist_fastbuild.copy';

CREATE INDEX ON gist_fastbuild USING gist (pt);
-- Time: 168469.405 ms (02:48.469)
```

```
=# \di+ gist_fastbuild_pt_idx
                                          List of relations
         Name          | Type  |     Table      |  Size
-----------------------+-------+----------------+--------
 gist_fastbuild_pt_idx | index | gist_fastbuild | 711 MB
```

```
EXPLAIN (ANALYZE, BUFFERS)
 SELECT pt FROM gist_fastbuild
  WHERE pt <@ box(point(.5,.5), point(.75,.75));

                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using gist_fastbuild_pt_idx on gist_fastbuild
  (cost=0.42..539.42 rows=10000 width=16)
  (actual time=0.492..107.536 rows=624484 loops=1)
   Index Cond: (pt <@ '(0.75,0.75),(0.5,0.5)'::box)
   Heap Fetches: 0
   Buffers: shared hit=17526
 Planning Time: 0.143 ms
 Execution Time: 126.951 ms
```

On voit que le temps d'exécution et la taille de l'index ont beaucoup diminué en version 14. Malheureusement, dans ce cas, le plan en version 14 montre une augmentation du nombre de pages lues et du temps d'exécution.

Pour permettre cette fonctionnalité, une nouvelle fonction de support optionnelle a été ajoutée à la méthode d'accès `gist`. Lorsqu'elle est définie, la construction de l'index passe par une étape de tri des données avec un ordre défini par la fonction de support. Cela permet de regrouper les enregistrements plus efficacement et donc de réduire la taille de l'index.

Actuellement seule la classe d'opérateur pour les types `point` dispose de cette fonction :

```
SELECT f.opfname AS famille,
       ap.amproc AS fonction_de_support,
       tl.typname AS type_gauche_op,
       tr.typname AS type_droit_op,
       m.amname AS methode
  FROM pg_amproc ap
       INNER JOIN pg_type tl ON tl.oid = ap.amproclefttype
```

```
       INNER JOIN pg_type tr ON tr.oid = ap.amprocrighttype
       INNER JOIN pg_opfamily f ON ap.amprocfamily = f.oid
       INNER JOIN pg_am m ON f.opfmethod = m.oid
 WHERE ap.amproc::text LIKE '%sortsupport'
   AND m.amname = 'gist';

  famille   |   fonction_de_support    | type_gauche_op | type_droit_op | methode
------------+--------------------------+----------------+---------------+---------
 point_ops  | gist_point_sortsupport | point          | point         | gist
```

Index SPGiST couvrants

Il est désormais possible d'inclure des colonnes dans les index SPGiST de la même façon qu'il était possible d'inclure des colonnes dans les index btree depuis la version v11 et GiST depuis la version v12.

```
CREATE INDEX airports_coordinates_quad_idx ON airports_ml
 USING spgist(coordinates) INCLUDE (name);
```

1.7.2 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`.

1.7.3 NOUVELLES CLASSES D'OPÉRATEURS POUR LES INDEX BRIN

- BRIN : index compact
  - jusque là : si corrélation ordres physique/logique
- Nouvelles classes d'opérateurs
  - `*_bloom_ops` : permet d'utiliser les index BRIN pour des données dont l'ordre physique ne coïncide pas avec l'ordre logique
  - `*_minmax_multi_ops` : permet d'utiliser les index BRIN avec des prédicats de sélection de plage de données

Les index BRIN permettent de créer des index très petits. Jusque là ils n'étaient efficaces que lorsque l'ordre physique des données est corrélé avec l'ordre logique. Malheureusement, dès que cette corrélation change les performances se dégradent, ce qui limite les cas d'utilisation : tables d'historisation, décisionnel rechargé régulièrement, etc.

PostgreSQL 14 élargit le champ d'utilisation des index BRIN. De nouvelles classes d'opérateurs ont été créées pour les index BRIN, déclinées pour chaque type de données[19] , et regroupées en deux grandes familles :

- `*_bloom_ops`
- `*_minmax_multi_ops`

à ne pas confondre avec les index bloom[20] , qui nécessitent une extension de même nom (des principes sont voisins mais l'implémentation est différente).

```
SELECT amname,
       CASE WHEN opcname LIKE '%bloom%' THEN '*_bloom_ops'
            WHEN opcname LIKE '%multi%' THEN '*_minmax_multi_ops'
            ELSE                             '*_mimmax_ops'
       END AS "classes d'opérateurs",
       count(*) as  "types supportés"
  FROM pg_opclass c
       INNER JOIN pg_am m ON c.opcmethod = m.oid
 WHERE opcname LIKE ANY(ARRAY['%bloom%', '%minmax%'])
GROUP BY 1, 2;

 amname | classes d'opérateurs | types supportés
--------+----------------------+-----------------
```

[19] https://docs.postgresql.fr/14/brin-builtin-opclasses.html
[20] https://docs.postgresql.fr/14/bloom.html

```
 brin   | *_mimmax_ops         |              26
 brin   | *_minmax_multi_ops   |              19
 brin   | *_bloom_ops          |              24
(3 rows)
```

Classe d'opérateur bloom_ops

Les classes d'opérateurs `*_bloom_ops` visent à permettre l'utilisation d'index BRIN pour satisfaire des prédicats d'égalité même si l'ordre physique de la table ne correspond pas à son ordre logique.

```
CREATE TABLE bloom_test (id uuid, padding text);

INSERT INTO bloom_test
  SELECT md5((mod(i,1000000)/100)::text)::uuid, md5(i::text)
    FROM generate_series(1,2000000) s(i);

VACUUM ANALYZE bloom_test;
```

Pour le test, nous allons désactiver le parallélisme et les parcours séquentiels afin de se focaliser sur l'utilisation des index :

```
SET enable_seqscan TO off;
SET max_parallel_workers_per_gather TO 0;
```

Commençons par tester avec un index B-tree :

```
CREATE INDEX test_btree_idx on bloom_test (id);

EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM bloom_test
   WHERE id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da';
```

Voici le plan de la requête :

```
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on bloom_test (cost=5.96..742.23 rows=198 width=49)
                                (actual time=0.069..0.130 rows=200 loops=1)
   Recheck Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
   Heap Blocks: exact=5
   Buffers: shared hit=9
   ->  Bitmap Index Scan on test_btree_idx
                                (cost=0.00..5.91 rows=198 width=0)
                                (actual time=0.043..0.044 rows=200 loops=1)
         Index Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
         Buffers: shared hit=4
 Planning Time: 0.168 ms
 Execution Time: 0.198 ms
```

Ce plan est optimal. `\di+` indique que l'index fait 13 Mo.

Essayons maintenant avec un index BRIN utilisant les `uuid_minmax_ops` (la classe d'opérateur par défaut) :

