Workshop 18

Nouveautés de PostgreSQL 18

18

Contents

1/ Introduction

- Développement depuis juin 2024
- Version finale : Septembre 2025
- Actuellement en v 18.3
- Des centaines de contributeurs

Le développement de la version 18 a suivi l'organisation habituelle : un démarrage en juin 2024, des *Commit Fests* tous les deux mois jusqu'en mars 2025, un *feature freeze*, 3 versions bêta, 1 version RC, et enfin la GA.

La version finale est parue le 25 septembre 2025. Plusieurs versions correctives sont sorties depuis. Au moment où ceci est écrit, nous en sommes à la version 18.3.

Son développement est assuré par des centaines de contributeurs répartis partout dans le monde.

2/ Utilisation

2.1 GESTION DES UUID V7

- Fonction de génération des UUID v7 : `uuidv7()`
  - début généré à partir d'un timestamp
  - assure la colocalité des données dans les index B-tree
- Fonctions d'extraction :
  - `uuid_extract_version()`
  - `uuid_extract_timestamp()`

Un UUID (*Universally Unique IDentifier*) est un nombre encodé sur 128 bits dont la génération assure son unicité au sein d'un système.

Il y a plusieurs implémentations des UUID, identifiées par leur numéro de version :

- les versions 1 et 6 se basent sur l'adresse MAC et un timestamp. L'ordre des bits de la version 6 permet de trier les UUID par date ;
- la version 2 se base aussi sur une adresse MAC et un timestamp mais ajoute le domaine local dans le calcul. Ce type d'UUID est souvent omis dans les librairies qui implémentent les UUID ;
- les versions 3 et 5 se basent sur le hachage d'un identifiant de namespace et un nom ;
- la version 4 utilise une génération aléatoire ;
- la version 7 est destinée à être utilisée dans les bases de données et systèmes distribués. Elle est générée avec un timestamp et une partie aléatoire ;
- la version 8 prévoit une spécification pour créer des UUID spécifiques à une application.

L'extension `uuid-ossp` fournit les fonctions de génération pour les versions 1, 3, 4 et 5.

```
CREATE EXTENSION "uuid-ossp";
\dx+ "uuid-ossp"

   Objects in extension "uuid-ossp"
         Object description
--------------------------------------
 function uuid_generate_v1()
 function uuid_generate_v1mc()
 function uuid_generate_v3(uuid,text)
 function uuid_generate_v4()
 function uuid_generate_v5(uuid,text)
 function uuid_nil()
 function uuid_ns_dns()
```

```
 function uuid_ns_oid()
 function uuid_ns_url()
 function uuid_ns_x500()
(10 rows)
```

Des fonctions sont également disponibles dans PostgreSQL pour les UUID en version 4 et 7. La version 17 avait commencé à préparer le terrain pour l'ajout des UUID v7 en créant un alias à la fonction `gen_random_uuid()` nommé `uuidv4()`. La version 18 intègre la fonction `uuidv7()`.

La différence entre les deux est visible au premier coup d'œil :

```
SELECT uuidv4(), uuidv7() FROM generate_series(1,5);
```

```
                uuidv4                |                uuidv7
--------------------------------------+--------------------------------------
 b300652d-4f26-482a-a314-22d2d7e8aad2 | 01999604-2ba2-7ea9-853e-657bbcd7596e
 aa0b526d-2893-4418-96f0-80e6d7f82fa6 | 01999604-2ba2-7ec9-b8c3-fd99c01e6447
 bda1ddcf-5c60-4ee6-a0bc-2d2b86881395 | 01999604-2ba2-7ed4-9194-a63b806d4e79
 b6cde31b-5378-4760-99ef-9fa91c69c104 | 01999604-2ba2-7edd-97e4-196d8638b868
 697ec82c-a73a-4e39-bfaf-71d9fd88a9f3 | 01999604-2ba2-7ee6-918c-5503cfa8f647
(5 rows)
```

On voit que les UUID V7 peuvent être triés par ordre de création.

Des fonctions d'extractions sont également fournies pour les UUID, quelle que soit la fonction utilisée pour les générer :

```
SELECT uuid_extract_version(x), uuid_extract_timestamp(x)
FROM (VALUES
        (uuid_generate_v1()),
        (uuid_generate_v3(uuid_generate_v1 (), 'postgres')),
        (uuid_generate_v4()),
        (uuidv4()),
        (uuid_generate_v5(uuid_generate_v1 (), 'postgres')),
        (uuidv7())
) AS F(x);
```

```
 uuid_extract_version |    uuid_extract_timestamp
----------------------+------------------------------
                    1 | 2025-09-29 17:55:11.98522+02
                    3 | ø
                    4 | ø
                    4 | ø
                    5 | ø
                    7 | 2025-09-29 17:55:48.267+02
(6 rows)
```

Les UUID ont très mauvaise réputation dans le monde des bases de données L'inconvénient principal des UUID v4 est l'impact désastreux sur la colocalité des données dans les index B-tree.

L'exemple suivant crée deux tables, avec chacune un index sur une colonne de type `uuid`, généré avec des UUID de type v4 pour l'une et v7 pour l'autre.

```
-- UUID v4
DROP TABLE IF EXISTS tuuid_v4;
CREATE TABLE tuuid_v4(i uuid, t text);
CREATE INDEX ON tuuid_v4(i);
INSERT INTO tuuid_v4 SELECT uuidv4(), 'id ' || x FROM generate_series(1,1000000) AS
 ↪  F(x);

-- UUID v7
DROP TABLE IF EXISTS tuuid_v4;
CREATE TABLE tuuid_v7(i uuid, t text);
CREATE INDEX ON tuuid_v7(i);
INSERT INTO tuuid_v7 SELECT uuidv7(), 'id ' || x FROM generate_series(1,1000000) AS
 ↪  F(x);

VACUUM ANALYZE tuuid_v4, tuuid_v4;
```

On constate que l'index sur l'UUID v4 est nettement plus volumineux que celui avec un UUID v7.

```
\di+ tuuid*
```

```
                                    List of indexes
 Schema |      Name      | Type  | Owner  |  Table   |…| Access method | Size  |…
--------+----------------+-------+--------+----------+…+---------------+-------+…
 public | tuuid_v4_i_idx | index | benoit | tuuid_v4 |…| btree         | 38 MB |…
 public | tuuid_v7_i_idx | index | benoit | tuuid_v7 |…| btree         | 30 MB |…
(2 rows)
```

Le nombre de pages dans l'index sur les UUID v4 est beaucoup plus important. Cela s'explique par le fait qu'avec ce genre d'UUID, les lignes sont souvent insérées dans des pages différentes.

Par contre, les UUID v7 sont triés grâce au timestamp, et insérés les uns après les autres dans les pages. Cet ordre entraîne une utilisation plus efficace des pages et donc du cache.

```
SELECT relname, relpages
  FROM pg_class
 WHERE relname IN ('tuuid_v4_i_idx', 'tuuid_v7_i_idx');
```

```
    relname     | relpages
----------------+----------
 tuuid_v7_i_idx |     3853
 tuuid_v4_i_idx |     4866
(2 rows)
```

Le coût de mise en place des UUID v4 est mineur à côté des gros soucis qu'ils posent dans la gestion du cache. En effet, deux UUID générés à la suite seront probablement dans des blocs d'index différents, à l'inverse des séquences habituelles où les numéros se suivent. Les UUID v7 corrigent cela.

Les exemples suivants montrent l'impact de la colocalité des données sur l'utilisation du cache en sélectionnant les deux premières lignes insérées dans chaque table par leur UUID. On peut observer que pour la table `tuuid_v4` cette sélection nécessite d'accéder 7 pages, contre 4 pour la table `tuuid_v7`.

```
SELECT * FROM tuuid_v4 LIMIT 2 ;
```

```
                  i                   |  t
--------------------------------------+------
 1ae3654b-7209-4977-bea1-e78694b3935e | id 1
 2f9d0709-87fe-4fec-ad4b-76eb89008f5b | id 2
```

```
--- UUID v4
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT *
  FROM tuuid_v4
 WHERE i IN ('1ae3654b-7209-4977-bea1-e78694b3935e',
  ↪  '2f9d0709-87fe-4fec-ad4b-76eb89008f5b');
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using tuuid_v4_i_idx on tuuid_v4 (actual rows=2.00 loops=1)
   Index Cond: (i = ANY ('{1ae3654b-7209-4977-bea1-e78694b3935e,
                           2f9d0709-87fe-4fec-ad4b-76eb89008f5b}'::uuid[]))
   Index Searches: 2
   Buffers: shared hit=7
```

```
SELECT * FROM tuuid_v7 LIMIT 2 ;
```

```
--- UUID v7
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT *
  FROM tuuid_v7
 WHERE i IN ('01999644-0a57-7a3a-bc85-b67dcc3061bf',
  ↪  '01999644-0a57-7ed0-ac2d-72a1036387f7');
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using tuuid_v7_i_idx on tuuid_v7 (actual rows=2.00 loops=1)
   Index Cond: (i = ANY ('{01999644-0a57-7a3a-bc85-b67dcc3061bf,
                           01999644-0a57-7ed0-ac2d-72a1036387f7}'::uuid[]))
   Index Searches: 1
   Buffers: shared hit=4
```

L'impact semble mineur, mais l'effet sur le cache est radical pour des appels répétés : par exemple, un index de clé primaire en UUID v4 aura tendance à se retrouver intégralement dans le cache de PostgreSQL, même si l'essentiel des lignes ne sont pas utilisées (voir cet exemple dans notre formation

PERF2[1]).

Pour les versions précédentes de PostgreSQL, il existe des scripts ou extensions pour générer des versions UUID v7.

Les UUID v7 contiennent leur date de génération : utilisés comme clé primaire, ils peuvent éviter l'ajout d'un champ `creation_date`. Pour la même raison, ils peuvent même faciliter le partitionnement temporel[2].

Pour d'autres détails sur les UUID, voir aussi notre module de formation S22[3].

---

[1] https://dali.bo/s22_html#uuid-2
[2] https://fljd.in/2025/06/17/le-partitionnement-par-uuid-v7/
[3] https://dali.bo/s22_html#uuid

2.2 COLONNES GÉNÉRÉES VIRTUELLES : PRÉSENTATION

```
ALTER TABLE paquets
ADD COLUMN volume int GENERATED ALWAYS
        AS ((longueur * hauteur * largeur))  VIRTUAL ;
```

- Les colonnes générées **virtuelles**
  - sont recalculées à chaque appel
  - ne prennent pas de place
  - ne nécessitent pas une réécriture de la table
  - sont facilement modifiables

Rappel sur les colonnes générées stockées :

Les colonnes générées *stockées* existent depuis PostgreSQL 13. Dans cet exemple, on rajoute une colonne `volume` calculée à partir de trois autres.

```
DROP TABLE IF EXISTS paquets ;
CREATE TABLE paquets (  id int PRIMARY KEY,
                        longueur    int,
                        hauteur     int,
                        largeur     int,
                        t           timestamptz,
                        filler      char(50) DEFAULT ' '
                        ) ;

INSERT INTO paquets
SELECT i, 1+mod(i,89), 1+mod(i,97), 1+mod(i,83),
'2006-01-01'::timestamptz+i*interval '10m'
FROM generate_series (1, 1_000_000) i ;

ALTER TABLE paquets
ADD COLUMN volume int GENERATED ALWAYS AS ((longueur * hauteur * largeur))  STORED;
Temps : 420,294 ms

\d paquets
                      Table « public.paquets »
 Colonne  |      Type       | C. | NULL-able |       Par défaut
----------+-----------------+----+-----------+---------------------------------
 id       | integer         |    | not null  |
 longueur | integer         |    |           |
 hauteur  | integer         |    |           |
```

```
 largeur  | integer         |    |           |
 filler   | character(50)   |    |           | ' '::bpchar
 volume   | integer         |    |           | generated always as
                                               (longueur * hauteur * largeur)
                                               stored
Index :
    "paquets_pkey" PRIMARY KEY, btree (id)
```

La colonne générée stockée `volume` est en lecture seule.

Elle est calculée automatiquement à l'insertion, et recalculée automatiquement si les champs dont elle dépend sont mis à jour. Elle se comporte comme une colonne normale, peut porter une contrainte, et possède ses statistiques propres dans `pg_stats`. On peut l'indexer, ce qui peut être très pratique.

Une sauvegarde logique avec `pg_dump` ne contient pas le contenu de la colonne générée, ce sera recalculé à la restauration. De même, la colonne générée stockée n'est exportée par la réplication logique que depuis PostgreSQL 18, et sur demande, pour les cas où la cible ne serait pas PostgreSQL.

Noter que l'ajout d'une colonne générée stockée nécessite une réécriture complète de la table, ce qui est très lourd. Il n'est possible de modifier l'expression sans la supprimer d'abord que depuis PostgreSQL 17, et cela exige de réécrire la table. Il est possible d'utiliser des fonctions définies par l'utilisateur, mais en cas de modification de la fonction, il faut penser à réécrire la table (ce n'est pas automatique).

Principes des colonnes générées virtuelles :

PostgreSQL 18 apporte les colonnes générées *virtuelles*, qui se définissent de la même manière, mais avec le mot clé `VIRTUAL` à la fin :

```
ALTER TABLE paquets DROP COLUMN volume;

ALTER TABLE paquets
ADD COLUMN volume int
GENERATED ALWAYS AS (longueur*hauteur*largeur) VIRTUAL ;

\d paquets

                        Table « public.paquets »
 Colonne  |      Type       | C | NULL-able |        Par défaut
----------+-----------------+---+-----------+---------------------------
 id       | integer         |   | not null  |
 longueur | integer         |   |           |
 hauteur  | integer         |   |           |
 largeur  | integer         |   |           |
 t        | timestamp with… |   |           |
 filler   | character(50)   |   |           | ' '::bpchar
```

```
 volume   | integer         |   |          | generated always as
                                             (longueur * hauteur * largeur)
```

VIRTUAL n'apparaît pas dans la description et peut être omis, car c'est à présent le défaut pour une colonne générée. Par clarté, il vaut mieux écrire VIRTUAL en toute lettre à la fin. Il n'y a pas de risque pour les scripts existants, car le mot-clé STORED était obligatoire.

Évolution des colonnes virtuelles :

La table n'a pas besoin d'être réécrite pour ajouter une colonne virtuelle, le verrou est donc bien plus court.

Comme pour une colonne générée stockée, on peut poser une contrainte. Sa création impose la lecture de la table pour vérification, ce peut être long. Modifier une colonne source violant cette contrainte bloquera une insertion :

```
ALTER TABLE paquets ADD CONSTRAINT vol_ck CHECK (volume>0);

INSERT INTO paquets (id, largeur, longueur, hauteur)
VALUES (2_000_000, 0, 1, 1) ;
ERROR:  new row for relation "paquets" violates check constraint "vol_ck"
DÉTAIL : Failing row contains (100100, 1, 1, 0,..., virtual).
```

Si la fonction à calculer change, la modification est facile et ne pose pas de souci de cohérence.

```
ALTER TABLE paquets DROP CONSTRAINT vol_ck ;

ALTER TABLE paquets ALTER COLUMN volume
  SET EXPRESSION AS (10*ceil (longueur*hauteur*largeur/10+1)::int) ;
```

La suppression préalable de la contrainte existante est nécessaire. En effet, PostgreSQL ne vérifie pas de lui-même qu'une contrainte existante restera valide, et il refuse la modification avec un message clair :

```
ERROR:  ALTER TABLE / SET EXPRESSION is not supported for virtual generated columns
↪  in tables with check constraint
```

La contrainte devra être redéfinie ensuite, si pertinente.

Un changement de type est trivial, mais peut exiger une relecture de la table :

```
ALTER TABLE paquets ALTER COLUMN volume TYPE bigint ;
ALTER TABLE paquets ALTER COLUMN volume TYPE int ;
```

Supprimer l'expression (ALTER TABLE … ALTER COLUMN … DROP EXPRESSION ;) n'a pas de sens pour une colonne générée virtuelle, il faut supprimer la colonne. (À l'inverse, pour une colonne générée stockée, DROP EXPRESSION permet de conserver les valeurs déjà écrites.)

2.3 COLONNES GÉNÉRÉES VIRTUELLES : LIMITES

- Limites sur les colonnes utilisables
- Limites sur les fonctions utilisables
- Limites sur l'indexation
- Limites sur les statistiques
- Limites sur les clés étrangères
- Limites sur la réplication logique

Limitations fondamentales des colonnes virtuelles :

Comme pour les colonnes générées stockées, les seuls champs utilisables dans l'expression d'une colonne virtuelle sont ceux présents sur la ligne.

Il n'est malheureusement pas possible d'utiliser une colonne virtuelle comme source d'une autre colonne virtuelle :

```
ALTER TABLE paquets ADD COLUMN tropgros boolean
    GENERATED ALWAYS AS ( volume > 800_000 ) VIRTUAL  ;

ERROR:  cannot use generated column "volume" in column generation expression
DÉTAIL : A generated column cannot reference another generated column.
```

Comme pour les colonnes générées stockées, les fonctions utilisées doivent être « immutables », c'est-à-dire ne pas dépendre d'autre chose que des champs de la ligne. Ces fonctions ne peuvent pas utiliser d'autres table, ni des paramètres de session, directement ou indirectement, comme le fuseau horaire ici :

```
ALTER TABLE paquets ADD COLUMN annee text
  GENERATED ALWAYS AS (extract('year' FROM t )) VIRTUAL ;
ERROR:  generation expression is not immutable
```

`extract` avec un `timestamp` (sans fuseau) est par contre immutable :

```
ALTER TABLE paquets ADD COLUMN annee_paris text
  GENERATED ALWAYS AS (extract('year' FROM t AS TIME ZONE 'Europe/Paris')) VIRTUAL ;
```

Limitation des fonctions utilisables des colonnes virtuelles :

Pendant le développement de PostgreSQL 18, il était prévu que les colonnes virtuelles puissent utiliser des fonctions utilisateurs. Ce n'est finalement pas possible pour des raisons de sécurité (un superutilisateur pourrait être amené à exécuter des fonctions alors qu'il croit juste consulter une table). On est

donc limité à l'ensemble des fonctions que connaît déjà PostgreSQL, et bien sûr uniquement celles immutables. Ceci ne fonctionnera donc pas :

```
CREATE OR REPLACE FUNCTION volume (l int, h int, p int)
RETURNS int
AS $$
  SELECT 10*ceil (l * h * p/10+1) ;
$$ LANGUAGE sql STRICT IMMUTABLE PARALLEL SAFE ;

ALTER TABLE paquets ALTER COLUMN volume
  SET EXPRESSION AS (volume (longueur, hauteur, largeur)) ;

ERROR:  generation expression uses user-defined function
DÉTAIL : Virtual generated columns that make use of user-defined functions are not
 ↪  yet supported.
```

Cela aurait pourtant été pratique pour des fonctions complexes comme celle ci-dessus, ou déjà existantes.