```
DROP INDEX test_btree_idx;

CREATE INDEX test_brin_minmax_idx ON bloom_test USING brin (id);
```

Ce nouvel index ne pèse que 48 ko, ce qui est dérisoire.

Relançons la même requête :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM bloom_test
   WHERE id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da';
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on bloom_test (cost=17.23..45636.23 rows=198 width=49)
                                (actual time=1.527..216.911 rows=200 loops=1)
   Recheck Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
   Rows Removed by Index Recheck: 1999800
   Heap Blocks: lossy=20619
   Buffers: shared hit=1 read=20620 written=2
   ->  Bitmap Index Scan on test_brin_minmax_idx
                                (cost=0.00..17.18 rows=2000000 width=0)
                                (actual time=1.465..1.465 rows=206190 loops=1)
         Index Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
         Buffers: shared hit=1 read=1
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.132 ms
 Execution Time: 216.968 ms
```

Le temps d'exécution de la requête avec cet index est beaucoup plus long qu'avec l'index B-tree. Cela s'explique par le grand nombre d'accès qui doivent être réalisés, environ 20 620 contre une dizaine, et surtout par le très grand nombre de vérifications qui doivent être faites dans la table (presque 2 millions). C'est logique : l'index BRIN est utilisé ici à contre-emploi, dans une configuration où les données (des UUID) sont presque aléatoires et non triés.

Pour terminer, essayons avec l'index BRIN et la nouvelle classe d'opérateur :

```
DROP INDEX test_brin_minmax_idx;

CREATE INDEX test_brin_bloom_idx on bloom_test USING brin (id uuid_bloom_ops);

EXPLAIN (ANALYZE,BUFFERS)
```

```
SELECT * FROM bloom_test
 WHERE id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da';
```

Voici le plan de la requête :

```
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on bloom_test (cost=145.23..45764.23 rows=198 width=49)
                                (actual time=5.369..7.502 rows=200 loops=1)
   Recheck Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
   Rows Removed by Index Recheck: 25656
   Heap Blocks: lossy=267
   Buffers: shared hit=301
   ->  Bitmap Index Scan on test_brin_bloom_idx
                                (cost=0.00..145.18 rows=2000000 width=0)
                                (actual time=5.345..5.345 rows=2670 loops=1)
         Index Cond: (id = 'cfcd2084-95d5-65ef-66e7-dff9f98764da'::uuid)
         Buffers: shared hit=34
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.129 ms
 Execution Time: 7.553 ms
```

Le nouvel index entraîne la vérification de soixante fois moins de blocs en mémoire que l'index BRIN *minmax* (301 contre 20 620), et le nombre de lignes vérifiées dans la table est également nettement inférieur (environ 27 000 contre 2 millions). Les performances globales sont en conséquence bien meilleures qu'avec l'index BRIN *minmax*. (Dans ce cas précis, le coût estimé est cependant légèrement supérieur à cause de la taille des index : si les deux index sont présents en même temps sur la table, l'index BRIN _minmax_sera donc choisi.)

Comparé au plan avec l'index B-tree, les performances restent nettement moins bonnes. C'est principalement dû au nombre d'accès nécessaires pour traiter le prédicat.

En répétant les tests avec des quantités de doublons différentes, on voit que l'index BRIN *bloom* permet d'accéder à un nombre plus petit de pages que l'index BRIN *minmax*, ce qui le rend souvent plus performant. L'index B-tree est toujours plus performant.

La comparaison des tailles montre que l'index BRIN utilisant les `uuid_bloom_ops` est plus grand que l'index BRIN classique, mais nettement plus petit que l'index B-tree.

```
                        List of relations
        Name         | Type  |   Table    | Access method |  Size
---------------------+-------+------------+---------------+--------
 test_brin_bloom_idx | index | bloom_test | brin          | 304 kB
 test_brin_minmax_idx | index | bloom_test | brin          | 48 kB
 test_btree_idx      | index | bloom_test | btree         | 13 MB
```

La classe d'opérateur `*_bloom_ops` accepte deux paramètres qui permettent de dimensionner l'index bloom :

- `n_distinct_per_range` : permet d'estimer le nombre de valeurs distinctes dans un ensemble de blocs BRIN. Il doit être supérieur à -1 et sa valeur par défaut est -0.1. Il fonctionne de la même manière que la colonne `n_distinct` de la vue `pg_stats`. S'il est positif, il indique le nombre de valeurs distinctes. S'il est négatif, il indique la fraction de valeurs distinctes pour cette colonne dans la table.
- `false_positive_rate` : permet d'estimer le nombre de faux positifs généré par l'index bloom. Il doit être compris entre 0.0001 et 0.25, sa valeur par défaut est 0.01.

Un paramétrage incorrect peut rendre impossible la création de l'index :

```
CREATE INDEX test_bloom_parm_idx on bloom_test
       USING brin (id uuid_bloom_ops(false_positive_rate=.0001)
);

ERROR:  the bloom filter is too large (8924 > 8144)
```

Il est impératif de bien tester les insertions comme le montre cet exemple :

```
CREATE TABLE bloom_test (id uuid, padding text);
CREATE INDEX test_bloom_parm_idx on bloom_test
       USING brin (id uuid_bloom_ops(false_positive_rate=.0001)
);
INSERT INTO bloom_test VALUES (md5('a')::uuid, md5('a'));
```

Si la table est vide, on voit que l'erreur ne survient pas lors de la création de l'index mais lors de la première insertion :

```
CREATE TABLE
CREATE INDEX
ERROR:  the bloom filter is too large (8924 > 8144)
```

Classe d'opérateur minmax_multi_ops

Cette version a également introduit les classes d'opérateurs `*_minmax_multi_ops` qui visent à permettre l'utilisation d'index BRIN pour satisfaire des prédicats de sélection de plages de valeurs même si l'ordre physique de la table ne correspond pas à son ordre logique.

```
CREATE TABLE brin_multirange AS
     SELECT '2021-09-29'::timestamp - INTERVAL '1 min' * x AS d
       FROM generate_series(1, 1000000) AS F(x);

UPDATE brin_multirange SET d ='2021-04-05'::timestamp   WHERE random() < .01;
```

Une fois de plus, nous allons désactiver le parallélisme et les parcours séquentiels afin de se concentrer sur l'utilisation des index :

```
SET enable_seqscan TO off;
SET max_parallel_workers_per_gather TO 0;
```

Commençons par tester une requête avec un BETWEEN sur un index B-tree :

```
CREATE INDEX brin_multirange_btree_idx
  ON brin_multirange USING btree (d);

EXPLAIN (ANALYZE, BUFFERS)
  SELECT * FROM brin_multirange
   WHERE d BETWEEN '2021-04-05'::timestamp AND '2021-04-06'::timestamp;
```

Voici le plan généré :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on brin_multirange (cost=107.67..4861.46 rows=5000 width=8)
                                     (actual time=0.254..0.698 rows=1429 loops=1)
   Recheck Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
              AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
   Heap Blocks: exact=7
   Buffers: shared hit=14
   ->  Bitmap Index Scan on brin_multirange_btree_idx
                                     (cost=0.00..106.42 rows=5000 width=0)
                                     (actual time=0.227..0.227 rows=1429 loops=1)
         Index Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
                  AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
         Buffers: shared hit=7
 Planning Time: 0.119 ms
 Execution Time: 0.922 ms
```

(Le plan exact peut varier en fonction du `random()` plus haut.) C'est satisfaisant, même si cet exemple est délibérément non optimal pour les comparaisons qui suivent (avec un VACUUM, on peut même arriver à un *Index Only Scan* encore plus rapide).

Testons la même requête en supprimant avec un index BRIN classique (*minmax*) :

```
DROP INDEX brin_multirange_btree_idx;

CREATE INDEX brin_multirange_minmax_idx
  ON brin_multirange USING brin (d);

EXPLAIN (ANALYZE, BUFFERS)
  SELECT * FROM brin_multirange
   WHERE d BETWEEN '2021-04-05'::timestamp AND '2021-04-06'::timestamp;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on brin_multirange (cost=12.42..4935.32 rows=1550 width=8)
                                     (actual time=5.486..7.959 rows=1429 loops=1)
   Recheck Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
              AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
   Rows Removed by Index Recheck: 53627
   Heap Blocks: lossy=246
   Buffers: shared hit=248
   ->  Bitmap Index Scan on brin_multirange_minmax_idx
                                     (cost=0.00..12.03 rows=30193 width=0)
                                     (actual time=0.056..0.056 rows=2460 loops=1)
         Index Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
                  AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
         Buffers: shared hit=2
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.146 ms
 Execution Time: 8.039 ms
```

Comparé à l'index B-tree, l'index BRIN *minmax* accède à beaucoup plus de blocs et effectue beaucoup plus de vérifications, cela se ressent au niveau du temps d'exécution de la requête qui est plus important.

Pour finir, testons avec l'index BRIN *multirange_minmax* et la méthode `timestamp_minmax_multi_ops` adaptée à ce champ :

```
DROP INDEX brin_multirange_minmax_idx;

CREATE INDEX brin_multirange_minmax_multi_idx
  ON brin_multirange USING brin (d timestamp_minmax_multi_ops);

EXPLAIN (ANALYZE, BUFFERS)
  SELECT * FROM brin_multirange
   WHERE d BETWEEN '2021-04-05'::timestamp AND '2021-04-06'::timestamp;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on brin_multirange (cost=16.42..4939.32 rows=1550 width=8)
                                     (actual time=5.689..6.300 rows=1429 loops=1)
   Recheck Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
              AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
   Rows Removed by Index Recheck: 27227
   Heap Blocks: lossy=128
   Buffers: shared hit=131
   ->  Bitmap Index Scan on brin_multirange_minmax_multi_idx
                                     (cost=0.00..16.03 rows=30193 width=0)
```

```
                                  (actual time=0.117..0.117 rows=1280 loops=1)
        Index Cond: ((d >= '2021-04-05 00:00:00'::timestamp without time zone)
                 AND (d <= '2021-04-06 00:00:00'::timestamp without time zone))
        Buffers: shared hit=3
Planning:
  Buffers: shared hit=1
Planning Time: 0.148 ms
Execution Time: 6.380 ms
```

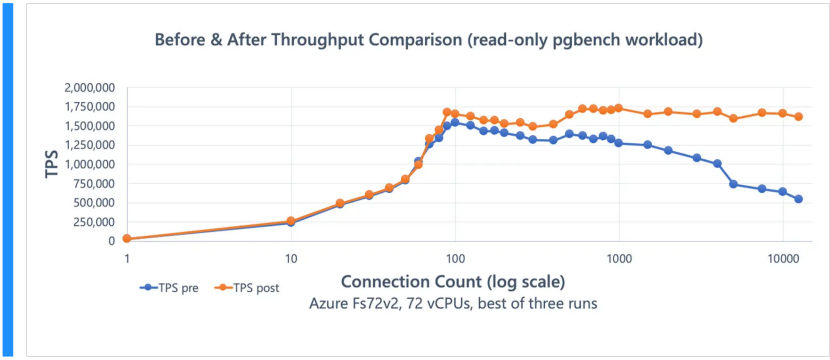
Le plan avec la nouvelle classe d'opérateur accède à moins de blocs que celui avec la classe d'opérateur par défaut. Le temps d'exécution est donc plus court. Le coût estimé par l'optimiseur est légèrement supérieur à l'index BRIN *minmax*. Si les deux index sont présents, l'index bin *minmax* sera donc choisi.

On peut voir que l'index BRIN avec la classe d'opérateur `*_minmax_multi_ops` est plus gros que l'index BRIN traditionnel, mais reste beaucoup plus petit que l'index B-tree.

```
               Name                | Type  |      Table       |Access method | Size
-----------------------------------+-------+------------------+--------------+-------
 brin_multirange_btree_idx         | index |brin_multirange   |btree         | 21 MB
 brin_multirange_minmax_idx        | index |brin_multirange   |brin          | 48 kB
 brin_multirange_minmax_multi_idx  | index |brin_multirange   |brin          | 56 kB
```

Pour conclure : les index B-tree sont toujours plus performants que les index BRIN. La nouvelle classe d'opérateur améliore les performances par rapport aux index BRIN classiques pour les données mal triées physiquement. Ce gain de performance est fait au prix d'une augmentation de la taille de l'index, mais elle reste toujours bien inférieure à celle d'un index B-tree. Cette nouvelle version permet donc de rendre plus polyvalents les index BRIN tout en conservant leurs atouts, dans des contextes où les tailles des index peuvent poser des problèmes.

1.7.4 CONNEXIONS SIMULTANÉES EN LECTURE SEULE



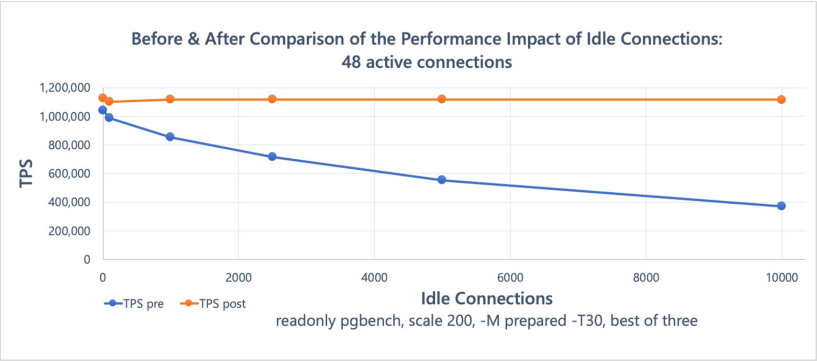
(Les graphiques sont empruntés à l'article de blog d'Andres Freund cité plus bas.)

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[21] , 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[22] , 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.

[21] https://www.postgresql.org/message-id/flat/20200301083601.ews6hz5dduc3w2se@alap3.anarazel.de
[22] 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.

---

2 ATELIERS

- Découvrir les nouveaux rôles prédéfinis
- Mise en place d'un sharding minimal
- Outil pg_rewind

2.1 DÉCOUVRIR LES NOUVEAUX RÔLES PRÉDÉFINIS

- Utiliser le rôle `pg_database_owner` dans une base *template*
- Exporter avec le rôle `pg_read_all_data`
- Importer avec le rôle `pg_write_all_data`

2.1.1 UTILISER LE RÔLE PG_DATABASE_OWNER DANS UNE BASE TEMPLATE

- Ouvrir un terminal puis emprunter l'identité de l'utilisateur `postgres` sur votre machine.