Limitation sur l'indexation de colonnes virtuelles :

Les colonnes virtuelles ne peuvent pas non plus être indexées. Cela entrerait en conflit avec leur principe de base, qui est d'être légères et facilement modifiables[4]

Cependant, pour éviter de devoir créer une colonne générée stockée et de réécrire la table, il est possible de contourner en créant un index fonctionnel **exactement identique à la formule**. Il faut maintenir les deux définitions en parallèle :

```
CREATE INDEX paquets_vol_idx
ON paquets ((10*ceil (longueur*hauteur*largeur/10+1)::int)) ;

EXPLAIN (costs off) SELECT id,volume FROM paquets WHERE volume > 950_000 ;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on paquets
   Recheck Cond: ((10 * (ceil((((((longueur * hauteur) * largeur) / 10) + 1))::double
 ↪  precision))::integer) > 950000)
   ->  Bitmap Index Scan on paquets_vol_idx
         Index Cond: ((10 * (ceil((((((longueur * hauteur) * largeur) / 10) +
 ↪  1))::double precision))::integer) > 950000)
```

Noter que `volume` est toujours remplacé par son expression complète.

Limitation sur les statistiques des colonnes virtuelles :

[4] https://www.postgresql.org/message-id/1178617.1753217685%40sss.pgh.pa.us

Plus gênant, la colonne `volume` ne possède pas de statistiques dans `pg_stats`. PostgreSQL ne sait donc pas estimer la volumétrie renvoyée par un prédicat sur une colonne virtuelle. Dans cet exemple, PostgreSQL croit ramener un tiers de la table au lieu de quelques lignes :

```
EXPLAIN (ANALYZE,BUFFERS OFF) SELECT id, volume FROM paquets WHERE volume > 950_000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on paquets  (cost=3807.76..35633.74 rows=333333 width=8) (actual
↪  time=0.034..0.034 rows=0.00 loops=1)
   Recheck Cond: ((10 * (ceil((((((longueur * hauteur) * largeur) / 10) + 1))::double
↪  precision))::integer) > 950000)
   ->  Bitmap Index Scan on paquets_vol_idx  (cost=0.00..3724.42 rows=333333 width=0)
↪  (actual time=0.023..0.024 rows=0.00 loops=1)
         Index Cond: ((10 * (ceil((((((longueur * hauteur) * largeur) / 10) +
↪  1))::double precision))::integer) > 950000)
         Index Searches: 1
 Planning Time: 0.221 ms
 Execution Time: 0.074 ms
```

Et un objet statistique explicite sur la colonne virtuelle n'est pas accepté :

```
CREATE STATISTICS paquets_volume_stats
ON (volume) FROM paquets ;
ERROR:  statistics creation on virtual generated columns is not supported
```

Là encore, il est possible de contourner en demandant la définition d'un objet statistique avec l'expression exacte du champ calculée :

```
CREATE STATISTICS paquets_volume_stats
ON (10*ceil (longueur*hauteur*largeur/10+1)::int)
FROM paquets ;
ANALYZE paquets ;
```

Le planificateur reconnaît l'expression et sait l'utiliser :

```
EXPLAIN (ANALYZE,BUFFERS OFF) SELECT * FROM paquets WHERE volume > 950_000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using paquets_vol_idx on paquets  (cost=0.42..8.47 rows=1 width=119)
↪  (actual time=0.005..0.005 rows=0.00 loops=1)
   Index Cond: ((10 * (ceil((((((longueur * hauteur) * largeur) / 10) + 1))::double
↪  precision))::integer) > 950000)
   Index Searches: 1
 Planning Time: 0.294 ms
 Execution Time: 0.033 ms
```

Limitation sur les clés étrangères :

Il n'est pas possible de poser une contrainte de clé étrangère sur une colonne virtuelle. Comme elle n'est pas indexable, cette même colonne ne peut être déclarée comme cible d'une clé étrangère depuis une autre table. Cela serait possible avec des colonnes calculées stockées.

Certaines de ces limites ont des patchs en développement pour PostgreSQL 19. L'avenir dira s'ils sont acceptés.

Réplication logique et colonnes virtuelles

Les colonnes virtuelles ne sont pas répliquées. Elles peuvent être ajoutées à la table d'origine ou cible sans impact sur la réplication.

Même le paramètre des publications `publish_generated_columns`, apparu aussi avec PostgreSQL 18, n'a pas d'influence (il ne vaut que `none` ou `generated`, et ce dernier ne permet de répliquer que les colonnes stockées).

Si l'on tente de forcer la publication d'une colonne virtuelle en publiant la liste explicite des colonnes (ce qui fonctionne avec des colonnes générées stockées), on obtient une erreur :

```
CREATE TABLE t2 (i int, v int GENERATED ALWAYS AS (i+1) VIRTUAL) ;
CREATE PUBLICATION pub_t2 FOR TABLE t2 (i,v);
```

```
ERROR:  cannot use virtual generated column "v" in publication column list
```

2.4 COMPARAISON ENTRE COLONNES GÉNÉRÉES STOCKÉES ET VIRTUELLES

Critère	Stockées	Virtuelles
PostgreSQL minimum	12	18
Fonctions utilisables	Immutables	Immutable et système (pas d'utilisateur)
Fonctions utilisateur	Oui	Non
Place sur le disque	Oui	Aucune
Réécriture de la table	Création + modification	Non
Statistiques	Oui	Non (*)
Indexables	Oui	Non (*)
Clés étrangères	Oui	Non
Réplication logique	>=18 (sur demande)	Jamais

(*) sauf indirectement via la définition d'autres objets

Le tableau précédent reprend les principales différences entre colonnes générées stockées et virtuelles.

2.5 CLÉ TEMPORELLES

- Interdit les recouvrements de période

```
ALTER TABLE reservations
ADD CONSTRAINT periodes_ne_se_recouvrent_pas
UNIQUE (salle, periode WITHOUT OVERLAPS);
```

- Clés primaires et étrangères possibles
- Extension `btree_gist` généralement nécessaire
- Était déjà partiellement possible avec des contraintes d'exclusion

Les clés temporelles permettent de poser des contraintes sur des périodes de temps qui ne doivent pas se recouvrir.

Cette fonctionnalité était présente en partie et depuis longtemps dans PostgreSQL, via les contraintes d'exclusion. Les clés temporelles permettent de poser des contraintes d'unicité à proprement parler, et même de vraies clés primaires. Cela autorise des clés étrangères pointant sur cette clé, et facilite la réplication logique, sans créer d'autre clé.

En pratique, l'extension `btree_gist` est nécessaire.

Dans l'exemple ci-dessous, la contrainte interdit qu'une salle soit réservée sur des plages de temps qui se recoupent :

```
CREATE EXTENSION IF NOT EXISTS btree_gist ;

CREATE TABLE reservations
(
  salle      text,
  professeur text,
  periode    tstzrange -- période des cours
);

-- Clé primaire : pour une même salle,
-- les réservations ne doivent pas se recouvrir

ALTER TABLE reservations ADD CONSTRAINT reservations_pk
PRIMARY KEY (salle, periode WITHOUT OVERLAPS);

\d reservations

                  Table « public.reservations »
```

```
  Colonne   |   Type    | Collationnement | NULL-able | Par défaut | …
------------+-----------+-----------------+-----------+------------+ …
 salle      | text      |                 | not null  |            | …
 professeur | text      |                 |           |            | …
 periode    | tstzrange |                 | not null  |            | …
Index :
    "reservations_pk" PRIMARY KEY (salle, periode WITHOUT OVERLAPS)


-- Deux réservations successives dans la même salle

INSERT INTO reservations (professeur,salle,periode) VALUES
( 'Marc', 'salle techno', '[2026-06-16 09:00:00, 2026-06-16 10:00:00)');

INSERT INTO reservations (professeur,salle,periode) VALUES
( 'Guillaume', 'salle techno', '[2026-06-16 10:00:00, 2026-06-16 11:00:00)');


-- Réservation dans une autre salle au même moment

INSERT INTO reservations (professeur,salle,periode) VALUES
( 'Jean', 'salle informatique', '[2026-06-16 10:00:00, 2026-06-16 11:00:00)');

TABLE reservations ;

       salle        | professeur |                       periode
--------------------+------------+-----------------------------------------------------
 ↪  ---
 salle techno       | Marc       | ["2026-06-16 09:00:00+02","2026-06-16 10:00:00+02")
 salle techno       | Guillaume  | ["2026-06-16 10:00:00+02","2026-06-16 11:00:00+02")
 salle informatique | Jean       | ["2026-06-16 10:00:00+02","2026-06-16
 ↪  11:00:00+02")



-- Refus de réserver une 2è fois dans la même salle
-- avec une période qui recouvre même partiellement

INSERT INTO reservations (professeur,salle,periode) VALUES
( 'Damien', 'salle techno', '[2026-06-16 10:30:00, 2026-06-16 11:00:00)');

ERROR:  conflicting key value violates exclusion constraint
 ↪  "periodes_ne_se_recouvrent_pas"
DÉTAIL : Key (salle, periode)=(salle techno, ["2026-06-16 10:30:00+02","2026-06-16
 ↪  11:00:00+02")) conflicts with existing key (salle, periode)=(salle techno,
 ↪  ["2026-06-16 10:00:00+02","2026-06-16 11:00:00+02")).
```

```
-- Table des inscriptions des élèves aux cours
-- La clé primaire est similaire.
-- La clé étrangère pointe sur la réservation

CREATE TABLE eleves (eleve text,
                     salle text,
                     periode tstzrange,
    PRIMARY KEY (eleve, salle, periode WITHOUT OVERLAPS),
    FOREIGN KEY (salle, PERIOD periode) REFERENCES reservations
    ) ;

\d eleves

                 Table « public.eleves »
 Colonne |   Type    | Collationnement | NULL-able | Par défaut
---------+-----------+-----------------+-----------+------------
 eleve   | text      |                 | not null  |
 salle   | text      |                 | not null  |
 periode | tstzrange |                 | not null  |
Index :
    "eleves_pkey" PRIMARY KEY (eleve, salle, periode WITHOUT OVERLAPS)
Contraintes de clés étrangères :
    "eleves_salle_periode_fkey" FOREIGN KEY (salle, PERIOD periode) REFERENCES
 ↪  reservations(salle, PERIOD periode)

-- Jonathan et Lucas vont en cours

INSERT INTO eleves (eleve,salle,periode) VALUES
( 'Lucas', 'salle informatique', '[2026-06-16 10:00:00, 2026-06-16 11:00:00)');
INSERT INTO eleves (eleve,salle,periode) VALUES
( 'Jonathan', 'salle informatique', '[2026-06-16 10:00:00, 2026-06-16 11:00:00)');


-- Pas d'inscription en double

INSERT INTO eleves (eleve,salle,periode) VALUES
( 'Jonathan', 'salle informatique', '[2026-06-16 10:00:00, 2026-06-16 11:00:00)');

ERROR:  conflicting key value violates exclusion constraint "eleves_pkey"
DÉTAIL : Key (eleve, salle, periode)=(Jonathan, salle informatique, ["2026-06-16
 ↪  10:00:00+02","2026-06-16 11:00:00+02")) conflicts with existing key (eleve,
 ↪  salle, periode)=(Jonathan, salle informatique, ["2026-06-16
 ↪  10:00:00+02","2026-06-16 11:00:00+02")).

-- Des périodes de taille inférieure comprises dans celles de la clé
-- sont possibles.
```

```
-- Léa peut s'absenter au milieu du cours.

INSERT INTO eleves (eleve,salle,periode) VALUES
('Léa', 'salle informatique', '[2026-06-16 10:00:00, 2026-06-16 10:15:00)');

INSERT INTO eleves (eleve,salle,periode) VALUES
('Léa', 'salle informatique', '[2026-06-16 10:45:00, 2026-06-16 11:00:00)');
```

Pour mémoire, la version avec des contraintes d'exclusion fonctionne ainsi pour les versions antérieures à PostgreSQL 18 :

```
CREATE EXTENSION IF NOT EXISTS btree_gist ;

CREATE TABLE reservations
( salle      TEXT,
  professeur TEXT,
  periode    tstzrange );

ALTER TABLE reservations
ADD CONSTRAINT periodes_ne_se_recouvrent_pas
EXCLUDE USING gist (salle WITH =,periode WITH &&);
```

Le fonctionnement à l'insertion dans la table est identique, mais une clé primaire ou étrangère n'est pas possible.

2.6 AJOUT DES PSEUDO TABLES OLD/NEW POUR RETURNING

- Clause RETURNING disponible depuis la 8.2
- Mais manquait la possibilité d'accéder aux anciennes valeurs
- Manque corrigé en v18
- Pseudo table OLD pour les anciennes valeurs
- Pseudo table NEW pour les nouvelles valeurs

Considérons la table t1 :

```
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1 VALUES (1, 'un'), (2, 'deux'), (3, 'trois');
```

Auparavant, la clause RETURNING d'un UPDATE ne permettait de ne renvoyer que la nouvelle valeur :

```
UPDATE t1 SET c1=c1+1 WHERE c1>1
RETURNING c1;
```

```
 c1
----
  3
  4
(2 rows)
```

À partir de la version 18, il est possible d'accéder aux valeurs anciennes et nouvelles :

```
UPDATE t1 SET c1=c1+1 WHERE c1>1
RETURNING old.c1, new.c1;
```

```
 c1 | c1
----+----
  3 |  4
  4 |  5
(2 rows)
```

Les pseudo-tables `OLD` et `NEW` sont utilisables pour toutes les requêtes DML, donc `INSERT`, `UPDATE`, `DELETE`, `MERGE`.

2.7 AMÉLIORATIONS DE COPY

- Nouvelle option `reject_limit`
- Nouvelle valeur `silent` pour l'option `log_verbosity`
- Possibilité de copier à partir de vues matérialisées

Nombre d'erreurs à l'import avec COPY :

Il est maintenant possible d'indiquer un nombre maximum d'erreurs acceptable lorsque des lignes sont ignorées parce que certaines valeurs n'ont pas le bon type de données. Ce nombre doit être indiqué à l'option `reject_limit`. Sans indication, le comportement est identique aux anciennes versions : aucune limite sur le nombre de lignes ignorées. Voici un exemple complet :

```
demo =# \! cat test.csv
1
2
a
3
b
4
c
5

demo =# create table t1 (c1 integer);
CREATE TABLE

demo =# \copy t1 from 'test.csv' (format 'csv')
ERROR:  invalid input syntax for type integer: "a"
CONTEXT:  COPY t1, line 3, column c1: "a"
```

Le fichier contient des entiers et des caractères, il ne convient pas pour une colonne de type `integer`, l'opération est entièrement annulée. En demandant d'ignorer les lignes avec des erreurs sur les types de données, la copie se passe bien :

```
demo =# \copy t1 from 'test.csv' (format 'csv', on_error ignore)
NOTICE:  3 rows were skipped due to data type incompatibility
```

```
COPY 5

demo =# select * from t1;
┌────┐
│ c1 │
├────┤
│  1 │
│  2 │
│  3 │
│  4 │
│  5 │
└────┘
(5 rows)
```

Trois lignes sont ignorées car le type de données est mauvais : normal. Toutes les lignes sans erreur sont insérées.

Maintenant, vidons la table et ajoutons une limite de deux erreurs :

```
demo =# truncate t1;
TRUNCATE TABLE

demo =# \copy t1 from 'test.csv' (format 'csv',on_error ignore,reject_limit 2)
ERROR:  skipped more than REJECT_LIMIT (2) rows due to data type incompatibility
CONTEXT:  COPY t1, line 7, column c1: "c"

demo =# select * from t1;
┌────┐
│ c1 │
├────┤
└────┘
(0 rows)
```

Cette fois, l'opération échoue parce que le nombre d'erreurs dépasse la limite fixée. Changeons la limite à 5 :

```
demo =# \copy t1 from 'test.csv' (format 'csv',on_error ignore,reject_limit 5)
NOTICE:  3 rows were skipped due to data type incompatibility
COPY 5
```

Cette fois, l'opération réussit.

Supprimer l'affichage des erreurs d'import :

Cette version 18 permet aussi de supprimer l'affichage des erreurs. Par exemple :

```
demo =# \copy t1 from 'test.csv'(format 'csv',on_error ignore,log_verbosity silent)
COPY 5
```

Trois lignes sont ignorées car en erreur mais aucun message ne l'indique vu que nous avons utilisé l'option `log_verbosity`.

Exporter le contenu d'une vue matérialisée :

Enfin, une petite limite de COPY est levée. En version 17 encore, il n'était pas possible de copier directement depuis une vue matérialisée :

```
demo =# create table t1 AS VALUES ('42') ;
CREATE TABLE

demo =# create materialized view mv1 as select * from t1;
SELECT 1

demo =# copy mv1 to '/tmp/mv1.csv' with (format csv);
ERROR:  cannot copy from materialized view "mv1"
HINT:  Try the COPY (SELECT ...) TO variant.
```

Ceci fonctionne mais complique inutilement :

```
\copy (select * from mv1) to 'mv1.csv' with (format csv);
```

En version 18, il n'y a plus besoin de contournement :

```
demo =# create materialized view mv1 as select * from t1;
SELECT 5

demo =# copy mv1 to '/tmp/mv1.csv' with (format csv);
COPY 5
```

2.8 PSQL - REQUÊTES PRÉPARÉES NOMMÉES

- Nouvelles métacommandes :
  - `\parse`, `\bind_named` et `\close`.
- Principalement destinées aux tests de non régression

Les métacommandes de psql ne supportaient jusqu'à maintenant pas les requêtes préparées nommées. `\bind` permettait uniquement la création de requêtes préparées non nommées. Voici un exemple de son utilisation :

```
postgres=# SELECT relname, relkind FROM pg_class WHERE relname = $1 \bind 'pg_class'
 ↪  \g
 relname  | relkind
----------+---------
 pg_class | r
(1 row)
```

Trois nouvelles options sont désormais disponibles pour créer des requêtes préparées nommées dans `psql` :

- `\parse` : crée une requête préparée ;
- `\bind_named` : bind et exécute une requête préparée ;
- `\close` : supprime une requête préparée.