```
sudo su - postgres
```

- Se connecter à l'instance PostgreSQL.

```
psql
```

- Créer une nouvelle base modèle `tp1_template` à l'aide de l'instruction `CREATE DATABASE`.

```
CREATE DATABASE tp1_template;
```

- Se connecter à la base de données `tp1_template`.

```
\c tp1_template
```

- Créer une table `members` comme suit dans la base de données `tp1_template`:

```
CREATE TABLE members (
  id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  name varchar(25),
  age int CHECK (age > 0)
);
```

Par défaut, le propriétaire d'un objet correspond au rôle qui le crée dans la base, en l'occurrence le rôle `postgres` dans notre exemple. La méta-commande `\d` de l'outil `psql` permet de lister les tables, vues et séquences présentes dans les schémas courants de l'utilisateur.

```
tp1_template=# \d
               List of relations
 Schema |      Name      |   Type   |  Owner
```

```
--------+----------------+----------+----------
 public | members        | table    | postgres
 public | members_id_seq | sequence | postgres
```

Remarque : L'utilisation du type `IDENTITY` déclenche la création automatique d'une séquence.

- Modifier le propriétaire des objets avec le nouveau rôle `pg_database_owner`.

```
ALTER TABLE members OWNER TO pg_database_owner;
```

En exécutant une nouvelle fois la méta-commande `\d` nous pouvons voir la modification du propriétaire de la séquence et de la table.

```
tp1_template=# \d
                  List of relations
 Schema |      Name      |   Type   |       Owner
--------+----------------+----------+-------------------
 public | members        | table    | pg_database_owner
 public | members_id_seq | sequence | pg_database_owner
```

Il est à noter que le propriétaire de la séquence a été modifié, cela vient du fait de l'utilisation du type `IDENTITY` pour la colonne `id` lors de la création de la table `members`.

- Créer un utilisateur `atelier`.

```
CREATE ROLE atelier NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

- Créer une base de données `tp1` basée sur le modèle `tp1_template`, dont l'utilisateur `atelier` sera propriétaire.

```
CREATE DATABASE tp1 OWNER atelier TEMPLATE tp1_template;
```

- Se connecter à la base de données `tp1`.

```
\c tp1
```

- Exécuter la méta-commande `\d`. Quel est le propriétaire des objets présents dans la base `tp1` ?

Les droits sont similaires à la base modèle `tp1_template`, notamment le propriétaire `pg_database_owner` sur les objets `members` et `members_id_seq`.

```
tp1=> \d
                  List of relations
 Schema |      Name      |   Type   |       Owner
--------+----------------+----------+-------------------
 public | members        | table    | pg_database_owner
 public | members_id_seq | sequence | pg_database_owner
```

- Se connecter à la base `tp1` avec le rôle propriétaire `atelier`.

```
\c tp1 atelier
```

- Ajouter 3 lignes dans la table `members`.

```
INSERT INTO members (name, age) VALUES
  ('Jean', 41),
  ('John', 38),
  ('Jessica', 26);
```

```
INSERT 0 3
```

Sans être explicitement autorisé à écrire dans la table `members`, le rôle `atelier` bénéficie de tous les droits des objets appartenant au rôle `pg_database_owner` en sa qualité de propriétaire de la base de données.

2.1.2 EXPORTER AVEC LE RÔLE PG_READ_ALL_DATA

- Se connecter à la base de données `postgres` avec l'utilisateur **postgres**.

```
\c postgres postgres
```

- Créer un nouvel utilisateur `dump_user`.

```
CREATE ROLE dump_user NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

- Se déconnecter de l'instance PostgreSQL.

```
\q
```

- Exporter les données de la base `tp1` avec l'outil `pg_dump` en tant qu'utilisateur `dump_user`. Que se passe-t-il ?

```
pg_dump --username=dump_user --dbname tp1
```

```
pg_dump: error: query failed: ERROR:  permission denied for table members
pg_dump: error: query was: LOCK TABLE public.members IN ACCESS SHARE MODE
```

- Se connecter de l'instance PostgreSQL.

```
psql
```

- Assigner les droits du rôle `pg_read_all_data` à l'utilisateur `dump_user`.

```
GRANT pg_read_all_data TO dump_user;
```

- Exporter de nouveaux les données de la base `tp1` avec l'outil `pg_dump` en tant qu'utilisateur `dump_user`. Que se passe-t-il ?

```
pg_dump --username=dump_user --dbname tp1
```

```
--
-- PostgreSQL database dump
--
```

```
--
-- Name: members; Type: TABLE; Schema: public; Owner: atelier
--

CREATE TABLE public.members (
    id integer NOT NULL,
    name character varying(25),
    age integer,
    CONSTRAINT members_age_check CHECK ((age > 0))
);

ALTER TABLE public.members OWNER TO atelier;

--
-- Name: members_id_seq; Type: SEQUENCE; Schema: public; Owner: atelier
--

ALTER TABLE public.members ALTER COLUMN id ADD GENERATED ALWAYS AS IDENTITY (
    SEQUENCE NAME public.members_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1
);

--
-- Data for Name: members; Type: TABLE DATA; Schema: public; Owner: atelier
--

COPY public.members (id, name, age) FROM stdin;
1   Jean    41
2   John    38
3   Jessica 26
\.

--
-- Name: members_id_seq; Type: SEQUENCE SET; Schema: public; Owner: atelier
--

SELECT pg_catalog.setval('public.members_id_seq', 3, true);

--
-- Name: members members_pkey; Type: CONSTRAINT; Schema: public; Owner: atelier
--
```

```
ALTER TABLE ONLY public.members
    ADD CONSTRAINT members_pkey PRIMARY KEY (id);

--
-- PostgreSQL database dump complete
--
```

Il est alors possible pour l'utilisateur `dump_user` de sauvegarder la base de données `tp1`.

- Exporter les données globales de l'instance à l'aide de l'outil `pg_dumpall` et le compte `dump_user`.

```
pg_dumpall --username dump_user --globals-only
```

```
--
-- PostgreSQL database cluster dump
--

--
-- Roles
--

CREATE ROLE atelier;
ALTER ROLE atelier
 WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS;
CREATE ROLE postgres;
ALTER ROLE postgres
 WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION BYPASSRLS;
CREATE ROLE dump_user;
ALTER ROLE dump_user
 WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS;

--
-- Role memberships
--

GRANT pg_read_all_data TO dump_user GRANTED BY postgres;

--
-- PostgreSQL database cluster dump complete
--
```

Le compte `dump_user` est bien adapté pour les exports multiples à l'aide de la commande `pg_dumpall` ou de logiciels plus complets comme pg_back[23] .

[23] https://github.com/orgrim/pg_back

2.1.3 IMPORTER AVEC LE RÔLE PG_WRITE_ALL_DATA

- Se connecter à l'instance PostgreSQL.

```
psql
```

- Créer un utilisateur `load_user`.

```
CREATE ROLE load_user WITH NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

- Assigner les droits du rôle `pg_write_all_data` à l'utilisateur `load_user`.

```
GRANT pg_write_all_data TO load_user;
```

- Se déconnecter de l'instance PostgreSQL.

```
\q
```

- Créer un fichier `members.csv` et y ajouter les lignes suivantes.

```
id,name,age
1,Jean,41
2,John,38
3,Jessica,26
4,Johnny,74
5,Joe,55
6,Jennifer,33
7,Jerôme,40
8,Jenny,21
9,Jesus,33
10,Jérémie,12
```

- Se connecter à la base de données `tp1` avec l'utilisateur `load_user`.

```
psql --username=load_user --dbname=tp1
```

- Tenter de lire les données de la table `members` dans la base de données `tp1`.

```
SELECT id, name, age FROM members LIMIT 10;
```

```
ERROR:  permission denied for table members
```

- Charger les données du fichier `members.csv` dans la table `members` à l'aide de la méta-commande `\copy`.

```
\copy members FROM 'members.csv' WITH DELIMITER ',' CSV HEADER;
```

```
ERROR:  duplicate key value violates unique constraint "members_pkey"
CONTEXT:  COPY members, line 2
```

Le chargement tombe en erreur car les premières lignes de la table entre en conflit sur la contrainte de clé primaire. Il est nécessaire d'enrichir le traitement d'import avec une gestion d'erreurs ou une solution plus complexe que l'instruction `COPY`.

- Créer une table `members_copy` qui est une copie de la table `members` sans les données.

```
CREATE TABLE members_copy AS TABLE members WITH NO DATA;
```

- Charger les données du fichier `members.csv` dans la table `members_copy` à l'aide de la méta-commande `\copy`.

```
\copy members_copy FROM 'members.csv' WITH DELIMITER ',' CSV HEADER;
```

- Réaliser une boucle avec une gestion d'erreurs pour ignorer les données déjà présentes dans la table cible `members`.

```
DO $$
DECLARE
  m RECORD;
BEGIN
  FOR m IN (SELECT id, name, age FROM members_copy) LOOP
    BEGIN
      INSERT INTO members (id, name, age) OVERRIDING SYSTEM VALUE
      VALUES (m.id, m.name, m.age);
    EXCEPTION
      WHEN unique_violation THEN null;
    END;
  END LOOP;
END;
$$;
```

Une autre solution, plus rapide, serait de supprimer le contenu de la table `members` avec un `DELETE` et de réaliser le chargement par `COPY`. Cependant, cela nécessite une bonne connaissance du modèle de données, notamment celle des contraintes de clés étrangères et des suppressions en cascade.

2.2 MISE EN PLACE D'UN SHARDING MINIMAL

- Préparer le modèle et configurer les accès distants
- Alimenter une table partitionnée répartie dans plusieurs bases de données
- Étudier les différents cas d'usage

Dans cet exercice, nous visons à mettre en place une architecture distribuée, dites de *sharding*, à travers une table partitionnée dont les lignes sont réparties entre plusieurs tables distantes. Cette répartition doit être longuement étudiée, car la clé et la méthode de partitionnement sont cruciales lors de l'élaboration du modèle de données.

Nous prenons l'exemple d'une table de recensement de la population française, répartie selon les dix-huit subdivisions régionales, telles qu'établies depuis le 1er janvier 2016.

Cette table a vocation de contenir des données démographiques anonymisées, telles que la date et la région de naissance, voire la date de décès si survenu.

2.2.1 PRÉPARER LE MODÈLE DE DONNÉES

- Créer la base `region_template`. Ce *template* permet de recréer le modèle à l'identique dans une nouvelle base sur la même instance par soucis de simplicité.

```
CREATE DATABASE region_template WITH IS_TEMPLATE = true;
```

- Créer dans ce *template* la table `regions` avec les correspondances des régions administratives.

L'identifiant de région correspond aux codes issus de la norme ISO 3166-2[24] et sera stocké dans un champ texte de six caractères.

```
\c region_template

CREATE TABLE regions (
  region_id varchar(6) PRIMARY KEY,
  region text
);

INSERT INTO regions VALUES
  ('FR-ARA', 'Auvergne-Rhône-Alpes'), ('FR-BFC', 'Bourgogne-Franche-Comté'),
  ('FR-BRE', 'Bretagne'), ('FR-CVL', 'Centre-Val de Loire'),
  ('FR-COR', 'Corse'), ('FR-GES', 'Grand Est'),
  ('FR-GUA', 'Guadeloupe'), ('FR-GUF', 'Guyane'),
  ('FR-HDF', 'Hauts-de-France'), ('FR-IDF', 'Île-de-France'),
  ('FR-LRE', 'La Réunion'), ('FR-MTQ', 'Martinique'),
  ('FR-MAY', 'Mayotte'), ('FR-NOR', 'Normandie'),
  ('FR-NAQ', 'Nouvelle-Aquitaine'), ('FR-OCC', 'Occitanie'),
  ('FR-PDL', 'Pays de la Loire'), ('FR-PAC', 'Provence-Alpes-Côte d''Azur');
```

- Ajouter la table `population` avec les champs souhaités pour le recensement et la contrainte de clé étrangère avec la table `regions`.

```
CREATE TABLE population (
  anon_id uuid DEFAULT gen_random_uuid(),
  region_naissance_id varchar(6) NOT NULL,
  date_naissance date NOT NULL,
  date_deces date,
    PRIMARY KEY (anon_id, region_naissance_id),
    FOREIGN KEY (region_naissance_id) REFERENCES regions (region_id)
);
```

[24] https://fr.wikipedia.org/wiki/ISO_3166-2:FR

L'identifiant `anon_id` est de type `uuid` pour obtenir une valeur unique quelle que soit l'instance, alternative indispensable aux séquences qui ne sont pas partagées entre les *shards*.