Voici un exemple de leur utilisation :

```
postgres=# SELECT relname, relkind FROM pg_class WHERE relname = $1 \parse pgclass

postgres=# \bind_named pgclass 'pg_class' \g
 relname  | relkind
----------+---------
 pg_class | r
(1 row)

postgres=# \bind_named pgclass 'pg_shadow' \g
  relname  | relkind
-----------+---------
 pg_shadow | v
(1 row)

postgres=# \close pgclass
```

Ces métacommandes permettent de tester le protocole de communication étendu de PostgreSQL directement via `psql`. Elles vont permettre d'améliorer les tests de non régression.

2.9 PSQL - MODE PIPELINE

- Nouvelles métas commandes :
  - `\startpipeline`, `\endpipeline`, `\syncpipeline`, `\flush`, `\flushrequest`, `\getresults` et `\sendpipeline`
- Nouvelles variables :
  - `PIPELINE_SYNC_COUNT`, `PIPELINE_COMMAND_COUNT`, `PIPELINE_RESULT_COUNT`
- Statut du pipeline dans le prompt avec %P (`on`, `off`, `abort`)
- Principalement destinés aux tests de non régression
- Permet de convertir un script en mode pipeline

Le mode pipeline permet d'envoyer un ensemble de commandes de manière groupée à PostgreSQL et de récupérer le résultat ensuite. Cela permet de diminuer le nombre d'allers-retours entre client et serveur en regroupant les messages envoyés, ce qui est particulièrement intéressant quand il y a une forte latence entre le serveur de base de données et le serveur d'application.

Les métacommandes suivantes permettent d'utiliser le mode pipeline du protocole de communication directement dans `psql` :

- `\startpipeline` : démarre un pipeline. Toutes les requêtes préparées ou les requêtes terminées par un `;` (elles sont en fait transformées en requêtes préparées) sont ajoutées dans une queue jusqu'à ce que le pipeline soit terminé ou synchonisé ;
- `\endpipeline` : termine un pipeline. Cette commande envoie toutes les commandes au serveur, les réponses sont ensuite traitées par `psql` ;
- `\syncpipeline` : place une demande de synchronisation dans la queue sans envoyer les commandes de la queue au serveur ;
- `\flush` et `\flushrequest` : force le backend à envoyer les données en attente dans son buffer d'envoi. Chaque commande utilise une portion du code de PostgreSQL différente ;
- `\getresults` : lit les résultat du pipeline. Il est possible de contrôler le nombre de résultats à lire, `0` signifiant que l'on souhaite tout récupérer ;
- `\sendpipeline` : cette commande remplace `\g` et `\gx` pour les pipelines.

Ces métacommandes permettent de tester le protocole de communication étendu de PostgreSQL directement via `psql`. Elles vont permettre d'améliorer les tests de non régression.

Dans un patch différent, trois compteurs ont été ajoutés pour suivre le statut d'un pipeline :

- `PIPELINE_SYNC_COUNT` : nombre de commandes sync dans la queue ;
- `PIPELINE_COMMAND_COUNT` : nombre de commandes dans la queue ;
- `PIPELINE_RESULT_COUNT` : nombre de rapports lisibles avec la commande `\getresult`.

Voici un exemple d'utilisation de ces fonctionnalités :

```
psql << '_END_SCRIPT_'
\set pipeline_info '\\echo sync: :PIPELINE_SYNC_COUNT, cmd: :PIPELINE_COMMAND_COUNT,
 ↪  result: :PIPELINE_RESULT_COUNT'

\startpipeline

-- La commande fonctionne avec des requêtes non préparées. Elles seront
-- transformées en requêtes préparées de manière transparente.
DROP TABLE IF EXISTS test_psql;
CREATE TABLE test_psql(i int);
:pipeline_info
\syncpipeline

-- ... des requêtes préparée non nommées
INSERT INTO test_psql(i)
SELECT generate_series($1::bigint, $2::bigint) \bind 1 10 \sendpipeline

-- ... et nommées
INSERT INTO test_psql(i)
SELECT generate_series($1::bigint, $2::bigint) \parse myinsert
\bind_named 'myinsert' 11 20 \sendpipeline
\bind_named 'myinsert' 21 30 \sendpipeline
\close 'myinsert'
:pipeline_info
\syncpipeline

SELECT * FROM test_psql;

:pipeline_info
\flushrequest
:pipeline_info
\getresults
:pipeline_info
\endpipeline
_END_SCRIPT_
```

Résultat :

```
sync: 0, cmd: 2, result: 0
sync: 1, cmd: 5, result: 2
```

```
sync: 2, cmd: 1, result: 7
sync: 2, cmd: 0, result: 8

DROP TABLE
CREATE TABLE
INSERT 0 10

INSERT 0 10
INSERT 0 10

 i
----
  1
...
 30
(30 rows)

sync: 0, cmd: 0, result: 0
```

Pour finir, la variable de prompt %P a été ajoutée pour donner des informations sur l'état du pipeline parmi on, `off`, `abort`.

Avec cette définition de prompt :

```
\set PROMPT1 'pipeline: %P (%:PIPELINE_SYNC_COUNT:, %:PIPELINE_COMMAND_COUNT:,
 ↪  %:PIPELINE_RESULT_COUNT:)%#%x '
```

nous obtenons le prompt suivant, avec un statut à `off` :

```
pipeline: off (0, 0, 0)# \startpipeline
```

Le démarrage du mode pipeline passe le statut à on :

```
pipeline: on (0, 0, 0)# SELECT failure;
pipeline: on (0, 1, 0)#* \flushrequest
```

… et une erreur le passe à `abort` :

```
pipeline: on (0, 0, 1)#* \getresults
ERROR:  column "failure" does not exist
LINE 1: SELECT failure;
pipeline: abort (0, 0, 0)#
```

2.10 PSQL - AUTRES AMÉLIORATIONS

- Nom de service :
  - %s : nouvelle variable de prompt
  - SERVICE : nouvelle variable d'environnement de psql
- La plupart des métacommandes de liste ont désormais une option \x
- \conninfo affiche désormais un tableau
- Informations sur la caractéristique leakproof des fonctions, opérateurs et casts
- \dP+ affiche désormais la méthode d'accès associée à une table partitionnée
- \dx affiche désormais la version par défaut des extensions
- \watch : temps d'attente configurable avec WATCH_INTERVAL

Prompt et nom de service :

psql met désormais à disposition des variables qui permettent de connaître le service en cours d'utilisation.

Avec ce fichier de service :

```
[postgres@rocky9 ~]$ cat ~/pgservice.conf
[pg18]
dbname=postgres
user=benoit
port=5432
```

l'information du service est disponible dans le prompt via %s et dans les scripts via la variable d'environnement SERVICE :

```
[postgres@rocky9 ~]$ PGSERVICEFILE=~/pgservice.conf psql "service=pg18"
psql (18beta1)
Type "help" for help.

postgres=> \set PROMPT1 '%n@%/ (%s) %R%#%x '
benoit@postgres (pg18) => \echo service: :SERVICE
service: pg18
```

Affichage étendu (\x) pour toutes les métacommandes :

L'option \x [on|off] permet de basculer psql en affichage étendu, c'est-à-dire avec les champs en ligne et non en colonne.

Elle est désormais directement intégrée aux métas commandes. Voici un exemple avec `\dt` :

```
postgres=# \dt
            List of tables
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | t1        | table | postgres
 public | t2        | table | postgres
(3 rows)

postgres=# \dtx
List of tables
-[ RECORD 1 ]-----
Schema | public
Name   | t1
Type   | table
Owner  | postgres
-[ RECORD 2 ]-----
Schema | public
Name   | t2
Type   | table
Owner  | postgres
```

Cela fonctionne aussi si l'on veut les détails additionnels en ajoutant `+`. D'ailleurs `\dt+x` et `\dtx+` renvoient la même chose.

La métacommande `\conninfo` affichait autrefois son résultat sous forme d'une phrase avec seulement l'utilisateur, l'hôte et le port. Elle affiche désormais des informations plus complètes sous forme de tableau :

```
postgres=# \conninfo
         Connection Information
      Parameter       |      Value
----------------------+-----------------
 Database             | postgres
 Client User          | postgres
 Socket Directory     | /run/postgresql
 Server Port          | 5432
 Options              |
 Protocol Version     | 3
 Password Used        | false
 GSSAPI Authenticated | false
 Backend PID          | 30801
 TLS Connection       | false
 Superuser            | on
 Hot Standby          | off
(12 rows)
```

Affichage de la caractéristique `leakproof` sur les fonctions :

Une fonction dite `leakproof` ne risque pas de provoquer de fuites d'informations, ce qui permet au planificateur de placer son évaluation au moment le plus opportun pour diminuer les coûts. Nos manuels détaillent ce thème plus en détail ici[5].

Il est désormais possible de visualiser si une fonction est `leakproof` dans le mode avancé (+) des méta commandes suivantes :

- `\df+` : liste des fonctions ;
- `\do+` : liste des opérateurs ;
- `\dAo+` : liste des familles d'opérateurs ;
- `\dC+` : liste des casts.

Voici un exemple avec la fonction `generate_series(bigint, bigint, bigint)` :

```
postgres=# \dfx+ generate_series bigint bigint bigint
List of functions
-[ RECORD 1 ]-------+--------------------------------
Schema              | pg_catalog
Name                | generate_series
Result data type    | SETOF bigint
Argument data types | bigint, bigint, bigint
Type                | func
Volatility          | immutable
Parallel            | safe
Owner               | postgres
Security            | invoker
Leakproof?          | no
Access privileges   |
Language            | internal
Internal name       | generate_series_step_int8
Description         | non-persistent series generator
```

Tables partitionnées et méthodes d'accès :

La version 18 permet de spécifier la méthode d'accès d'une table partitionnée.

Voici un exemple avec la méthode d'accès `heap2` :

```
postgres=# \dPx+ tpartaccess
List of partitioned relations
-[ RECORD 1 ]-+------------------
Schema        | public
Name          | tpartaccess
Owner         | postgres
```

[5] https://dali.bo/dba1_html#restreindre-les-droits

```
Type          | partitioned table
Parent name   |
Table         |
Access method | heap2
Total size    | 0 bytes
Description   |
```

Affichage de la version par défaut des extensions dans `\dx` :

La méta commande `\dx` affiche désormais la version disponible sur l'OS, en plus de celle installée dans PostgreSQL :

```
postgres=# \dxx
List of installed extensions
-[ RECORD 1 ]---+----------------------------
Name            | plpgsql
Version         | 1.0
Default version | 1.0
Schema          | pg_catalog
Description     | PL/pgSQL procedural language
```

L'objectif est de rendre plus visible un décalage de version nécessitant l'exécution de `ALTER EXTENSION UPDATE`.

Intervalle de rafraîchissement de `\watch` :

L'intervalle de rafraîchissement de la métacommande `\watch` est modifiable et consultable via la variable `WATCH_INTERVAL` ;

```
postgres=# \set WATCH_INTERVAL 0.1
postgres=# SELECT 'blip' \watch
Mon 21 Jul 2025 01:08:39 PM UTC (every 0.1s)

 ?column?
----------
 blip
(1 row)

postgres=# \set WATCH_INTERVAL 1
postgres=# SELECT 'blip' \watch
Mon 21 Jul 2025 01:10:17 PM UTC (every 1s)

 ?column?
----------
 blip
(1 row)
```

3/ Administration

3.1 NOUVELLE MÉTHODE D'AUTHENTIFICATION OAUTH

- Méthode oauth
- Nécessite
  - un fournisseur d'identité ou IdP (OAuth2 ou OIDC)
  - un module de validation côté serveur
  - un client PostgreSQL connaissant cette méthode
- Module de validation
  - paramètre `oauth_validator_libraries`
  - un module OIDC en cours d'écriture par Percona

Cette méthode d'auhentification fait appel à un fournisseur d'identité via un module de validation. Ce module de validation est une librairie dont le nom doit être indiqué dans le paramètre `oauth_validator_libraries`. Nous n'avons connaissance que d'un module de validation en cours d'écriture, celui de Percona, appelé `pg_oidc_validator`.

Il est ensuite nécessaire de configurer le fichier `pg_hba.conf` avec la méthode `oauth` et ses arguments (`scope`, `issuer`, `validator` et `map`). L'option `map` nécessite aussi la configuration du fichier `pg_ident.conf`.

Pour l'instant, cette fonctionnalité reste compliquée à tester, par manque justement d'un module fiable.

3.2 ACTIVATION PAR DÉFAUT DES SOMMES DE CONTRÔLE

- Activation des sommes de contrôle par défaut dans `initdb`
- Nouvelle option `--no-sync-data-files`

Activation des sommes de contrôle par défaut dans initdb :

Les sommes de contrôle sont désormais activées par défaut lors de la création d'une instance avec `initdb`. Elles peuvent être désactivées lors de la création avec l'option `--no-data-checksums`, ou ensuite avec la commande `pg_checksums --disable` sur une instance arrêtée.

Les arguments *contre* ce changement était principalement centrés sur l'impact sur les performances que peut induire cette fonctionnalité (la valeur de -5% de transactions par seconde a été annoncée). Cet impact n'est pas directement liée au calcul des sommes de contrôle, mais à l'augmentation de la volumétrie de WAL due à l'activation des `wal_log_hints` et des conséquences qui en découlent.

Le principal argument *pour* est la sécurité qu'apportent les sommes de contrôle, en permettant d'éviter une corruption silencieuse. De plus, l'impact n'est pas présent dans tous les cas. Enfin, beaucoup de gens utilisent cette fonctionnalité, il serait préférable de l'activer par défaut pour que les tests des nouvelles fonctionnalités prennent en compte les sommes de contrôle.

Nouvelle option `--no-sync-data-files`

Par défaut, `initdb` écrit tous les fichiers des bases de données de manière durable. L'option `--no-sync-data-files` évite de demander la synchronisation individuelle des fichiers dans le répertoire `base` et les répertoires des tablespaces utilisateurs. Cette option est destinée à être utilisée avec des outils qui peuvent garantir que les fichiers sont synchronisés d'une autre manière, ou plus tard, par exemple lors d'une migration avec `pg_upgrade`.

3.3 NOUVEAU RÔLE PG_SIGNAL_AUTOVACUUM_WORKER

- But du rôle `pg_signal_backend`
  - arrêter une requête ou une connexion
  - … mais juste pour les processus du même rôle
  - … donc pas pour `autovacuum`
- Nouveau rôle `pg_signal_autovacuum_worker`
  - permet à un utilisateur d'arrêter un *autovacuum*

Le rôle `pg_signal_backend` donne le droit à tout utilisateur d'annuler une requête ou d'arrêter une connexion. Cependant, il permettait à tout utilisateur de le faire, y compris pour les processus `autovacuum` ainsi que le `logical replication launcher`. En novembre 2023, un correctif a interdit cette pratique, qui était pourtant utilisée volontairement par certaines personnes.

Cette version intègre donc un nouveau rôle : `pg_signal_autovacuum_worker` qui permet à un administrateur de donner le droit à un utilisateur simple d'annuler une requête ou d'arrêter une connexion pour un `autovacuum worker`, même si cet utilisateur n'a pas l'attribut SUPERUSER.

Comme pour tous les rôles `pg_*`, il convient d'être très prudent lors de l'attribution de ces rôles.

3.4 CHANGEMENT SUR LES PARAMÈTRES DE TRACES

- Transformation de `log_connections` en liste
  - `receipt`, `authentication`, `authorization`, `setup_durations`
  - `all`
  - on et `off` toujours acceptés
- Modification de la trace des connexions
  - contient les durées d'établissement de la connexion si `setup_durations`
- Nouveau paramètre `log_lock_failures`
  - trace les échecs de demande de verrou
- Nouveau joker `%L` pour `log_line_prefix` pour les IP

log_connections :

Le paramètre `log_connections` a profondément changé en version 18. Il traçait tout ce qui concernait les connexions : la réception d'une demande de connexion, l'authentification et l'autorisation. Il est maintenant possible de sélectionner les étapes à tracer : `receipt` pour la réception, `authentication` pour l'authentification, `authorization` pour l'autorisation. Il est possible d'en spécifier plusieurs en les séparant d'une virgule, par exemple :

```
log_connections = 'receipt,authorization'
```

La granularité de cette configuration est donc plus importante et permet d'éviter des traces inutiles, même si, à l'heure actuelle, il est préconisé de les tracer toutes.

Une nouvelle information est traçable : la durée des étapes d'une connexion. Cela se fait avec l'option `setup_durations` et permet d'obtenir ce type de trace :

```
LOG:  connection ready: setup total=3.541 ms, fork=0.917 ms,
      authentication=0.576 ms
```

Dans cet exemple, la demande de connexion a duré au total 3,541 ms, sur lequel la création du processus a pris 0,917 ms et l'authentification 0,576 ms. Ces informations sont intéressantes pour prendre

en compte la durée de l'authentification, notamment si vous utilisez un serveur d'authentification externe (LDAP ou Active Directory par exemple). Elles sont aussi intéressantes pour savoir si le système d'exploitation fonctionne rapidement (quand la partie `fork` est rapide).

L'option `all` contient évidemment toutes les traces déjà citées (voir exemple plus bas).

Les valeurs on et `off` sont conservées pour la compatibilité. on équivaut à `receipt`, `authentication`, `authorization` (donc sans `setup_durations`) pour le même comportement qu'en PostgreSQL 17.

Une liste vide ou la valeur `off` désactive toute trace de connexion.

`log_line_prefix` et `%L` :

Enfin, le paramètre `log_line_prefix` dispose d'un nouveau caractère joker, `%L`, qui permet de tracer l'adresse IP **du serveur** par lequel le client s'est connecté. En effet, une instance peut répondre sur plusieurs IP différentes.

Les connexions via un socket Unix affichent `[local]`. Les *background workers* affichent `[none]`. Il est à noter que ce champ n'a pas été ajouté aux sorties `csvlog` et `jsonlog`.

Exemple :

Exemple de résultat pour une connexion avec `psql -h 127.0.0.1` (et non `localhost`, équivalent généralement à `::1`) avec `log_connections` à `all`, et `log_line_prefix` à `%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h,via %L`.

```
2026-03-09 16:37:57 CET [597443]: [1-1]
 ↪  user=[unknown],db=[unknown],app=[unknown],client=127.0.0.1,via 127.0.0.1 LOG:
 ↪  connection received: host=127.0.0.1 port=37454
2026-03-09 16:37:57 CET [597443]: [2-1]
 ↪  user=postgres,db=postgres,app=[unknown],client=127.0.0.1,via 127.0.0.1 LOG:
 ↪  connection authenticated: identity="postgres" method=scram-sha-256
 ↪  (/etc/postgresql/18/defo/pg_hba.conf:106)
2026-03-09 16:37:57 CET [597443]: [3-1]
 ↪  user=postgres,db=postgres,app=[unknown],client=127.0.0.1,via 127.0.0.1 LOG:
 ↪  connection authorized: user=postgres database=postgres application_name=psql SSL
 ↪  enabled (protocol=TLSv1.3, cipher=TLS_AES_256_GCM_SHA384, bits=256)
2026-03-09 16:37:57 CET [597443]: [4-1]
 ↪  user=postgres,db=postgres,app=psql,client=127.0.0.1,via 127.0.0.1 LOG:
 ↪  connection ready: setup total=8.861 ms, fork=0.510 ms, authentication=2.210 ms
```

On voit tout de suite la différence de durée avec une connexion *peer* (par la *socket* Unix) qui n'a besoin ni du SSL ni de l'authentification SCRAM, tous deux assez chronophages :

```
2026-03-09 16:48:07 CET [600591]: [2-1]
 ↪  user=postgres,db=postgres,app=[unknown],client=[local],via [local] LOG:
 ↪  connection authenticated: identity="christ" method=peer
 ↪  (/etc/postgresql/18/defo/pg_hba.conf:87)
```

```
2026-03-09 16:48:07 CET [600591]: [3-1]
 ↪  user=postgres,db=postgres,app=[unknown],client=[local],via [local] LOG:
 ↪  connection authorized: user=postgres database=postgres application_name=psql
2026-03-09 16:48:07 CET [600591]: [4-1]
 ↪  user=postgres,db=postgres,app=psql,client=[local],via [local] LOG:  connection
 ↪  ready: setup total=1.860 ms, fork=0.451 ms, authentication=0.177 ms
```

log_lock_failures :

Un nouveau paramètre apparaît pour les traces : `log_lock_failures`. Il permet de tracer uniquement les demandes de verrou non satisfaites via un `SELECT … FOR UPDATE NOWAIT`. Il y aura peut-être d'autres verrous dans le futur[1].

Voici un exemple de la trace obtenue :

```
LOG:  process 56906 could not obtain ShareLock on transaction 53579
DETAIL:  Process holding the lock: 56228, Wait queue: .
STATEMENT:  SELECT * FROM t1 WHERE c1=1 FOR UPDATE NOWAIT;
```

`log_lock_failures` ne permet pas de tracer les verrous provoquant un dépassement de `lock_timeout` (1 s par défaut), si ce paramètre a été activé,

Rappelons que les verrous en attente depuis plus d'une seconde (par défaut) sont déjà tracés avec `log_lock_waits` à on.

---

[1] https://www.postgresql.org/message-id/ecd83f43-ed95-4ccf-8503-457b2423c3e4%40oss.nttdata.com

3.5 AMÉLIORATIONS SUR L'AUTOVACUUM

- Possibilité de modifier `autovacuum_max_workers` sans redémarrer PostgreSQL
  - seuil maximum : nouveau paramètre `autovacuum_worker_slots` (16)
- Vacuum plus fréquent des très grandes tables
  - nouveau paramètre `autovacuum_vacuum_max_threshold`

autovacuum_max_workers :

Avant la version 18, la modification du paramètre `autovacuum_max_workers` nécessitait le redémarrage de l'instance pour qu'elle soit prise en compte. C'était problématique pour les serveurs en 24/7 pour lesquels une augmentation de ce paramètre aurait permis de traiter plus de bases et de tables en même temps.

En version 18, le changement de ce paramètre ne nécessite plus qu'un rechargement de la configuration. Cependant, pour que PostgreSQL puisse allouer en mémoire partagée statique les structures nécessaires pour chaque worker potentiel, il a fallu définir une limite et c'est là que rentre en jeu le nouveau paramètre, `autovacuum_worker_slots`. Au démarrage, `autovacuum_worker_slots` est utilisé pour calculer la taille des structures nécessaires au fonctionnement de l'`autovacuum`. La valeur 16, par défaut, devrait suffire à la plupart des installations. Le paramètre `autovacuum_max_worker` a une valeur comprise entre 1 et `autovacuum_worker_slots` et est utilisé comme limite du nombre de workers pour l'`autovacuum`.

autovacuum_vacuum_max_threshold :

Une autre amélioration a été apportée en version 18. Pour éviter que le nombre cible de lignes mortes soit trop important, il est possible de configurer une limite haute. Un nouveau paramètre a été créé : `autovacuum_vacuum_max_threshold`. Si sa valeur est inférieure au résultat du calcul standard `autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * nb_lignes_vivantes`, c'est lui qui est pris en compte pour la comparaison avec le nombre de lignes mortes.

Ce nouveau paramètre permet de résoudre partiellement le problème des tables avec de très nombreuses lignes que l'autovacuum visite trop rarement, et qui accumulent une fragmentation importante. En effet, avec la valeur par défaut de `autovacuum_vacuum_threshold`, il faut attendre que 20 % des lignes soient modifiées pour que l'autovacuum se déclenche. Pour une table

d'un milliard de lignes, il fallait donc attendre la modification de 200 millions d'entre elles. Il était alors (et il reste) conseillé de modifier `autovacuum_vacuum_threshold` et/ou `autovacuum_vacuum_scale_factor` sur les grandes tables, et/ou de mener des VACUUM la nuit ou le week-end.

Avec sa valeur par défaut de 100 millions de lignes, `autovacuum_vacuum_max_threshold` modifie le comportement à partir de 500 millions de lignes (pour une table monolithique, hors autre paramétrage) et réduit le besoin d'administration. Le seuil peut être modifié globalement s'il est jugé trop haut, ou par table ainsi :

```
ALTER TABLE t1 SET (AUTOVACUUM_VACUUM_MAX_THRESHOLD = 10_000_000);
```

La valeur `-1` revient à l'ancien comportement.

La discussion[2] pointe les avantages et inconvénients de cet autovacuum plus agressif : mise à jour plus fréquente des statistiques sur les données récentes, *visibility map* plus à jour, VACUUM insensible, temps perdu à parcourir de gros index de manière répétée…

Attention : ce paramètre ne change le comportement du démon `autovacuum` que pour ce qui concerne la partie VACUUM. Le démon ne lancera pas plus souvent qu'avant des ANALYZE (les développeurs ne sont pas sûrs qu'un ANALYZE plus fréquent est toujours une bonne chose).

---

[2] https://www.postgresql.org/message-id/956435f8-3b2f-47a6-8756-8c54ded61802%40dalibo.com

3.6 AMÉLIORATIONS SUR LES OUTILS - 1

- `vacuumdb`
  - nouvelle option `--missing-stats-only`
- `pg_combinebackup`
  - Nouvelle option `-k`/`--link`
- `pg_verifybackup`
  - vérifier les sauvegardes au format `tar` (compressées ou non)
  - requiert l'option `-n`/`--no-parse-wal`

Outil vacuumdb :

La nouvelle option `--missing-stats-only` de `vacuumdb` permet de calculer uniquement les statistiques pour les tables pour lesquelles il manquent des données statistiques sur l'un de ces trois types d'objet : colonne, expressions d'index ou statistiques étendues. C'est également utile après un `pg_upgrade` (puisqu'il permet désormais de conserver la plupart des statistiques).

Cette option peut être utilisée avec `--analyze-only` ou `--analyze-in-stages`.

Outil pg_combinebackup :

La commande `pg_combinebackup` permet de combiner les backups incrémentaux de `pg_basebackup` depuis la version 17. s'est vu ajouter une nouvelle option `-k` (`--link`) qui permet de combiner des sauvegardes en effectuant un lien physique avec les fichiers des sauvegardes. L'avantage de cette méthode est qu'elle permet de s'affranchir de la copie. Le désavantage est que les sauvegardes seront altérées lors du démarrage de l'instance cible. Cette méthode est donc utile lorsque les répertoires sources sont une copie qui sera supprimée après l'exécution de `pg_combinebackup`.

Pour réaliser une sauvegarde incrémentale sur un serveur avec la configuration par défaut, les adaptations suivantes doivent être faites.

```
summarize_wal = on
archive_mode = on
archive_command = 'rsync -a %p /var/lib/pgsql/18/backups/wal/%f'
```

Une sauvegarde initiale est ensuite réalisée, avant de modifier les données et réaliser une sauvegarde incrémentale.

```
cd /var/lib/pgsql/18/backups/
pg_basebackup --pgdata FULL
psql -c "CREATE TABLE t(i int); INSERT INTO t SELECT generate_series(1, 1000);"
pg_basebackup --pgdata INC --incremental ./FULL/backup_manifest
```

Les deux sauvegardes sont ensuite combinées pour créer le nouveau répertoire de données.

```
pg_combinebackup -o NEW/ --link FULL INC
```

Un message d'avertissement est émis pour nous prévenir du risque d'altération de la sauvegarde.

```
pg_combinebackup: warning: --link mode was used; any modifications to the output
↪  directory might destructively modify input directories
```

La commande suivante nous montre qu'un certain nombre de fichiers sont en commun entre les répertoires de sauvegardes (`FULL` et `INC`) et le répertoire de données (`NEW`). La seconde colonne affichée ci-dessous permet d'identifier les fichiers qui ont le même inode (et donc sont des liens physiques).

```
find . -type f -printf "%n %p\n" | sed -E "s/(.\/[A-Z]*)\/.*/\1/" \
| sort | uniq -c | grep -v wal
```

```
    312 1 ./FULL
    670 1 ./INC
      2 1 ./NEW
    668 2 ./FULL
    310 2 ./INC
    978 2 ./NEW
```

Outil pg_verifybackup :

`pg_verifybackup` permet désormais de vérifier des sauvegardes au format `tar`.

Commençons par réaliser une sauvegarde au format tar (compressée ou non).

```
pg_basebackup --pgdata FULL --format=t --gzip
```

La sauvegarde comprend un fichier manifeste, une archive des fichiers de la base de données, et une autre pour les archives de journaux de transactions. Si la base de données contenait des *tablespaces* utilisateurs, chaque *tablespace* aurait son archive.

```
ls -al FULL/
```

```
total 3220
drwx------. 2 postgres postgres      69 Jul 22 14:34 .
drwxr-xr-x. 3 postgres postgres      18 Jul 22 14:34 ..
-rw-------. 1 postgres postgres  139391 Jul 22 14:34 backup_manifest
-rw-------. 1 postgres postgres 3129402 Jul 22 14:34 base.tar.gz
-rw-------. 1 postgres postgres   17097 Jul 22 14:34 pg_wal.tar.gz
```

La commande de vérification des sauvegardes utilise `pg_waldump` pour vérifier les WAL. Malheureusement, elle ne sait pas encore vérifier des WAL qui sont stockés dans un fichier `tar`. `pg_verifybackup` nous informe de cette limitation lorsqu'on tente de l'exécuter tel quel :

```
/usr/pgsql-18/bin/pg_verifybackup FULL
```

```
pg_verifybackup: error: pg_waldump cannot read tar files
pg_verifybackup: hint: You must use -n/--no-parse-wal when verifying a tar-format
↪  backup.
```

Avec l'option `-n/--no-parse-wal`, la vérification peut être réalisée :

```
/usr/pgsql-18/bin/pg_verifybackup --no-parse-wal FULL
```

```
backup successfully verified
```

3.7 AMÉLIORATIONS SUR LES OUTILS - 2

- pg_bench
  - affiche désormais les erreurs de sérialisation et deadlock dans ses rapports.
- pg_createsubscriber
  - --all, --clean=publication, --enable-two-phase, --enable-failover
- pg_rewind
  - amélioration de --write-recovery-conf pour les *failover slots*

Outil pgbench :

Lorsque c'est approprié, les statistiques globales et par script contiennent maintenant le nombre de transactions échouées à cause d'erreurs de sérialisation ou de deadlock.

Voici un exemple tiré de la discussion du patch :

```
psql -dbname bench --command "CREATE TABLE t1(i int, j int);"
```

```
CREATE TABLE
```

```
cat script1.sql
```

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO t1 SELECT max(i)+1,2 FROM t1;
END;
```

```
cat script2.sql
```

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO t1 SELECT MAX(i)+1,2 FROM t1;
END;
```

```
/usr/pgsql-18/bin/pgbench \
  --time=30 \
  --client=2 \
  --file=script1.sql \
  --file=script2.sql \
  --failures-detailed \
```

```
  --report-per-command \
  bench


pgbench (18beta1)
starting vacuum...end.
transaction type: multiple scripts
scaling factor: 1
query mode: simple
number of clients: 2
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 11656
number of failed transactions: 11744 (50.188%)
number of serialization failures: 11744 (50.188%)
number of deadlock failures: 0 (0.000%)
latency average = 2.564 ms (including failures)
initial connection time = 5.911 ms
tps = 388.583176 (without initial connection time)
SQL script 1: script1.sql
 - weight: 1 (targets 50.0% of total)
 - 11639 transactions (49.7% of total)
 - number of transactions actually processed: 5773 (tps = 192.458019)
 - number of failed transactions: 5866 (50.400%)
 - number of serialization failures: 5866 (50.400%)
 - number of deadlock failures: 0 (0.000%)
 - latency average = 2.567 ms
 - latency stddev = 0.549 ms
 - statement latencies in milliseconds and failures:
         0.030           0 BEGIN;
         0.028           0 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
         1.141        5786 INSERT INTO t1 SELECT max(i)+1,2 FROM t1;
         1.383          80 END;
SQL script 2: script2.sql
 - weight: 1 (targets 50.0% of total)
 - 11761 transactions (50.3% of total)
 - number of transactions actually processed: 5883 (tps = 196.125156)
 - number of failed transactions: 5878 (49.979%)
 - number of serialization failures: 5878 (49.979%)
 - number of deadlock failures: 0 (0.000%)
 - latency average = 2.578 ms
 - latency stddev = 0.556 ms
 - statement latencies in milliseconds and failures:
         0.029           0 BEGIN;
         0.028           0 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
         1.147        5800 INSERT INTO t1 SELECT MAX(i)+1,2 FROM t1;
         1.388          78 END;
```

Les lignes suivantes ont été ajoutées :

```
- number of failed transactions
- number of serialization failures
```

Outil pg_createsubscriber :

Depuis PostgreSQL 17, la commande `pg_createsubscriber` permet de transformer une instance standby en réplica logique en créant les publications/souscriptions nécessaires. Elle s'est vu ajouter plusieurs options qui permettent de simplifier ce processus.

La nouvelle option `--all` de `pg_createsubscriber` permet de s'assurer que l'outil va créer un couple publication/souscription pour chaque base de données présente à la fois sur la source et la cible. Précédemment, la liste de toutes les bases de données devait être spécifiée avec l'option `-d, --database` pour arriver au même résultat, ce qui pouvait être fastidieux.

Les options `-d, --database`, `--publication`, `--subscription` et `--replication-slot` ne sont pas compatible avec `--all`.

L'option `--clean` permet de supprimer tous les objets correspondant au type spécifié sur le serveur cible (souscripteur). Pour le moment, le seul type disponible est `publication`.

L'option `--enable-two-phase` permet d'activer le mode *two-phase commit* sur toutes les souscriptions lors de leur création. La modification pouvait déjà être réalisée manuellement, mais nécessitait de désactiver la souscription au préalable.

Si cette option est activée, les modifications faites par chaque transaction préparée peuvent être envoyées au souscripteur à partir du moment où la commande `PREPARE TRANSACTION` est exécutée, plutôt que lors du `COMMIT PREPARED`. Elle sera traitée comme une transaction *two-phase*. Cela permet donc d'utiliser le *streaming* pour ce genre de transaction.

L'option `--enable-failover` permet de créer les slots de réplication en activant l'option *failover*. Cette option peut être utilisée avec l'option `--create-slot`.

Outil pg_rewind :

L'utilisation des slots de réplication logique synchronisés (ou *failover slots*) nécessite :

- d'activer `sync_replication_slots` sur la standby ;
- d'utliser un slot de réplication physique pour la réplication entre l'instance primaire et la standy en question (et donc le renseigner dans `primary_slot_name`) ;
- d'activer `hot_standby_feedback` sur la standby ;
- de fournir un `dbname` dans la chaîne de connexion de la réplication (`primary_conninfo`) ;
- de renseigner le nom du slot dans `synchronized_standby_slots` sur la primaire (conseillé).

Lorsque `--write-recovery-conf` est spécifié, `pg_rewind` est désormais capable d'ajouter le nom de la base de données dans le paramètre `primary_conninfo` du fichier `postgresql.auto.conf` en se basant sur la chaîne de connection spécifiée via le paramètre `--source-server`.

Cela simplifie donc le retour en service de l'instance en évitant une modification manuelle de la configuration. C'est intéressant puisque, généralement, on doit utiliser `pg_rewind` dans le contexte d'un incident ou avec un outil d'automatisation comme `patroni`.

3.8 ÉVOLUTIONS DE PG_DUMP/PG_DUMPALL/PG_RESTORE

- `pg_dump`, `pg_dumpall` et `pg_restore` :
  - `--no-data`, `-no-schema`
  - `--no-policies`
- `pg_dump` et `pg_dumpall` :
  - `--sequence-data`

Sélectivité des outils de sauvegardes et restauration :

Les trois outils `pg_dump`, `pg_dumpall` et `pg_restore`, connaissaient déjà les options `--data-only` (export des données uniquement) `--schema-only` (ordres DDL, index…), et, pour `pg_dump`/`pg_restore`, `--section[pre-data|data|post-data]`.

S'ajoutent d'abord les options `--no-data` (export sans données) et `--no-schema` (export sans les ordres de définition de structure).