Dans un cas réel, on imaginerait une transformation de type *hash* sur un identifiant civile, tel que le numéro de sécurité sociale. Cet identifiant permet de préserver la correspondance avec l'individu réel pour mettre à jour ses informations, telle que la date du décés.

- Cloner la base `region_template` pour les dix-huits régions administratives. Dans la pratique, il s'agira de dix-huit bases de données ayant leurs propres ressources serveurs (CPU, mémoire et stockage) sur des instances dédiées.

```
SELECT concat(
  'CREATE DATABASE region_', replace(lower(region_id), '-', ''),
  ' WITH TEMPLATE = region_template;'
) FROM regions;
\gexec
```

2.2.2 CONFIGURER LES ACCÈS DISTANTS

- Activer le mode `trust` ou `md5` pour l'interface `localhost` dans le fichier `pg_hba.conf`. Dans cet exercice, le compte `postgres` est propriétaire des bases et réalise les connexions distantes. Recharger le service PostgreSQL

```
# TYPE  DATABASE        USER            ADDRESS             METHOD
host    all             postgres        localhost           trust
```

```
SELECT pg_reload_conf();
```

- Créer la base principale `recensement` à partir du *template* précédent.

```
CREATE DATABASE recensement WITH TEMPLATE = region_template;
```

- Dans cette nouvelle base, installer l'extension `postgres_fdw`, et créer les serveurs distants pour chaque base de région, ainsi que les correspondances d'utilisateur nécessaires à l'authentification des tables distantes.

> Le paquet `postgresql14-contrib` est requis pour installer l'extension.

```
\c recensement

CREATE EXTENSION postgres_fdw;
SELECT concat(
  'CREATE SERVER IF NOT EXISTS server_', replace(lower(region_id), '-', ''),
  ' FOREIGN DATA WRAPPER postgres_fdw OPTIONS (',
  '  host ''localhost'', port ''' || current_setting('port') || ''',',
  '  dbname ''region_', replace(lower(region_id), '-', ''), ''',',
```

```
  '  async_capable ''on'', fetch_size ''1000''',
  ' );'
) FROM regions;
\gexec

SELECT concat(
  'CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER server_',
    replace(lower(region_id), '-', ''), ';'
) FROM regions;
\gexec
```

L'option `async_capable` doit être activée pour bénéficier de la lecture concurrente sur les partitions distantes.

- Dans la base principale `recensement`, recréer la table `population` en table partitionnée de type `LIST` sur la colonne `region_naissance_id`.

```
DROP TABLE IF EXISTS population;
CREATE TABLE population (
  anon_id uuid DEFAULT gen_random_uuid(),
  region_naissance_id varchar(6) NOT NULL,
  date_naissance date NOT NULL,
  date_deces date
) PARTITION BY LIST (region_naissance_id);
```

PostgreSQL ne supporte pas (encore) les contraintes de clés primaires et de clés étrangères pour des partitions distantes. C'est pour cette raison que la table partitionnée n'en fait pas mention. Le contournement consiste à les définir scrupuleusement sur les serveurs distants et de reposer sur un identifiant universellement unique comme le type `uuid` pour régler les risques de conflits.

- Pour chaque base région, créer une partition distante rattachée à la table principale `population`.

```
SELECT concat(
  'CREATE FOREIGN TABLE population_', replace(lower(region_id), '-', ''),
  '  PARTITION OF population FOR VALUES IN (''',region_id,''')',
  '  SERVER server_', replace(lower(region_id), '-', ''),
  '  OPTIONS (table_name ''population'');'
) FROM regions;
\gexec
```

2.2.3 ALIMENTER UNE TABLE PARTITIONNÉE RÉPARTIE DANS PLUSIEURS BASES DE DONNÉES

- Insérer des données aléatoires dans la table principale.

```
\timing on
INSERT INTO population (region_naissance_id, date_naissance)
SELECT region_id, d FROM regions
 CROSS JOIN generate_series(1, 1000) i
 CROSS JOIN generate_series('1970-01-01', '2010-01-01', '1 year'::interval) d;

INSERT 0 738000
Time: 79229.338 ms (01:19.229)
```

L'insertion est pénalisée par la distribution des lignes à travers les différentes tables distantes, les contraintes d'intégrité qui s'appliquent, la mise à jour des index de clé primaire, voire l'appel à la fonction `gen_random_uuid()`.

- Exécuter la commande `ANALYZE` sur la table principale.

```
ANALYZE VERBOSE population;
```

2.2.4 ÉTUDIER LES DIFFÉRENTS CAS D'USAGE

On ajoute artificiellement une latence de 10ms sur l'interface *loopback* pour simuler un trafic réaliste entre les différents nœuds de calcul.

```
sudo tc qdisc add dev lo root netem delay 10msec
```

- Afficher le plan d'exécution d'une requête permettant de comptabiliser le nombre de naissance en 2010.

```
EXPLAIN (analyze, costs off)
 SELECT count(anon_id) FROM population
  WHERE date_naissance BETWEEN '2010-01-01' AND '2011-01-01';

                                QUERY PLAN
--------------------------------------------------------------------------------
Aggregate (actual time=499.397..499.439 rows=1 loops=1)
   ->  Append (actual time=449.175..498.574 rows=18000 loops=1)
         ->  Async Foreign Scan on population_frara population_1
               (actual time=27.010..27.121 rows=1000 loops=1)
         ->  Async Foreign Scan on population_frbfc population_2
               (actual time=24.218..24.306 rows=1000 loops=1)
...
         ->  Async Foreign Scan on population_frpac population_17
               (actual time=26.453..26.552 rows=1000 loops=1)
         ->  Async Foreign Scan on population_frpdl population_18
               (actual time=43.529..43.802 rows=1000 loops=1)
```

```
 Planning Time: 18.564 ms
 Execution Time: 4466.645 ms
```

On constate que les nœuds `Async Foreign Scan` se terminent presque au même instant (`actual time=26..`). Sur de hautes volumétries, les lectures asynchrones sont plus performantes, grâce à une répartition de travail entre les différentes instances, aussi appelées nœuds de calcul.