Les différences avec les options précédentes peuvent être subtiles. Par exemple, `--no-data` et `--schema-only` contiennent tous deux les définitions des vues matérialisées, mais `--section=[pre|post]-data` y ajoute des ordres `REFRESH MATERIALIZED VIEW`.

pg_dumpall, pg_restore et RLS :

`pg_dump` avait déjà une option `--no-policies` pour contrôler l'ajout de la configuration du `ROW LEVEL SECURITY` sur les objets de base de données dans le dump. Elle n'était pas disponible pour `pg_dumpall` et `pg_restore` ; le manque est enfin comblé.

Export des données de séquence :

`pg_dump` et `pg_dumpall` permettent désormais d'exporter spécifiquement les données associées aux séquences comme le montre l'exemple ci-dessous.

```
pg_dump --no-data --no-schema --sequence-data

--
-- PostgreSQL database dump
--

-- Dumped from database version 18beta1
-- Dumped by pg_dump version 18beta1
```

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

--
-- Name: test_i_seq; Type: SEQUENCE SET; Schema: public; Owner: postgres
--

SELECT pg_catalog.setval('public.test_i_seq', 3, false);


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

L'option est notamment utilisée par `pg_upgrade --swap` pour migrer les données des séquences. En effet, depuis la version 10, qui a introduit le catalogue `pg_sequences`, les fichiers de données associés aux séquences ne sont pas transférés sur le nouveau cluster ; la migration se fait en utilisant `pg_restore`, ce qui entre en contradiction avec le concept de `pg_upgrade --swap` discuté plus loin. Cette nouvelle option permet donc de ne récupérer que les séquences dans le catalogue source.

3.9 GESTION DES STATISTIQUES PAR PG_DUMP/PG_DUMPALL/PG_RESTORE

- Nouvelles options
  - `--statistics`
  - `--statistics-only`, `--no-statistics`
- Pas par défaut !
- Pas les statistiques étendues

Principe :

`pg_dump`, `pg_dumpall` et `pg_restore` permettent désormais d'importer ou exporter les statistiques des données. Ces options apparaissent :

- `--no-statistics` : exclure explicitement les statistiques du traitement (le défaut) ;
- `--statistics` : importer/exporter les statistiques en même temps que les données ;
- `--statistics-only` : importer/exporter uniquement les statistiques.

Les statistiques ne sont *pas* exportées par défaut car les développeurs ne sont pas parvenus à un consensus sur le sujet.

Deux fonctions pour le réimport des statistiques apparaissent dans les sorties de `pg_dump` : `pg_restore_relation_stats()` et `pg_restore_attribute_stats()`.

Exemple d'export de pg_dump :

Voici un exemple de commande d'export de statistiques pour une table.

```
pg_dump -t demo --statistics-only
```

- Cette première fonction apparaît pour chaque table et renseigne des statistiques présents dans `pg_class` (taille en ligne et blocs, partie gelée ou visible) :

```
…
SELECT * FROM pg_catalog.pg_restore_relation_stats(
        'version', '180001'::integer,
        'schemaname', 'public',
        'relname', 'demo',
        'relpages', '5406'::integer,
        'reltuples', '1e+06'::real,
        'relallvisible', '5406'::integer,
```

```
        'relallfrozen', '0'::integer
);
```

- Ces données contiennent un numéro de version : en effet, le client pg_dump en version 18 sait exporter les statistiques depuis une instance d'une version précédente. Certains champs peuvent alors être vides. Par contre, il n'y a pas moyen de restaurer ces statistiques sur une instance antérieure à la 18.
- Pour chaque index de la table apparaît un appel similaire à `pg_restore_relation_stats`.
- La commande suivante apparaît pour chaque champ de la table avec l'histogramme, les valeurs les plus courantes, la corrélation…, c'est-à-dire ce qui se retrouve dans `pg_statistics`.

```
SELECT * FROM pg_catalog.pg_restore_attribute_stats(
        'version', '180001'::integer,
        'schemaname', 'public',
        'relname', 'demo',
        'attname', 'j',
        'inherited', 'f'::boolean,
        'null_frac', '0'::real,
        'avg_width', '4'::integer,
        'n_distinct', '10'::real,
        'most_common_vals', '{2,1,9,5,8,0,7,3,6,4}'::text,
        'most_common_freqs',
        ↪  '{0.10236666,0.10146666,0.10123333,0.1008,0.09983333,0.099366665,0.09923334,0.0988,0.0
        'correlation', '0.09628114'::real
);
```

- Chaque index fonctionnel donne lieu à un appel de chaque fonction, comme une table.
- Les statistiques des vues matérialisées sont exportées de la même manière que pour les tables.
- Si l'on utilise `--statistics --section=…`, selon les objets, les statistiques sont incluses dans `--section==data` (tables) ou dans `--section=post-data` (index, vues matérialisées…)

Limites :

Les statistiques étendues ne sont *pas* incluses, pour des raisons techniques. (Peut-être dans une version future ?) Les définitions des objets statistiques sont bien sûr présentes dans la sauvegarde (ordres `CREATE STATISTICS`) mais nécessitent un `ANALYZE` explicite pour les remplir.

Des statistiques importées ne sont pas pérennes : elles seront écrasées lors d'un prochain `ANALYZE`.

Faut-il continuer à lancer ANALYZE après un pg_restore ?

Un import devrait déclencher un `ANALYZE`, l'export-réimport des statistiques est donc à priori inutile.

Il y a toutefois des cas de migration par `pg_dump`/`pg_restore` où l'on peut vouloir désactiver l'autovacuum, et réimporter des statistiques de la base source. Cependant, il manquera les éventuelles statistiques étendues, et les statistiques importées ne reproduiront pas forcément la fragmentation des données réimportées.

Peut-on utiliser les fonctions pour modifier les statistiques d'une table ?

Oui, et ce peut être pratique pour des tests, ou à titre pédagogique, ou pour faire des tests sur une base en fait vide.

Par contre, « figer » des statistiques en production avec cet outil est déconseillé. Les statistiques modifiées seront remplacées lors du prochain `ANALYZE`. Ou alors, il faut inhiber l'autoanalyze sur la table, ce qui est fortement déconseillé.

3.10 PG_UPGRADE

- Préservation des statistiques sur les données
  - `--no-statistics`
- Parallélisation des contrôles : `--jobs`
- Nouvelle stratégie de transfert des données : `--swap`
- Compatibilité entre architectures : `--set-char-signedness` (x86/arm)

`pg_upgrade` s'est vu ajouter quelques options afin de réduire le temps d'indisponibilité associé à une migration. Une option concernant les migrations entre plateformes avec des architectures différentes a également été ajoutée.

Import des statistiques :

`pg_upgrade` exploite la nouvelle option `--statistics-only` de `pg_dump` pour préserver les statistiques de l'optimiseur lors de la mise à jour. Seules les statistiques étendues ne sont pas transférées. Il est possible de ne pas conserver les statistiques avec l'option `--no-statistics`.

La procédure de mise à jour recommandée a aussi été enrichie, le calcul des statistiques se fait désormais en deux étapes afin de pouvoir réouvrir l'instance aux utilisateurs au plus vite :

```
vacuumdb [--jobs X] --all --analyze-in-stages --missing-stats-only
-- open the instances to users if needed.
vacuumdb [--jobs X] --all --analyze-only
```

Parallélisation des contrôles (`--jobs`) :

Beaucoup de traitements réalisés par `pg_upgrade` nécessitent d'exécuter des vérifications sur toutes les bases de données de l'instance. Jusqu'à maintenant, ces contrôles étaient fait séquentiellement, ce qui pouvait prendre beaucoup de temps avec des instances contenant de nombreuses bases de données. L'option `--jobs` permet de déterminer le niveau de parallélisation de ces tâches.

Nouvelle stratégie de migration `--swap` :

L'option `--swap` vient s'ajouter aux options `--link`, `--clone`, `--copy` et `--copy-file-range`. Elle est destinée aux migrations d'instances avec beaucoup de relations, et pour lesquelles source et cible sont sur le même système de fichier. En effet, avec ce nouveau mode, les données sont déplacées du répertoire de données de l'ancienne instance vers celui de la nouvelle. Les tablespaces sont aussi déplacés de la même manière. Le catalogue généré pour la nouvelle instance vient ensuite

remplacer celui de l'ancienne. Il est recommandé d'utiliser l'option `--sync-method=fsync` avec ce mode.

En cas d'erreur pendant ou après le transfert des fichiers, il est recommandé de repartir d'une sauvegarde.

Pour finir, cette méthode n'est disponible que pour les instances à migrer depuis une version supérieure ou égale à 10.

Encodage des caractères (`--set-char-signedness`) :

L'option `--set-char-signedness` a été ajoutée à `pg_upgrade`. Elle peut prendre la valeur `signed` ou `unsigned` et permet de définir si l'encodage des caractères est signé ou non. Par défaut, `pg_upgrade` fait cette configuration tout seul en se basant sur l'architecture du serveur : `signed` sur x86 et `unsigned` sur ARM. Cette option peut être utilisée dans certains cas de migration entre plateformes ayant des architectures différentes et pourrait également servir de base pour le support de la réplication entre plateformes hétérogènes.

L'information est ajoutée dans le *control file* de l'instance.

```
/usr/pgsql-18/bin/pg_controldata | grep "char data"
```

```
Default char data signedness:         signed
```

Pour réaliser cette modification du *control file* , l'option `--char-signedness` a été ajoutée à `pg_resetwal`. Elle est destinée à être utilisée par `pg_upgrade` uniquement.

3.11 NOUVEAU PARAMÈTRE EXTENSION_CONTROL_PATH

- Nouveau paramètre `extension_control_path`
- Emplacements supplémentaires pour les extensions

Le paramètre `extension_control_path` est désormais disponible. Il permet de spécifier des nouveaux emplacements où PostgreSQL cherchera les fichiers de contrôle des extensions (`extension.control`). Jusqu'à présent tous les fichiers devaient être placés dans un seul dossier. Ce nouveau paramètre apporte donc plus de flexibilité, notamment pour faciliter l'ajout d'extensions dans des environnements qui doivent demeurer immuables, typiquement des images de conteneurs. L'ajout d'extension nécessitait alors la reconstruction de l'image. Le couplage de cette nouveauté avec, par exemple, le mécanisme d'ImageVolume dans Kubernetes apporte de nouvelles possibilités.

Par défaut, `extension_control_path` est positionné à `$system` (valeur positionnée à la compilation de PostgreSQL).

```
postgres=# SHOW extension_control_path;
┌──────────────────────────┐
│ extension_control_path   │
├──────────────────────────┤
│ $system                  │
└──────────────────────────┘
(1 row)
```

L'emplacement correspondant à `$system` peut être trouvé avec `pg_config --sharedir`.

La valeur de ce paramètre doit être une liste d'emplacements séparés par `:`. Un rechargement de la configuration est nécessaire.

```
postgres=# ALTER SYSTEM SET extension_control_path =
 ↪  '$system:/usr/local/share/nos_extensions_maison';
ALTER SYSTEM

postgres=# SELECT pg_reload_conf();
┌────────────────┐
│ pg_reload_conf │
├────────────────┤
│ t              │
└────────────────┘
(1 row)

postgres=# SHOW extension_control_path;
```

```
┌──────────────────────────────────────────────────┐
│                extension_control_path            │
├──────────────────────────────────────────────────┤
│ $system:/usr/local/share/nos_extensions_maison   │
└──────────────────────────────────────────────────┘
(1 row)
```

PostgreSQL parcourra les emplacements dans l'ordre d'apparition. Si une même extension (même nom) est présente dans deux emplacements, seule la première version sera visible pour PostgreSQL. Elle n'apparaîtra donc pas dans `pg_available_extensions`.

4/ Réplication

4.1 INVALIDATION DES SLOTS DE RÉPLICATION BASÉE SUR LE TEMPS D'INACTIVITÉ

- Nouveau paramètre `idle_replication_slot_timeout`
- Invalidation basée sur la durée d'inactivité
- Exprimé en minute
- Réplication physique et logique

Depuis la version 13 de PostgreSQL, nous disposons d'un moyen pour invalider les slots de réplication en fonction de la quantité de WAL retenus : `max_slot_wal_keep_size`. Ce paramètre permet de préserver le service en éliminant une source possible de saturation du stockage contenant les WAL.

La version 18 introduit le paramètre `idle_replication_slot_timeout` qui permet d'invalider des slots en se basant sur la durée d'inactivité du slot. Celle-ci est calculée avec le champ `inactive_since` de `pg_replication_slots`, apparu en version 17. Le défaut (0) conserve indéfiniment les slots. La durée est exprimée en minute.

Comme l'invalidation a lieu lors des checkpoints, qui ont lieu tous les `checkpoint_timeout` (5 minutes par défaut, parfois plus), ou une fois `max_wal_size` atteinte, il est possible que le slot soit invalidé plus tard que prévu. Dans ce cas, il est possible de forcer un checkpoint manuellement.

Ce mécanisme d'invalidation ne fonctionne que pour des slots qui réservent des WAL, ce qui est le mode habituel. Cela exclut les slot synchronisés (`synced`) depuis le primaire (une nouveauté de la version 17), c'est-à-dire des slots posés sur un serveur secondaire (réplica physique), en complément d'un primaire, et destinés à garantir le maintien de la réplication en cas de bascule. En effet, ces slots sont considérés comme inactifs car ils ne réalisent pas de décodage logique.

4.2 RÉPLICATION LOGIQUE - PUBLICATION DES COLONNES GÉNÉRÉES STOCKÉES

- Les colonnes `GENERATED ALWAYS AS () STORED` sont publiées si :
  - les colonnes sont dans la liste des colonnes spécifiées
  - ou : `publish_generated_columns = stored` (defaut : none)
- Les colonnes générées présentes dans l'identité de réplication doivent être répliquées.
- Colonne virtuelles (v18) : jamais exportées

La réplication physique permet désormais de répliquer les modifications effectuées sur des colonnes générées stockées (pas virtuellles). Ce genre de colonne ne pose habituellement pas de problème pour des réplications entre serveurs PostgreSQL puisque le DDL déployé sur la souscription permet d'y regénérer les valeurs (à condition que ce DDL soit correctement synchronisé bien sûr). Pour des outils de *Change Data Capture*, c'est plus gênant. La cible ne sait à priori pas regénérer les données calculées.

Note : dans les exemples qui suivent, nous partons du principe que la configuration préalable de l'instance est effectuée (`wal_level`, `pg_hba.conf` et `.pg_pass` principalement).

Spécification d'une liste de colonne :

Voici le cas d'une table avec des colonnes générées :

```
CREATE TABLE postgresql (
   id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
   major_version TEXT UNIQUE,
   supported BOOLEAN,
   first_release DATE,
   final_release DATE GENERATED ALWAYS AS (first_release - INTERVAL '5 years' )
 ↪  STORED,
   designation TEXT GENERATED ALWAYS AS ('PostgreSQL ' || major_version ) STORED
);

INSERT INTO postgresql (major_version, supported, first_release)
  VALUES
    ('17', true, '2024-09-26'::DATE),
    ('16', true, '2023-09-14'::DATE),
    ('15', true, '2022-10-13'::DATE),
    ('14', true, '2021-09-30'::DATE),
```

```
    ('13', true, '2020-09-24'::DATE),
    ('12', true, '2020-10-03'::DATE);
```

Nous créons une publication en prenant soin d'inclure les colonnes générées :

```
CREATE PUBLICATION pgversions
  FOR TABLE postgresql (id, designation, first_release);
```

Dans la description de la publication, nous pouvons noter la colonne `Generated columns` qui est à none, et sur laquelle nous reviendrons plus tard :

```
\dRp+
                                Publication pgversions
 Owner  | All tables | Inserts | Updates | Deletes | Truncates | Generated columns | Via root
--------+------------+---------+---------+---------+-----------+-------------------+----------
 benoit | f          | t       | t       | t       | t         | none              | f
Tables:
    "public.postgresql" (id, first_release, designation)
```

Nous allons d'abord utiliser le plugin de décodage logique `test_decoding`. Il a besoin que l'on crée à la main le slot de réplication logique avec cette commande :

```
SELECT *
  FROM pg_create_logical_replication_slot('decode_generated_cols', 'test_decoding',
   ↪  false, true);
```

Mettons à jour les données :

```
INSERT INTO postgresql (major_version, supported, first_release)
  VALUES ('18', true, '2025-09-25'::DATE);

UPDATE postgresql
  SET supported = false
WHERE major_version = '12';

TABLE postgresql;
```

```
 id | major_version | supported | first_release | final_release |  designation
----+---------------+-----------+---------------+---------------+---------------
  1 | 17            | t         | 2024-09-26    | 2019-09-26    | PostgreSQL 17
  2 | 16            | t         | 2023-09-14    | 2018-09-14    | PostgreSQL 16
  3 | 15            | t         | 2022-10-13    | 2017-10-13    | PostgreSQL 15
  4 | 14            | t         | 2021-09-30    | 2016-09-30    | PostgreSQL 14
  5 | 13            | t         | 2020-09-24    | 2015-09-24    | PostgreSQL 13
  7 | 18            | t         | 2025-09-25    | 2020-09-25    | PostgreSQL 18
  6 | 12            | f         | 2020-10-03    | 2015-10-03    | PostgreSQL 12
(7 rows)
```

Les modifications sont immédiatement visibles via le plugin, y compris les colonnes générées, ce que nous voulions :

```
SELECT encode(data, 'escape') as conv_data
  FROM pg_logical_slot_peek_binary_changes('decode_generated_cols', null, null);

-- Note : la sortie mise en forme
                                  conv_data
------------------------------------------------------------------------------
 BEGIN 49770
 table public.postgresql: INSERT: id[integer]:7
                                  major_version[text]:'18'
                                  supported[boolean]:true
                                  first_release[date]:'2025-09-25'
                                  final_release[date]:'2020-09-25'
                                  designation[text]:'PostgreSQL 18'
 COMMIT 49770
 BEGIN 49771
 table public.postgresql: UPDATE: id[integer]:6
                                  major_version[text]:'12'
                                  supported[boolean]:false
                                  first_release[date]:'2020-10-03'
                                  final_release[date]:'2015-10-03'
                                  designation[text]:'PostgreSQL 12'
 COMMIT 49771

(6 rows)
```

Comme second exemple illustrant la synchronisation initiale, créons une souscription depuis un autre serveur en ne redéfinissant qu'une des colonnes générées :

```
-- Serveur 1
CREATE ROLE repli WITH LOGIN REPLICATION;
\password repli
GRANT SELECT ON postgresql TO repli;

-- Serveur 2
CREATE TABLE postgresql (
   id SERIAL PRIMARY KEY,
   designation TEXT,
   first_release DATE,
   final_release DATE GENERATED ALWAYS AS (first_release - INTERVAL '5 years' )
 ↪  STORED
);

CREATE SUBSCRIPTION pgversions
  CONNECTION 'host=/var/run/postgresql port=5435 user=repli dbname=benoit
 ↪  application_name=serveur2'
  PUBLICATION pgversions;

-- Attendre un peu
```

```
TABLE postgresql;
```

On constate que `designation` a bien été alimenté depuis la publication alors que `final_release` a été recalculé.

```
 id |  designation  | first_release | final_release
----+---------------+---------------+---------------
  1 | PostgreSQL 17 | 2024-09-26    | 2019-09-26
  2 | PostgreSQL 16 | 2023-09-14    | 2018-09-14
  3 | PostgreSQL 15 | 2022-10-13    | 2017-10-13
  4 | PostgreSQL 14 | 2021-09-30    | 2016-09-30
  5 | PostgreSQL 13 | 2020-09-24    | 2015-09-24
  7 | PostgreSQL 18 | 2025-09-25    | 2020-09-25
  6 | PostgreSQL 12 | 2020-10-03    | 2015-10-03
(7 rows)
```

Nouveau paramètre : `publish_generated_columns` :

Le paramétrage des publications a également été enrichi d'un nouveau paramètre : `publish_generated_columns`. S'il est configuré à `stored`, les colonnes générées stockées sont publiées par défaut. `none`, la valeur par défaut, désactive leur publication.

La liste des colonnes à publier a la priorité sur ce nouveau paramètre, si elle est présente, comme dans les premiers exemples plus haut.

Créons une nouvelle table avec une colonne générée, et modifions la publication pour y ajouter cette table sans spécifier de colonne et configurer `publish_generated_columns` à `stored`.

```
-- Serveur 1
CREATE TABLE doc (
   id SERIAL PRIMARY KEY,
   major_version TEXT UNIQUE,
   link TEXT GENERATED ALWAYS AS ('https://www.postgresql.org/docs/' ||
    ↪  major_version ) STORED
);
INSERT INTO doc(major_version) SELECT generate_series(12, 18)::text;
GRANT SELECT ON doc TO repli;

ALTER PUBLICATION pgversions ADD TABLE doc;
ALTER PUBLICATION pgversions SET (publish_generated_columns = stored);
```

La méta-commande `\dRp+` permet de visualiser la changement de configuration :

```
                                Publication pgversions
 Owner  | All tables | Inserts | Updates | Deletes | Truncates | Generated columns | Via root
--------+------------+---------+---------+---------+----------+-------------------+----------
 benoit | f          | t       | t       | t       | t        | stored            | f
Tables:
    "public.doc"
    "public.postgresql" (id, first_release, designation)
```

La table doit aussi être créée sur l'instance abonnée et la publication rafraîchie sur la souscription.

```
-- Serveur 2
CREATE TABLE doc (
   id SERIAL PRIMARY KEY,
   major_version TEXT,
   link TEXT
);
ALTER SUBSCRIPTION pgversions REFRESH PUBLICATION;

-- Attendre un peu
TABLE doc;
```

On constate, comme prévu, que les valeurs des colonnes générées ont été répliquées :

```
 id | major_version |                link
----+---------------+-----------------------------------
  1 | 12            | https://www.postgresql.org/docs/12
  2 | 13            | https://www.postgresql.org/docs/13
  3 | 14            | https://www.postgresql.org/docs/14
  4 | 15            | https://www.postgresql.org/docs/15
  5 | 16            | https://www.postgresql.org/docs/16
  6 | 17            | https://www.postgresql.org/docs/17
  7 | 18            | https://www.postgresql.org/docs/18
(7 rows)
```

Cas particulier :

Si un `replica identity` est défini sur une table et qu'il inclut une colonne générée stockée, il faut que cette colonne soit répliquée. Dans le cas contraire les UPDATE et DELETE échoueront. (Rappelons que dans l'idéal, la réplication se fait sur une clé primaire ou d'unicité.)

Voici un exemple (très artificiel) pour lequel nous allons créer une nouvelle table :

```
-- Serveur 1
CREATE TABLE test(
  id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  id2 int GENERATED ALWAYS AS ( id + 1) STORED UNIQUE NOT NULL,
  data text
);
ALTER TABLE test REPLICA IDENTITY USING INDEX test_id2_key;
INSERT INTO test (data) VALUES ('some data');
```

Utiliser `REPLICA IDENTITY FULL` aurait le même effet pour cette démonstration puisqu'il prend l'ensemble de la ligne comme identité de réplication.

Modifions la publication en retirant les anciennes tables, en ajoutant la nouvelle et en désactivant `publish_generated_columns` :

```
ALTER PUBLICATION pgversions DROP TABLE postgresql, doc;
ALTER PUBLICATION pgversions ADD TABLE test;
ALTER PUBLICATION pgversions SET (publish_generated_columns = none);
```

Créons ensuite la table sur l'autre serveur et rafraîchissons la souscription :

```
-- Serveur 2
CREATE TABLE test(
  id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  id2 int GENERATED ALWAYS AS ( id + 1) STORED UNIQUE NOT NULL,
  data text
);

ALTER SUBSCRIPTION pgversions REFRESH PUBLICATION;
```

Réaliser un UPDATE ou un DELETE est alors impossible car l'identité de réplication ne fait pas partie des données publiées :

```
-- Serveur 1
UPDATE test SET data = 'other data' WHERE id = 1;

ERROR:  cannot update table "test"
DETAIL:  Replica identity must not contain unpublished generated columns.
```

NB : Tout ce qui précède ne concerne pas les colonnes générées *virtuelles*, apparues aussi dans PostgreSQL 18, qui sont complètement ignorées par la réplication.

4.3 RÉPLICATION LOGIQUE - LIMITER LE NOMBRE DE SOUSCRIPTIONS

- Nouveau paramètre `max_active_replication_origins`
- Nombre maximal d'origines de réplication actives sur un serveur
- Limite le nombre de souscriptions et de *table sync workers*
- Précédemment limité par `max_replication_slots`
- Configuré à `10` par défaut

Qu'est ce qu'une origine de réplication ?

Les origines de réplication consistent en un identifiant interne (`roident`) associé à un identifiant externe (`roname`). Ils ont pour objectif de permettre un traçage efficace, persistant et durable de la progression de la réplication logique.

```
TABLE pg_replication_origin \gx
```

```
-[ RECORD 1 ]-----
roident | 1          -> oid
roname  | pg_16510   -> text
```

Le nom de l'origine de réplication est utilisé pour la communication entre systèmes. Il faut donc qu'ils soit unique pour éviter une confusion dans les origines de réplication. C'est pour cette raison que le slot contient ici l'`oid` de la souscription.

```
TABLE pg_subscription \gx
```

```
-[ RECORD 1 ]-------+---------
oid                 | 16510
subdbid             | 16385
...
origin              | any
```

Les informations sur la progression de la réplication logique sont stockées en mémoire partagée et écrites sur disque à chaque *checkpoint*. Le *LSN* (*log sequence number*, la position dans les journaux de transaction) sur le serveur d'origine est conservé et ajouté dans les messages de `COMMIT` écrits dans les journaux de la souscription. Cela permet de faire avancer la progression lors du *recovery* après un crash.

Voici l'exemple d'une insertion dans une table `cible` sur la publication d'un dispositif de réplication logique :

```
INSERT INTO cible VALUES (2, 'blah');
SELECT pg_current_wal_lsn();
```

```
 pg_current_wal_lsn
--------------------
 0/6D001CF8
(1 row)
```

Sur le serveur abonné, dans les WAL, on voit :

```
/usr/lib/postgresql/18/bin/pg_waldump 000000010000000000000001C

rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/1CA7D580, prev
 ↪  0/1CA7D508, desc: RUNNING_XACTS nextXid 860 latestCompletedXid 859
 ↪  oldestRunningXid 860
rmgr: Heap        len (rec/tot):     57/   301, tx:        860, lsn: 0/1CA7D5B8, prev
 ↪  0/1CA7D580, desc: INSERT off: 5, flags: 0x00, blkref #0: rel 1663/16385/16507
 ↪  blk 0 FPW
rmgr: Transaction len (rec/tot):     57/    57, tx:        860, lsn: 0/1CA7D6E8, prev
 ↪  0/1CA7D5B8, desc: COMMIT 2025-10-03 15:39:36.598054 CEST; origin: node 1, lsn
 ↪  0/6D001CF8, at 2025-10-03 15:39:36.590290 CEST
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/1CA7D728, prev
 ↪  0/1CA7D6E8, desc: RUNNING_XACTS nextXid 861 latestCompletedXid 860
 ↪  oldestRunningXid 861
```

Voici le contenu de la vue `pg_replication_origin_status`, qui permet de suivre la progression de la réplication synchrone :

```
TABLE pg_replication_origin_status \gx

-[ RECORD 1 ]-----------
local_id    | 1
external_id | pg_16510
remote_lsn  | 0/6D001CF8
local_lsn   | 0/1CA7D728
```

On observe que la ligne `COMMIT` que l'on voit dans le retour de la commande pg_wadump, contient le *LSN* : `0/6D001CF8`, associé au `node 1`. Ce node correspond à l'origine de réplication avec l'identifiant `1` (voir `pg_replication_origin_status.local_id` et `pg_replication_origin.roident`). Ce *LSN* correspond bien à celui obtenu après l'insertion sur la publication qui est également enregistré dans la vue de progression sous le nom `remote_lsn`.

On voit aussi que le *LSN* local après l'application de ce *COMMIT* est `0/1CA7D728` comme l'indique la colonne `local_lsn` de la vue de progression.

À titre d'exemple, voici un `COMMIT` déclenché directement sur le serveur de souscription dans le fonctionnement normal de l'instance :

```
rmgr: Transaction len (rec/tot):    373/   373, tx:        861, lsn: 0/1CA95B70, prev
↪ 0/1CA95B28, desc: COMMIT 2025-10-06 09:25:28.575205 CEST; inval msgs: catcache
↪ 82 catcache 81 catcache 82 catcache 81 catcache 57 catcache 56 catcache 7
↪ catcache 6 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache 6
↪ catcache 7 catcache 6 catcache 7 catcache 6 snapshot 2608 relcache 16534
```

Les origines de réplication sont notamment utilisées conjointement avec le décodage logique pour détecter des boucles dans la réplication bidirectionnelle. Si la souscription est créée avec `WITH (origin = none)` seules les modifications des WAL qui ne portent pas d'origine de réplication seront décodées. Dans le cas d'une réplication croisée, cela évite donc de répliquer à nouveau des modifications provenant de l'autre serveur (c'est ce qui se passerait avec `WITH (origin = any)` qui est la configuration par défaut).

Si vous remontez plus haut, vous verrez que la vue `pg_subscription` contient une colonne `origin` qui est valorisée à `any`.

Note : La réplication bidirectionnelle ne permet pas de créer un serveur PostgreSQL multimaître complet. Au mieux facilite-t-elle les échanges mutuels entre des instances dont chacune gère un sous-ensemble strict des données. En effet, entre autres choses, PostgreSQL ne dispose pas encore de mécanisme de résolution des conflits, ou de réplication des séquences (peut-être dans une prochaine version ?).

Nouveau paramètre :

Le nombre maximal d'origines de réplication était jusqu'en PostgreSQL 17 contrôlé via le paramètre `max_replication_slots`. Ce n'était pas satisfaisant puisque le nombre de souscription n'a pas de lien direct avec le nombre de slots de réplication possibles et ne permet pas un contrôle précis du nombre de souscriptions. En effet dans certains cas, les souscriptions ne nécessitent pas de slots, mais requièrent toujours une origine de réplication.

Le nouveau paramètre `max_active_replication_origins` permet de contrôler directement le nombre de souscription et est valorisé à `10` par défaut. C'est la valeur par défaut de `max_replication_slots`, ce nouveau paramètre ne provoque donc pas de régression.

4.4 RÉPLICATION LOGIQUE - CONFLITS DE RÉPLICATION

- Détection de 7 types de conflits de réplication logique
- 7 nouvelles colonnes dans `pg_stat_subscription_stats`
  - comptabilise les types de conflits
- Nouveaux messages d'informations et d'erreurs associés

Le « multimaître » semble être le graal de la réplication logique. Pour y arriver, il faut entre autre ajouter à PostgreSQL un mécanisme de résolution des conflits. Les fonctionnalités décrites dans cet article et ajoutées dans la version 18 s'attellent à implémenter la détection de conflit de réplication logique. Leur résolution sera intégrée dans une future version.

Les mises à jour issues de la réplication logique peuvent échouer si des conflits sont détectés : ce genre d'événement bloque la réplication. Certaines opérations, comme la mise à jour ou la suppression d'une donnée qui n'est pas présente, ne génèrent pas d'erreur : l'opération est simplement ignorée. Pourtant ce genre d'évènement est aussi un conflit de réplication. La version 18 catégorise les conflits en sept groupes, les détecte, collecte des statistiques à leur sujet dans la vue `pg_stat_subscription_stats` et émet des messages les concernant dans les traces de l'instance.

Voici la liste de ces conflits :

- `update_missing` : la ligne mise à jour est manquante ;
- `delete_missing` : la ligne supprimée est manquante ;
- `insert_exists` : la ligne insérée viole une contrainte unique non déferrable ;
- `update_exists` : la ligne mise à jour viole une contrainte unique non déferrable ;
- `multiple_unique_conflicts` : la ligne insérée ou mise à jour viole plusieurs contraintes uniques non déferrables ;
- `update_origin_ differ` : la ligne mise à jour a été précédemment modifiée par une autre origine ;
- `delete_origin_differ` : la ligne supprimée a été modifiée précédemment par une autre origine.

Les deux derniers types de conflits nécessitent `track_commit_timestamp` pour être détectés. Attention, il n'est pas activé par défaut. Les trois précédents nécessitent l'activation de ce paramètre pour compléter les messages avec des infos sur l'origine du commit et sur son timestamp.

La suite de l'article présente ces conflits plus en détail :

- `update_missing` : la ligne mise à jour est manquante.

  Mise en place du test :

```
-- Serveur 1
CREATE ROLE repli WITH LOGIN REPLICATION PASSWORD 'repli';
CREATE TABLE conflits (i int PRIMARY KEY, t text);
CREATE PUBLICATION srv1 FOR TABLE conflits;
GRANT SELECT ON conflits TO repli;
INSERT INTO conflits(i,t) VALUES (1, 'first');

-- Serveur 2
CREATE TABLE conflits (i int PRIMARY KEY, t text);
CREATE SUBSCRIPTION from_srv1
  CONNECTION 'host=/var/run/postgresql port=5435 user=repli dbname=benoit
↪  application_name=srv2'
  PUBLICATION srv1;
```

  La vue de supervision ne contient pour le moment pas d'erreur :

```
-- Serveur 2
TABLE pg_stat_subscription_stats \gx

-[ RECORD 1 ]-------------------+----------
subid                          | 112863
subname                        | from_srv1
apply_error_count              | 0
sync_error_count               | 0
confl_insert_exists            | 0
confl_update_origin_differs    | 0
confl_update_exists            | 0
confl_update_missing           | 0
confl_delete_origin_differs    | 0
confl_delete_missing           | 0
confl_multiple_unique_conflicts | 0
stats_reset                    | ø
```

  Si la ligne 1 est supprimée sur le **serveur 2** avant d'être mise à jour sur le **serveur 1** :

```
-- Serveur 2
DELETE FROM conflits WHERE i = 1;

-- Serveur 1
UPDATE conflits SET t = 'updated' WHERE i = 1;
```

  Nous pouvons observer qu'un conflit est tracé dans la vue sur le **serveur 2** :

```
-[ RECORD 1 ]-------------------+----------
```

```
subid                           | 112863
subname                         | from_srv1
apply_error_count               | 0
sync_error_count                | 0
confl_insert_exists             | 0
confl_update_origin_differs     | 0
confl_update_exists             | 0
confl_update_missing            | 1
confl_delete_origin_differs     | 0
confl_delete_missing            | 0
confl_multiple_unique_conflicts | 0
stats_reset                     | ø
```

L'update a été ignoré, et un message est écrit dans les traces du **serveur 2** nous informant du conflit :

```
[1223274] LOG:  conflict detected on relation "public.conflits":
 ↪  conflict=update_missing
[1223274] DETAIL:  Could not find the row to be updated.
 Remote row (1, updated); replica identity (i)=(1).
[1223274] CONTEXT:  processing remote data for replication origin "pg_112863"
 ↪  during message type "UPDATE" for replication target relation
 ↪  "public.conflits" in transaction 49504, finished at 0/224E8158
```

Le message permet d'identifier le type de conflit et la table (ligne 1 et 2), la ligne sur la source (ligne 3). La dernière ligne permet d'identifier l'origine de réplication et donc la souscription (puisque l'origine est nommée `pg_<oid souscription>`).

- `delete_missing` : la ligne supprimée est manquante.

  Si nous supprimons maintenant la ligne 1 de l'exemple précédent sur le **serveur 1** :

```
-- Serveur 1
DELETE FROM conflits WHERE i = 1;
```

  Nous pouvons observer que le conflit est tracé dans la vue sur le **serveur 2** :

```
-[ RECORD 1 ]-------------------+----------
subid                           | 112863
subname                         | from_srv1
apply_error_count               | 0
sync_error_count                | 0
confl_insert_exists             | 0
confl_update_origin_differs     | 0
confl_update_exists             | 0
confl_update_missing            | 1
confl_delete_origin_differs     | 0
confl_delete_missing            | 1
confl_multiple_unique_conflicts | 0
```

```
stats_reset                      | ø
```

Le delete a été ignoré et un message est écrit dans les traces du **serveur 2** nous informant du conflit :

```
[1223274] LOG:  conflict detected on relation "public.conflits":
 ↪  conflict=delete_missing
[1223274] DETAIL:  Could not find the row to be deleted.
 Replica identity (i)=(1).
[1223274] CONTEXT:  processing remote data for replication origin "pg_112863"
 ↪  during message type "DELETE" for replication target relation
 ↪  "public.conflits" in transaction 49505, finished at 0/224E8200
```

- `insert_exists` : la ligne insérée viole une contrainte d'unicité non déferrable.