- Réexécuter la requête en désactivant l'option `async_capable` sur les partitions.

```
SELECT concat(
  'ALTER FOREIGN TABLE ', ftrelid::regclass,
  '  OPTIONS (ADD async_capable ''off'');'
) FROM pg_foreign_table;
\gexec

EXPLAIN (analyze, costs off)
 SELECT count(anon_id) FROM population
  WHERE date_naissance BETWEEN '2010-01-01' AND '2011-01-01';

                              QUERY PLAN
-----------------------------------------------------------------------------------
Aggregate (actual time=1986.016..1986.047 rows=1 loops=1)
   ->  Append (actual time=90.848..1982.710 rows=18000 loops=1)
         ->  Foreign Scan on population_frara population_1
               (actual time=90.844..112.269 rows=1000 loops=1)
         ->  Foreign Scan on population_frbfc population_2
               (actual time=72.032..93.117 rows=1000 loops=1)
...
         ->  Foreign Scan on population_frpac population_17
               (actual time=91.000..112.080 rows=1000 loops=1)
         ->  Foreign Scan on population_frpdl population_18
               (actual time=87.655..108.378 rows=1000 loops=1)
 Planning Time: 7.878 ms
 Execution Time: 6161.891 ms
```

Cette fois-ci, on constate un retard dans les lectures (`actual time=112..`) et que le nœud `Append` ne termine l'union des 18 résultats après 1986 millisecondes au lieu de 499 dans le cas d'une configuration asynchrone.

Pour réactiver les options sur les tables, exécuter la requête suivante :

```
SELECT concat(
  'ALTER FOREIGN TABLE ', ftrelid::regclass,
  '  OPTIONS (DROP async_capable);'
) FROM pg_foreign_table;
\gexec
```

On supprime la latence avec la commande suivante :

```
sudo tc qdisc del dev lo root
```

- Procéder à la mise à jour de la table `population` pour ajouter une date de décès à une portion de la population. Récupérer le nombre de décès survenus entre 1970 et 1980, regroupés par région.

```
EXPLAIN (analyze, verbose, costs off)
UPDATE population
   SET date_deces = date_naissance + trunc(random() * 70)::int
 WHERE anon_id::text like 'ff%'
   AND date_naissance < '1980-01-01';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Update on population (actual time=979.144..979.152 rows=0 loops=1)
  Foreign Update on public.population_frara population_1
   Remote SQL: UPDATE public.population SET date_deces = $2 WHERE ctid = $1
  ...
  Foreign Update on public.population_frpdl population_18
   Remote SQL: UPDATE public.population SET date_deces = $2 WHERE ctid = $1
   -> Append (actual time=137.350..1200.934 rows=701 loops=1)
      -> Foreign Scan on public.population_frara population_1
          (actual time=137.347..138.112 rows=37 loops=1)
           Filter: ((population_1.anon_id)::text ~~ 'ff%'::text)
           Rows Removed by Filter: 9963
           Remote SQL:
            SELECT anon_id, region_naissance_id, date_naissance, date_deces, ctid
              FROM public.population WHERE ((date_naissance < '1980-01-01'::date))
               FOR UPDATE
 ...
      -> Foreign Scan on public.population_frpdl population_18
          (actual time=38.566..38.676 rows=39 loops=1)
           Filter: ((population_18.anon_id)::text ~~ 'ff%'::text)
           Rows Removed by Filter: 9961
           Remote SQL:
            SELECT anon_id, region_naissance_id, date_naissance, date_deces, ctid
              FROM public.population WHERE ((date_naissance < '1980-01-01'::date))
               FOR UPDATE
 Planning Time: 2.370 ms
 Execution Time: 1312.767 ms
```

Lors d'une mise à jour des lignes, le `foreign data wrapper` implémente une prise de verrou `FOR UPDATE` avant la modification, comme le montre le plan d'exécution avec l'option `VERBOSE`. Cependant, cette étape ne parcourt pas les tables en asychrone bien que l'option soit activée. Le temps cumulé pour ces verrous représente une grande partie de l'exécution de la requête `UPDATE`, soit environ 1200 milliseconde sur les 1312 au total.

Pour une architecture distribuée minimale, les lectures sont grandement améliorées avec la nouvelle option `async_capable` sur des tables partitionnées. Pour les modifications, il convient de favoriser l'usage de la clé primaire distante pour garantir des temps de réponse optimum.

2.3 OUTIL PG_REWIND

- Création d'une instance primaire ;
- Mise en place de la réplication sur deux secondaires ;
- Promotion d'un secondaire pour réaliser des tests ;
- Utilisation de `pg_rewind` pour raccrocher l'instance secondaire à la réplication.

2.3.1 MISE EN PLACE DE L'ENVIRONNEMENT

- Ouvrir un terminal puis emprunter l'identité de l'utilisateur `postgres` sur votre machine.

```
sudo su - postgres
```

Pour cet atelier, nous créons des instances temporaires dans le répertoire `~/tmp/rewind` :

- Configurer la variable `DATADIRS`.

```
export DATADIRS=~/tmp/rewind
```

- Créer le répertoire `~/tmp/rewind` et le répertoire `~/tmp/rewind/archives`.

```
mkdir --parents ${DATADIRS}/archives
```

2.3.2 CRÉATION D'UNE INSTANCE PRIMAIRE

- Configurer les variables d'environnement pour l'instance à déployer.

```
export PGNAME=srv1
export PGDATA=$DATADIRS/$PGNAME
export PGPORT=5636
```

- Créer le répertoire `~/tmp/rewind/srv1`.

```
mkdir --parents ${DATADIRS}/${PGNAME}
```

- Créer une instance primaire dans le dossier `~/tmp/rewind/srv1` en activant les sommes de contrôle.

```
/usr/pgsql-14/bin/initdb --data-checksums --pgdata=${PGDATA} --username=postgres
```

> Pour utiliser `pg_rewind`, il est nécessaire d'activer le paramètre `wal_log_hints` dans le `postgresql.conf` ou les sommes de contrôles au niveau de l'instance.