  Insérons maintenant une ligne sur le **serveur 2** puis sur le **serveur 1** :

```
--- Serveur 2
INSERT INTO conflits(i,t) VALUES (1, 'conflit');

--- Serveur 1
INSERT INTO conflits(i,t) VALUES (1, 'first');
```

  Dans la vue `pg_stat_subscription_stats`, nous voyons que les colonnes `apply_error_count` et `confl_insert_exists` s'incrémentent au fil du temps. C'est normal, car contrairement aux deux premiers exemples l'`INSERT` ne peut être ignoré. PostgreSQL retente donc de l'appliquer indéfiniment à moins que l'option `disable_on_error` de la souscription ne soit configurée à `true`, auquel cas la souscription est désactivée.

  Dans les traces de l'instance du **serveur 2**, le message d'erreur suivant nous informe du conflit :

```
[1224136] ERROR:  conflict detected on relation "public.conflits":
 ↪  conflict=insert_exists
[1224136] DETAIL:  Key already exists in unique index "conflits_pkey", modified
 ↪  locally in transaction 80750 at 2025-10-09 11:59:33.845548+02.
        Key (i)=(1); existing local row (1, conflit); remote row (1, first).
[1224136] CONTEXT:  processing remote data for replication origin "pg_112863"
 ↪  during message type "INSERT" for replication target relation
 ↪  "public.conflits" in transaction 49506, finished at 0/224E82F0
```

  Le message permet la encore d'identifier le type de conflit et la table en ligne 1. Le nom de la contrainte violée est ensuite mentionné avec le contenu des lignes sur la source et la cible. Lorsque `track_commit_timestamp` est désactivé (c'est le défaut), la mention `at 2025-10-09 11:59:33.845548+02` en ligne 2 n'est pas incluse dans le message.

  Les messages suivant montrent que le *background worker* responsable d'appliquer les modifications redémarre pour échouer ensuite à nouveau, ce qui confirme ce que l'on observait dans la vue :

```
[1024037] LOG:  background worker "logical replication apply worker" (PID
 ↪  1224136) exited with exit code 1
[1224153] LOG:  logical replication apply worker for subscription "from_srv1"
 ↪  has started
```

Il faut supprimer la ligne sur le **serveur 2** pour que la réplication reprenne.

```
-- Serveur 2
DELETE FROM conflits WHERE i = 1;
```

- `update_exists` : la ligne mise à jour viole une contrainte d'unicité non déferrable.

  Pour observer ce conflit, les requêtes suivantes doivent être exécutées :

  ```
  -- Serveur 2
  INSERT INTO conflits(i,t) VALUES (2, 'conflit');

  -- Serveur 1
  UPDATE  conflits SET i = 2 WHERE i = 1;
  ```

  Cette fois-ci, ce sont les colonnes `apply_error_count` et `confl_update_exists` qui sont incrémentées. Là encore, le nombre d'erreurs augmente indéfiniment avec la configuration actuelle.

  Cette fois-ci dans les traces de **serveur 2**, on voit l'erreur suivante :

  ```
  [1235951] ERROR:  conflict detected on relation "public.conflits":
   ↪  conflict=update_exists
  [1235951] DETAIL:  Key already exists in unique index "conflits_pkey", modified
   ↪  locally in transaction 80867 at 2025-10-09 14:40:36.769303+02.
          Key (i)=(2); existing local row (2, conflit); remote row (2, first);
   ↪  replica identity (i)=(1).
  [1235951] CONTEXT:  processing remote data for replication origin "pg_112863"
   ↪  during message type "UPDATE" for replication target relation
   ↪  "public.conflits" in transaction 49508, finished at 0/224E86C8
  ```

  Pour ce message aussi, l'activation du paramètre `track_commit_timestamp` permet de tracer le timestamp de commit de la transaction en ligne 2.

  Là encore, le *background worker* s'arrête et un nouveau est démarré. Le cycle continue tant que le problème persiste :

  ```
  [1024037] LOG:  background worker "logical replication apply worker" (PID
   ↪  1235951) exited with exit code 1
  [1235954] LOG:  logical replication apply worker for subscription "from_srv1"
   ↪  has started
  ```

  Il faut supprimer la ligne sur le **serveur 2** pour que la réplication reprenne.

  ```
  -- Serveur 2
  DELETE FROM conflits WHERE i = 2;
  ```

- `multiple_unique_conflicts` : la ligne insérée ou mise à jour viole plusieurs contraintes uniques non déferrables.

  Avant l'ajout de ce conflit, PostgreSQL s'arrêtait au premier conflit. Si une opération entrait en conflit avec plusieurs contraintes, il fallait donc s'y reprendre plusieurs fois avant de corriger complètement le problème. L'expérience utilisateur était donc mauvaise.

  Pour démontrer le conflit, nous allons reproduire le cas précédent avec des contraintes supplémentaires sur la table :

  ```
  -- Serveur 1 et 2
  ALTER TABLE conflits ADD UNIQUE (t);

  -- Serveur 2
  INSERT INTO conflits(i,t) VALUES (1, 'second');

  -- Serveur 1
  UPDATE  conflits SET i = 1, t = 'second' WHERE i = 2;
  ```

  Cette fois, le message est différent dans les traces de l'instance : il liste l'ensemble des contraintes violées.

  ```
  [1319115] ERROR:  conflict detected on relation "public.conflits":
   ↪  conflict=multiple_unique_conflicts
  [1319115] DETAIL:  Key already exists in unique index "conflits_pkey", modified
   ↪  locally in transaction 83328 at 2025-10-10 16:39:28.91569+02.
          Key (i)=(1); existing local row (1, second); remote row (1, second);
   ↪  replica identity (i)=(2).
         Key already exists in unique index "conflits_t_key", modified locally in
   ↪  transaction 83328 at 2025-10-10 16:39:28.91569+02.
          Key (t)=(second); existing local row (1, second); remote row (1,
   ↪  second); replica identity (i)=(2).
  [1319115] CONTEXT:  processing remote data for replication origin "pg_112863"
   ↪  during message type "UPDATE" for replication target relation
   ↪  "public.conflits" in transaction 49625, finished at 0/2E316CA0
  ```

  Les lignes 2 et 4 contiennent des *timestamps* qui ne sont visibles que si `track_commit_timestamp` est activé.

  Là encore, le *background worker* s'arrête et un nouveau est démarré. Le cycle continue tant que le problème persiste :

  ```
  [1024037] LOG:  background worker "logical replication apply worker" (PID
   ↪  1319115) exited with exit code 1
  [1319131] LOG:  logical replication apply worker for subscription "from_srv1"
   ↪  has started
  ```

  Les compteurs `confl_multiple_unique_conflicts` et `apply_error_count` de la vue de statistique sont incrémentés.

Afin de poursuivre la démonstration, vidons la table sur le **serveur 1** et le **serveur 2** et retirons la contrainte d'unicité sur la colonne `t` :

```
-- Serveur 1 et 2
TRUNCATE conflits;
ALTER TABLE conflits DROP CONSTRAINT conflits_t_key;
```

Avant de de voir les deux derniers types de conflits, nous allons devoir compléter l'architecture et mettre en place des réplications bidirectionnelles.

Nous allons commencer par compléter la configuration des **serveurs 1** et **2** en créant l'utilisateur de réplication et la publication sur le **serveur2** :

```
-- Serveur 2
CREATE ROLE repli WITH LOGIN REPLICATION PASSWORD 'repli';
CREATE PUBLICATION srv2 FOR TABLE conflits;
GRANT SELECT ON conflits TO repli;
```

Nous allons maintenant compléter en créant la souscription sur le **serveur1** qui consomme les modifications depuis la publication du **serveur 2**. Profitons de la mise en place de cette réplication bidirectionnelle pour illuster l'importance des origines de réplication et du paramètre de souscription `origin`. Il est justement apparu dans PostgreSQL 16 pour faciliter des réplications logiques croisées. Par défaut, il vaut `any` :

```
-- Serveur 1
CREATE SUBSCRIPTION from_srv2
  CONNECTION 'host=/var/run/postgresql port=5436 user=repli dbname=benoit
 ↪  application_name=srv1'
  PUBLICATION srv2;
```

Testons une insertion :

```
-- Serveur 1
INSERT INTO conflits VALUES (1, 'first');
```

Le message d'erreur suivant est visible dans les traces du **serveur 1**, il s'agit d'un message que nous avons déjà rencontré précédemment :

```
[1252871] ERROR:  conflict detected on relation "public.conflits":
 ↪  conflict=insert_exists
[1252871] DETAIL:  Key already exists in unique index "conflits_pkey", modified
 ↪  locally in transaction 49569 at 2025-10-09 17:45:34.455884+02.
        Key (i)=(1); existing local row (1, first); remote row (1, first).
[1252871] CONTEXT:  processing remote data for replication origin "pg_123045" during
 ↪  message type "INSERT" for replication target relation "public.conflits" in
 ↪  transaction 82761, finished at 0/8633678
[1023946] LOG:  background worker "logical replication apply worker" (PID 1252871)
 ↪  exited with exit code 1
```

À ce stade, la réplication est bloquée.

La ligne CONTEXT nous informe que l'insert provient de l'origine `pg_123045`. La requête suivante exécutée sur le **serveur 1** montre qu'il s'agit de origine de réplication rattaché à la souscription nommée `from_srv2`.

```
SELECT ro.*, sub.oid, sub.subname
  FROM pg_replication_origin AS ro, pg_subscription AS sub;
```

```
 roident |  roname   |  oid   |  subname
---------+-----------+--------+-----------
       1 | pg_123045 | 123045 | from_srv2
(1 row)
```

L'insertion est donc exécutée sur le **serveur 1** et répliquée sur le **serveur 2**. Elle est ensuite redécodée sur le **serveur 2** et renvoyée au **serveur 1** qui ne peut appliquer la modification car la ligne est déjà présente, ce qui provoque une violation de contrainte d'unicité.

Pour corriger le problème, il faut modifier la souscription pour qu'elle ne récupère que les modifications qui n'ont pas d'origine de réplication (en d'autres termes, celles qui sont issues des requêtes DML et des TRUNCATE du serveur qui fait le décodage logique) :

```
-- Serveur 1
ALTER SUBSCRIPTION from_srv2 SET (origin = none);

-- Serveur 2
ALTER SUBSCRIPTION from_srv1 SET (origin = none);
```

Cela débloque la réplication et règle le problème.

Vidons la table sur le **serveur 1** et le **serveur 2** :

```
-- Serveur 1 et 2
TRUNCATE conflits;
```

Ajoutons maintenant une troisieme instance dont les modifications sont consommées par avec le **serveur 2** !

```
-- Serveur 3
CREATE ROLE repli WITH LOGIN REPLICATION PASSWORD 'repli';
CREATE TABLE conflits (i int PRIMARY KEY, t text);
CREATE PUBLICATION srv3 FOR TABLE conflits;
GRANT SELECT ON conflits TO repli;

-- Serveur 2
CREATE SUBSCRIPTION from_srv3
  CONNECTION 'host=/var/run/postgresql port=5437 user=repli dbname=benoit
↪  application_name=srv2'
  PUBLICATION srv3
  WITH (origin=none);
```

Nous pouvons maintenant nous intéresser aux deux derniers types de conflits :

- `update_origin_ differ` : la ligne mise à jour a été précédemment modifiée par une autre origine.

  ```
  -- Serveur 3
  INSERT INTO conflits(i,t) VALUES (1, 'from srv3');
  ```

  L'ordre est répliqué vers le **serveur 2** mais pas le **serveur 3** à cause du paramètre `origin = none` des souscriptions.

  ```
  -- Serveur 2
  DELETE FROM conflits;

  -- Serveur 1
  INSERT INTO conflits(i,t) VALUES (1, 'from srv1');
  ```

  La ligne est bien répliquée sur le **serveur 2** :

  ```
  -- Server 2
  TABLE conflits ;

   i |     t
  ---+-----------
   1 | from srv1
  (1 row)

  -- Serveur 3
  UPDATE conflits SET t = 'from srv3 !!' WHERE i = 1;
  ```

  À nouveau, la ligne est bien répliquée sur le **serveur 2** :

  ```
  -- Server 2
  TABLE conflits ;

   i |     t
  ---+--------------
   1 | from srv3 !!
  (1 row)
  ```

  Dans la vue `pg_stat_subscription_stats`, la colonne `confl_update_origin_differs` a été incrémentée pour la souscription `from_srv3`. Dans les traces apparaît également le message suivant :

  ```
  [1301971] LOG:  conflict detected on relation "public.conflits":
   ↪  conflict=update_origin_differs
  [1301971] DETAIL:  Updating the row that was modified by a different origin
   ↪  "pg_112863" in transaction 83317 at 2025-10-10 14:13:58.64601+02.
         Existing local row (1, from srv1); remote row (1, from srv3 !!); replica
   ↪  identity (i)=(1).
  ```

```
[1301971] CONTEXT:  processing remote data for replication origin "pg_112871"
 ↪  during message type "UPDATE" for replication target relation
 ↪  "public.conflits" in transaction 792, finished at 0/1C86ED8
```

Le message nous donne une information supplémentaire : le nom de l'origine de réplication responsable de la précédente modification de la ligne (`pg_112863`).

Ce type de conflit est complètement ignoré si `track_commit_timestamp` est désactivé.

La mise à jour ne provoque pas d'erreur lors de son exécution, mais provoque une incohérence entre les données du **serveur 1** et du **serveur 2**. Pour le moment PostgreSQL ne propose pas de solution pour réagir à ce conflit. Il s'agit d'un second volet de la fonctionnalité qui sera ajouté dans une version future.

- `delete_origin_differ` : la ligne supprimée a été modifiée précédemment par une autre origine.

```
-- Serveur 1
DELETE FROM conflits WHERE i = 1;
```

La ligne est bien supprimée sur le **serveur 2**.

Dans la vue `pg_stat_subscription_stats` la colonne `confl_delete_origin_differs` a été incrémentée pour la souscription `from_srv1`. Dans les traces apparaît également le message suivant :

```
[1253168] LOG:  conflict detected on relation "public.conflits":
 ↪  conflict=delete_origin_differs
[1253168] DETAIL:  Deleting the row that was modified by a different origin
 ↪  "pg_112871" in transaction 83318 at 2025-10-10 14:14:12.814603+02.
        Existing local row (1, from srv3 !!); replica identity (i)=(1).
[1253168] CONTEXT:  processing remote data for replication origin "pg_112863"
 ↪  during message type "DELETE" for replication target relation
 ↪  "public.conflits" in transaction 49620, finished at 0/2E2F34A0
```

Comme l'autre conflit lié aux origines de réplication, ce type de conflit est complètement ignoré si `track_commit_timestamp` est désactivé.

5/ Performances

5.1 ASYNCIO

- IO asynchrones
  - PostgreSQL connaît mieux le contexte que l'OS
  - 15: `recovery_prefetch`
  - 16: extension des relations
  - 17: infrastructure pour combiner des IO (`io_combine_limit`)
  - 18: infrastructure pour les IO asynchrone (`io_method`)
- De gros gains
  - *Bitmap Scan* en premier lieu
  - en lecture uniquement
- *Direct IO* (pas "encore" disponible)
  - 16: `debug_io_direct`

Pourquoi utiliser les IO directes et asynchrones ?

Jusqu'à maintenant, PostgreSQL se base sur l'OS pour l'ensemble de ses IO. PostgreSQL lui demande des blocs, que l'OS a déjà en cache, ou qu'il va chercher sur disque. Cependant un chantier est en cours pour passer, au moins partiellement, de ce mode d'accès aux données appelé *buffered IO*, à ce qu'on appelle des IO directes (ou *direct IO*[1]).

Dans beaucoup de cas, cela permet de déplacer les données directement du stockage vers le cache de PostgreSQL sans faire intervenir le CPU en utilisant la technologie DMA[2]. Cela réduit la charge CPU et diminue la latence des IO. Un autre avantage de ce changement est une meilleure utilisation de la mémoire, puisque l'on évite une double mise en cache de certaines données (OS + PostgreSQL). Pour finir, cela permet de mieux contrôler les IO en tirant partie du fait que PostgreSQL a une meilleure connaissance du contexte dans lequel les IO sont réalisées. Cela donne, par exemple, plus de contrôle sur le timing où les écritures sont effectivement faites sur le stockage (`fsync`).

Cependant, en donnant à PostgreSQL la main sur l'exécution des IO, on se passe des mécanismes en place au niveau du système d'exploitation comme la mise en cache des fichiers et le *kernel read ahead*. Seule, cette modification serait donc délétère pour les performances.

On peut facilement s'en rendre compte en expérimentant l'option `debug_io_direct`, qui a été

[1] *Direct IO*
[2] *Direct Memory Access*

ajoutée en version 16 et qui permet de demander à l'OS d'utiliser le cache au minimum pour les relations et/ou les WAL. Comme son nom l'indique, cette option est destinée aux développeurs de PostgreSQL, et surtout pas à la production.

Un préalable aux IO directes est donc de mettre en place toute une infrastructure pour réaliser des IO asynchrones (appelées AIO[3] dans la suite de ce texte) afin de grouper et de réaliser les IO en avance de phase (*prefetch*) et de manière concurrente.

Ces améliorations laissent espérer beaucoup d'améliorations de performances notamment dans le cadre des écritures et synchronisations des WAL.

Le parcours séquentiel d'une table avec plusieurs segments est un autre cas où le contexte des IO est important. En effet, pour le *kernel read ahead* un changement de fichiers provoque des fluctuations dans l'usage du *prefetch* car le noyau ne sait pas qu'il s'agit de la même table et que l'on va continuer à lire le segment et les suivants jusqu'au bout. PostgreSQL, lui, est au courant de ce contexte et pourra continuer le prefetch avec la même agressivité.

Pour finir, les IO directes et asynchrones nécessitent des quantités de *shared buffers* importantes pour fonctionner de manière optimale.

Mettre en place ce nouveau mode de fonctionnement touche donc énormément d'aspects du fonctionnement de PostgreSQL et il faudra plusieurs versions pour en tirer tous les bénéfices.

Certaines fonctionnalités exploitaient déja les io asynchrones du système d'exploitation comme la recovery en version 15[4] (`recovery_prefetch`) et les extensions de relations en masse en version 16. Plus ancien, le *bitmap heap scan* a longtemps été le seul cas où le *prefetch* était utilisé.