- Configurer PostgreSQL.

```
cat <<_EOF_ >> ${PGDATA}/postgresql.conf
port = ${PGPORT}
listen_addresses = '*'
logging_collector = on
archive_mode = on
archive_command = '/usr/bin/rsync -a %p ${DATADIRS}/archives/%f'
restore_command = '/usr/bin/rsync -a ${DATADIRS}/archives/%f %p'
cluster_name = '${PGNAME}'
_EOF_
```

- Démarrer l'instance primaire.

```
/usr/pgsql-14/bin/pg_ctl start --pgdata=${PGDATA} --wait
```

- Créer une base de données `pgbench`.

```
psql --command="CREATE DATABASE pgbench;"
```

- Initialiser la base de données `pgbench` avec la commande **pgbench**.

```
/usr/pgsql-14/bin/pgbench --initialize --scale=10 pgbench
```

- Créer un utilisateur `replication`.

```
psql --command="CREATE ROLE replication WITH LOGIN REPLICATION PASSWORD 'replication';"
```

- Ajouter le mot de passe au fichier `.pgpass`.

```
cat << _EOF_ >> ~/.pgpass
*:5636:replication:replication:replication # srv1
*:5637:replication:replication:replication # srv2
*:5638:replication:replication:replication # srv3
_EOF_
chmod 600 ~/.pgpass
```

2.3.3 METTRE EN PLACE LA RÉPLICATION SUR DEUX SECONDAIRES

- Configurer les variables d'environnement pour l'instance à déployer.

```
export PGNAME=srv2
export PGDATA=${DATADIRS}/${PGNAME}
export PGPORT=5637
```

- Créer une instance secondaire à l'aide de l'outil `pg_basebackup`.

```
pg_basebackup --pgdata=${PGDATA} --port=5636 --progress --username=replication --checkpoint=fast
```

- Ajouter un fichier `standby.signal` dans le répertoire de données de l'instance **srv2**.

```
touch ${PGDATA}/standby.signal
```

- Modifier la configuration.

```
cat << _EOF_ >> ${PGDATA}/postgresql.conf
port = ${PGPORT}
primary_conninfo = 'port=5636 user=replication application_name=${PGNAME}'
cluster_name = '${PGNAME}'
_EOF_
```

- Démarrer l'instance secondaire.

```
/usr/pgsql-14/bin/pg_ctl start --pgdata=${PGDATA} --wait
```

La requête suivante doit renvoyer un nombre de lignes égal au nombre d'instances secondaires. Elle doit être exécutée depuis l'instance primaire **srv1** :

```
psql --port=5636 --expanded --command="SELECT * FROM pg_stat_replication;"
```

- Faire les mêmes opérations pour construire une troisième instance.

```
export PGNAME=srv3
export PGDATA=${DATADIRS}/${PGNAME}
export PGPORT=5638
```

2.3.4 DÉCROCHAGE VOLONTAIRE DE L'INSTANCE SECONDAIRE SRV3

- Faire un checkpoint sur l'instance **srv3**.

```
psql --port=5638 --command="CHECKPOINT;"
```

- Promouvoir l'instance secondaire **srv3**.

```
/usr/pgsql-14/bin/pg_ctl promote --pgdata=${DATADIRS}/srv3 --wait
```

- Ajouter des données aux instances **srv1** et **srv3** afin de les faire diverger (une minute d'attente par instance).

```
# Simulation d'une activité normale sur l'instance srv1
/usr/pgsql-14/bin/pgbench --port=5636 --client=10 --time=60 --no-vacuum pgbench
```

```
# Simulation d'une activité normale sur l'instance srv3
/usr/pgsql-14/bin/pgbench --port=5638 --client=10 --time=60 --no-vacuum pgbench
```

Les deux instances ont maintenant divergé. Sans action supplémentaire, il n'est donc pas possible de raccrocher l'ancienne instance secondaire **srv3** à l'instance primaire **srv1**.

- Stopper l'instance **srv3** proprement.

```
/usr/pgsql-14/bin/pg_ctl stop --pgdata=${DATADIRS}/srv3 --mode=fast --wait
```

2.3.5 UTILISATION DE PG_REWIND

- Donner les autorisations à l'utilisateur `replication` sur les fonctions `pg_ls_dir`, `pg_stat_file`, `pg_read_binary_file` et `pg_read_binary_file` du schéma `pg_catalog` afin qu'il puisse utiliser `pg_rewind`.

```
psql --port=5636 <<_EOF_
GRANT EXECUTE
  ON function pg_catalog.pg_ls_dir(text, boolean, boolean)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_stat_file(text, boolean)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_read_binary_file(text)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean)
  TO replication;
_EOF_
```

- Sauvegarder la configuration qui diffère entre **srv1** et **srv3** (ici `postgresql.conf`) car les fichiers de **srv1** vont écraser ceux de **srv3** pendant le *rewind*.

```
cp ${DATADIRS}/srv3/postgresql.conf ${DATADIRS}/postgresql.srv3.conf
```

- Utiliser `pg_rewind` pour reconstruire l'instance **srv3** depuis l'instance **srv2** (commencer par un passage à blanc `--dry-run`).

```
/usr/pgsql-14/bin/pg_rewind --target-pgdata ${DATADIRS}/srv3           \
          --source-server "port=5637 user=replication dbname=postgres" \
          --restore-target-wal                                         \
          --progress                                                   \
          --dry-run
```

Une fois le résultat validé, relancer `pg_rewind` sans `--dry-run`.

```
/usr/pgsql-14/bin/pg_rewind --target-pgdata ${DATADIRS}/srv3           \
          --source-server "port=5637 user=replication dbname=postgres" \
```

```
        --restore-target-wal                                    \
        --progress
```

- Restaurer le postgresql.conf de **srv3**.

```
cp ${DATADIRS}/postgresql.srv3.conf ${DATADIRS}/srv3/postgresql.conf
```

À l'issue de l'opération, les droits donnés à l'utilisateur de réplication peuvent être révoqués :

```
psql --port=5636 <<_EOF_
REVOKE EXECUTE
  ON function pg_catalog.pg_ls_dir(text, boolean, boolean)
  FROM  replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_stat_file(text, boolean)
  FROM replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_read_binary_file(text)
  FROM replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean)
  FROM replication;
_EOF_
```

2.3.6 REDÉMARRER SRV3

- Mettre à jour la configuration de **srv3** pour en faire une instance secondaire.

```
touch ${DATADIRS}/srv3/standby.signal

cat <<_EOF_ >> ${DATADIRS}/srv3/postgresql.conf
# Forcer la même timeline que l'instance principal pour la recovery
recovery_target_timeline = 1
_EOF_
```

- Redémarrer l'instance **srv3**.

```
/usr/pgsql-14/bin/pg_ctl start --pgdata=${DATADIRS}/srv3 --wait
```

La requête suivante doit renvoyer un nombre de lignes égal au nombre d'instances secondaires. Elle doit être exécutée depuis l'instance primaire **srv1** :

```
psql --port=5636 --expanded --command="SELECT * FROM pg_stat_replication;"
```

2.3.7 REMARQUES

Commenter le paramètre `recovery_target_timeline` de la configuration de l'instance **srv3**, car elle pourrait poser des problèmes par la suite.

Avec la procédure décrite dans cet atelier, le serveur **srv3** archive dans le même répertoire que le serveur **srv1**. Il serait préférable d'archiver dans un répertoire différent. Cela introduit de la complexité. En effet, `pg_rewind` aura besoin des *WAL* avant la divergence (répertoire de **srv3**) et ceux générés depuis le dernier *checkpoint* précédant la divergence (répertoire de **srv1**).