D'autres améliorations ont été faites en version 17, avec la création d'une nouvelle API permettant de regrouper plusieurs IO[5] en un seul appel système. Actuellement seules l'implémentation du parcours séquentiel (*Seq Scan*), celle de `ANALYZE`, et celle de l'extension `pg_prewarm` utilisent cette nouvelle API.

Cette nouvelle version continue donc l'effort engagé sur l'amélioration du système d'IO de PostgreSQL en ajoutant un système pour gérer les IO asynchrones qui s'intègrent avec les fonctionnalités existantes.

Le travail pour utiliser les IO directes ne viendra « que » plus tard.

---

[3] *Async IO*
[4] https://dali.bo/workshop15_html#permettre-le-pre-fetch-du-contenu-des-fichiers-wal-pendant-le-recovery
[5] https://dali.bo/workshop17_html#regroupement-des-io

5.2 ASYNCIO - NOUVEAUX PARAMÈTRES

Paramètre	Défaut	Remarque
io_combine_limit	128 ko	
io_max_combine_limit	128 ko	
io_max_concurrency	-1	Souvent 64
io_method	worker	ou io_uring,sync
io_workers	3	jusque 32

- `io_method` = `worker` à priori un bon défaut
- `io_workers` sans doute à monter

Paramétrage :

Voici les cinq paramètres relatifs à l'infrastructure d'AIO :

```
                    List of configuration parameters
     Parameter       |  Value |  Type   |  Context   | Access privileges
---------------------+--------+---------+------------+-------------------
 io_combine_limit    | 128kB  | integer | user       |
 io_max_combine_limit | 128kB  | integer | postmaster |
 io_max_concurrency  | 64     | integer | postmaster |
 io_method           | worker | enum    | postmaster |
 io_workers          | 3      | integer | sighup     |
```

Trois **modes de gestion des IO asynchrones** sont disponibles en version 18. Le changement de méthode nécessite un redémarrage :

- **`io_method=worker`**

  `io_method=worker` est disponible sur toutes les plateformes. Avec ce mode, les IO asynchrones sont dispatchées à des *io workers* gérés par PostgreSQL. Ils exécutent les IO de manière synchrone et attendent leur retour. Cela permet de simuler des IO asynchrones du point de vue des autres processus sans dépendre des capacités du système. C'est la méthode par défaut. Il faut redémarrer pour changer `io_method`.

Le nombre d'*io workers* dépend du paramètre `io_workers`. Il vaut 3 par défaut. Changer n'est pas possible dans une session, uniquement dans la configuration globale (`postgresql.conf`, `ALTER SYSTEM SET …`).

Théoriquement, ce mode est moins performant lorsqu'il y a beaucoup de petites IO avec une latence importante. En effet, l'utilisation de processus différents induit un coût supplémentaire dû aux changements de contextes. Pour finir, le nombre de workers étant fixe, il peut donc être un facteur limitant si la configuration n'est pas adaptée à la charge, d'autant plus qu'ils se chargent de certaines opérations comme la vérification des *checksums*, assurées dans les autres modes par chaque *backend*.

Pourtant, nous verrons plus bas que les workers sont généralement l'option à conserver par défaut.

Andres Freund rapporte des gains encore plus grands sur Windows[6].

Il est possible de voir les nouveaux *io workers* dans la liste de processus ci-dessous :

```
/usr/pgsql-18/bin/postgres -D /var/lib/pgsql/18/data/
 \_ postgres: logger
 \_ postgres: io worker 2
 \_ postgres: io worker 1
 \_ postgres: io worker 0
 \_ postgres: checkpointer
 \_ postgres: background writer
 \_ postgres: walwriter
 \_ postgres: autovacuum launcher
 \_ postgres: archiver last was 000000010000000000000001D
 \_ postgres: logical replication launcher
```

La capture de la vue `pg_stat_activity` qui suit, montre des backends en attente de terminaison d'AIO par un autre processus (`AioIoCompletion`). Les *io workers* sont en train de faire des IO (`DataFileRead`), attendre une demande d'IO (`IoWorkerMain`) ou sont dans une portion de code qui n'a pas d'évènement d'attente.

```
  backend_type   | application_name |   wait_event
-----------------+------------------+-----------------
 client backend | pgbench          | AioIoCompletion
 client backend | pgbench          | AioIoCompletion
 io worker      |                  | DataFileRead
 io worker      |                  |
 io worker      |                  | IoWorkerMain
```

La documentation décrit aussi l'évènement `AioWorkerSubmissionQueue` qui correspond à une attente dans la queue de soumission des *io_workers*. Cela pourrait indiquer que le nombre

[6] https://www.postgresql.org/message-id/brdaw5wke274lubirrl4v2k4qdacylvgwwqztifn7m27pkth3s%40rh7wie47pfcp

d'*io_workers* est insuffisant.

Comme les *io workers* peuvent être utilisés pendant l'arrêt du cluster, la séquence d'arrêt de PostgreSQL a été modifiée. Les *io workers* sont arrêtés après que le *shutdown checkpoint* ait terminé et que les processus *walsender* et *archiver* soient arrêtés, mais avant que le processus *checkpointer* soit arrêté.

Une autre raison de ce changement est qu'il est nécessaire d'avoir un processus capable de sérialiser les statistiques lorsque tous les processus sont arrêtés, chose que le *checkpointer* est capable de faire. Cela sera utile quand le processus *walsender* générera des statistiques lors de l'arrêt. Jusqu'à maintenant, le *walsender* s'arrêtait après le *checkpointer*.

- **`io_method=io_uring`**

  `io_method=io_uring` n'est disponible que sur Linux et uniquement pour les noyaux dont la version est plus récente que la 5.1 (parue en 2019). Ce mode nécessite l'utilisation de la bibliothèque `liburing`. Dans ce mode, les IO sont déclenchées directement par les backends.

  Afin de pouvoir activer `io_uring`, il faut également que la configuration du noyau le permette. Le paramètre `kernel.io_uring_disabled` sert à contrôler ce comportement et peut prendre trois valeurs :

  - `0` : permet à tout processus d'utiliser *io_uring* (défaut sur Debian 13, Ubuntu 24.04) ;
  - `1` : désactive *io_uring* pour les processus qui ne sont pas lancés par le groupe `io_uring_group` ;
  - `2` : désactive *io_uring* pour tous les processus (défaut sur Rocky Linux 9 et 10)

  Le paramètre `io_uring_group` peut lui, prendre deux valeurs :

  - `-1` : seuls les processus avec la capacité `CAP_SYS_ADMIN` peuvent utiliser une instance *io_uring* (défaut sur les distributions citées plus haut) ;
  - `1` : seuls les processus avec la capacité `CAP_SYS_ADMIN` ou membre du groupe `io_uring_group` peuvent utiliser une instance *io_uring*.

  PostgreSQL connaît de nouveaux évènements d'attente liés à l'utilisation de *io_uring*. Ils sont visibles dans la table `pg_wait_events`.

  - `AioIoUringExecution` : en attente d'AIO exécutée via *io_uring* ;
  - `AioIoUringSubmit` : en attente de soumission d'AIO via *io_uring* ;
  - `AioUringCompletion` : en attente de la finalisation de l'AIO par un autre processus via *io_uring*.

  Le tableau suivant montre un exemple des évènements d'attente que l'on peut voir dans `pg_stat_activity` :

```
         backend_type          | application_name |     wait_event
-------------------------------+------------------+---------------------
 client backend                | pgbench          | AioIoUringExecution
 client backend                | pgbench          | AioIoUringSubmit
 client backend                | psql             |
 parallel worker               | pgbench          | AioIoUringExecution
 parallel worker               | pgbench          | AioIoUringExecution
 parallel worker               | pgbench          | AioUringCompletion
 parallel worker               | pgbench          |
```

- **`io_method=sync`**

  Ce dernier mode permet d'utiliser l'API d'IO asynchrones avec des accès synchrones. Le code utilisé pour ce mode est aussi utilisé lorsque le mode `worker` doit réaliser des IO qui ne peuvent pas être réalisées avec un *io worker*.

La version 17 de PostgreSQL avait introduit le paramètre `io_combine_limit` qui permet de définir la taille maximale d'une opération d'IO. Cela permet de regrouper plusieurs lectures de blocs dans une IO, et peut réduire considérablement le nombre d'appels systèmes. La valeur par défaut est de 128 ko, la modification de ce paramètre peut être prise en compte en rechargeant la configuration, permettant de tester assez facilement différentes valeurs. En version 18, la valeur maximale de ce paramètre est passée de 256 ko à 1 Mo. Cette augmentation aligne PostgreSQL avec ce que font la plupart des autres RDBMS.

La version 18 introduit un autre paramètre `io_max_combine_limit`, qui sert de limite maximale à `io_combine_limit`. Sa valeur par défaut est également de 128 ko, à monter au besoin à 1 Mo, mais sa modification nécessite un redémarrage de l'instance.

io_max_concurrency :

Un dernier paramètre a été ajouté : `io_max_concurrency`. Il permet de définir la limite maximale du nombre d'opérations d'IO qu'un processus peut exécuter simultanément. Cette valeur a une influence sur la quantité de mémoire partagée à allouer au démarrage de PostgreSQL. La modification de ce paramètre nécessite donc un redémarrage de l'instance.

Il existe depuis longtemps `effective_io_concurrency`, dont la définition est ressemblante. Il peut, lui, être modifié par session ou tablespace et contrôle le maximum d'IO réalisable pour une opération (par exemple: un *bitmap scan*).

La valeur par défaut de `io_max_concurrency` est de `-1`. Cette valeur signifie que PostgreSQL va adapter le paramétrage en fonction de la valeur de `shared_buffers` et du nombre maximal de processus que l'instance peut démarrer (`max_connections`, `autovacuum_worker_slots`, `max_worker_processes` and `max_wal_senders`) avec un maximum de 64. Une fois modifié, ce paramètre nécessite le redémarrage de l'instance pour être pris en compte.

La formule de calcul de `io_max_concurrency` utilisée par PostgreSQL est :

$$io\_max\_concurrency = \frac{(shared\_buffers\ en\ blocs)}{(nombre\ de\ connexions\ max + processus\ auxiliaire)}$$

Avec la configuration par défaut, le nombre de backends maximum est `121` :

- `max_connections` : 100 ;
- `autovacuum_worker_slots` : 3 ;
- `max_worker_processes` : 8 ;
- `max_wal_senders` : 10.

Voici une idée des valeurs obtenues avec différentes valeurs de `shared_buffers` et nombre de processus :

processes	shared_buffers (Mo)	io_max_concurrency
121	128	64
121	256	64
121	512	64
121	1024	64
200	128	64
200	256	64
200	512	64
200	1024	64
500	**128**	**32**
500	256	64
500	512	64
500	1024	64
1000	**128**	**16**
1000	**256**	**32**
1000	512	64
1000	1024	64

La plupart du temps, `io_max_concurrency` sera donc de 64.

Quelques chiffres

Tomas Vondra a effectué de nombreux tests sur AIO, qu'il a envoyés sur la liste de discussion pgsql-hackers début juillet 2025[7] puis fin juillet[8]. Il y compare les performances en faisant varier :

- la méthode d'accès aux données :
  - PostgreSQL 17 ;
  - PostgreSQL 18 et `io_method=worker` ;
  - PostgreSQL 18 et `io_method=io_uring` ;
  - PostgreSQL 18 et `io_method=sync` ;
- `effective_io_concurrency` : 0, 1, 16, 32 ;
- `shared_buffers` : 4 Go et 8 Go ;
- avec un cache chaud et froid.

Il y compare les performances avec des *scans séquentiels*, *scans d'index* et *bitmap scans*.

Il y a peu de différences lorsque le cache est chaud. C'est compréhensible puisque l'objectif est de comparer les IO.

Lorsque le cache est froid, il observe peu de différences entre les méthodes pour les *index scans*, cela s'explique par le fait que ces accès n'utilisent pas encore l'API de streaming des IO.

Les *bitmap scans* montrent de gros gains de performances en faveur de la version 18, et ce même avec la méthode `sync` (en partie grâce à d'autres optimisations).

Les *scans séquentiels* sont plus performants pour la méthode `worker` devant `sync`, la version 17 et `io_uring`.

Globalement, dans ces tests, la méthode `worker` était la plus performante et augmenter le nombre de workers était bénéfique. Le défaut de 3 lui semble un peu bas (mais la configuration par défaut de PostgreSQL est conservatrice).

Exemple avec Bitmap Scan :

Voici un exemple avec une table ayant une répartition uniforme des données tirée des tests de Tomas Vondra. Le test est fait dans une VM avec 8 CPU, 4 Go de RAM et un disque NVMe. La configuration de PostgreSQL est celle par défaut à l'exception de `shared_buffers=2GB`. La configuration de l'OS (Rocky 9, kernel 5.14) est celle par défaut à l'exception de `kernel.io_uring_disabled=0`.

Création de la table :

```
CREATE TABLE uniform (a double precision, b text) WITH (fillfactor=25);

WITH x AS (SELECT 100000 * random() AS x FROM generate_series(1,10000000) s(i))
```

[7] https://www.postgresql.org/message-id/e6db33f3-50de-43d3-9d9f-747c3b376e80%40vondra.me
[8] https://www.postgresql.org/message-id/21674729-87b9-4bdc-9f36-13f8072b550d%40vondra.me

```
INSERT INTO uniform SELECT x, sha256(x::text::bytea) FROM x;

CREATE INDEX ON uniform (a);
```

On obtient :

```
postgres=# \dt+
                   List of tables
 Schema |  Name   | Type  |…|  Size   | Description
--------+---------+-------+…+---------+-------------
 public | uniform | table |…| 4341 MB |
(1 row)

postgres=# \di+ uniform*
                          List of indexes
 Schema |     Name      | Type  |  Owner   |  Table  |…|  Size  | Description
--------+---------------+-------+----------+---------+…+--------+-------------
 public | uniform_a_idx | index | postgres | uniform |…| 214 MB |
(1 row)
```

Le script de test est le suivant :

```
export VERSION=18
for run in {1..5}; do
  sudo systemctl restart postgresql-$VERSION;
  psql << __eof__ | grep "Time:"
  SET max_parallel_workers_per_gather = 0;
  \timing on
  SELECT * FROM uniform WHERE a BETWEEN 1000 AND 5000 ;
__eof__
done
```

Le plan de la requête est basé sur un *bitmap scan*, d'après les tests cités plus tôt, c'est le cas le plus favorable pour voir une amélioration de performances :

```
                       QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on uniform
   Recheck Cond:
     ((a >= '1000'::double precision) AND (a <= '5000'::double precision))
   ->  Bitmap Index Scan on uniform_a_idx
         Index Cond:
            ((a >= '1000'::double precision) AND (a <= '5000'::double precision))
(4 rows)
```

Voici les résultats obtenus :

pg18 + sync	pg18 + io_uring	pg18 + 3 wrk	pg18 + 5 wrk	pg18 + 7 wrkr	pg17
4438.244 ms	3419.866 ms	3499.033 ms	2671.436 ms	2648.514 ms	9479.477 ms
4472.246 ms	3417.822 ms	3557.140 ms	2812.695 ms	2737.403 ms	11912.639 ms
4851.445 ms	3568.149 ms	3559.098 ms	2812.675 ms	2587.516 ms	11985.930 ms
4503.540 ms	3444.836 ms	3547.592 ms	2799.502 ms	2538.387 ms	11861.980 ms
4774.222 ms	3595.460 ms	3550.044 ms	2804.983 ms	2650.884 ms	11950.929 ms

On voit que, pour cette requête, PostgreSQL 18 est plus rapide que PostgreSQL 17. Parmi les méthodes d'accès testées, `worker` est la plus performante avec une amélioration sensible des performances quand le nombre de *workers* augmente. `io_uring` a des performances similaires au paramétrage avec 3 *io workers*. `sync` est la méthode la moins performante pour la version 18, ce qui est normal.

Ces tests montrent que le gain en performances peut être très important avec le nouveau système AIO. Il faudra probablement effectuer des tests applicatifs afin de choisir le bon paramétrage.

Lors des tests avec `io_uring`, nous avons rencontré l'erreur suivante. Il s'agit d'un problème au niveau du noyau qui peut être rencontré sur différentes distributions :

```
ERROR:  could not read blocks 546932..546932 in file "base/5/16447": Operation
↪  canceled
```

C'est un argument supplémentaire pour conserver le paramétrage par défaut de `io_method` : `worker`.

Test avec CREATE INDEX :

La première partie de la création d'un index est le parcours complet de la table. Cette partie peut être nettement accélérée avec les *async IO*.

Par exemple, sur une machine à 16 processeurs, un disque NVMe, le parcours pour indexer le champ `bid` d'une table `pgbench_accounts` de 250 Go s'effectue à :

- 1,6 Go/s avec la configuration par défaut ;
- 2,0 Go/s en passant `max_parallel_maintenance_workers` de 4 à 9 ;
- 2,2 Go/s en montant aussi `io_workers` de 3 à 6 ;
- 2,9 Go/s en montant `io_workers` à 24.

Monter `io_workers` est donc ici très intéressant, mais il faut aussi que `max_parallel_maintenance_workers` soit assez élevé pour en profiter, et il faudra aussi monter très haut `maintenance_work_mem` pour éviter un fichier temporaire, si possible. Dans la création de l'index, les phases finales, le chargement

des fichiers de tri et l'insertion des données triées dans l'index, restent les goulots d'étranglement. Le temps d'indexation n'a ici baissé que de 12 à 10,5 minutes.

Avec `io_method = io_uring`, le temps minimal peut être similaire.

Avec `io_method = sync`, reproduisant le comportement de PostgreSQL 17, on ne dépasse pas 2,2 Go/s sur cette machine.

Là encore, conserver `io_method = worker` semble le plus sûr, et le tuning de `io_workers` nécessitera des tests.

Cas d'usage non adaptés :

Il n'y a pas d'effet notable de la nouvelle infrastructure pour les `Index Scan` : cela concerne beaucoup de petites requêtes OLTP classiques avec peu de lignes (par exemple, `pgbench`).

Pour les sauvegardes logiques (`pg_dump`), la configuration par défaut reste recommandable, mais ne représente qu'un intérêt mineur par rapport aux autres options.

5.3 ASYNCIO - NOUVELLES VUES ET WAIT EVENTS

- Nouvelle vue `pg_aios` :
  - IO en cours, taille, *handle*
- Nouveaux *wait events*
  - ex : `AioIoCompletion`

Vues pg_stat_io, pg_aios et fonctionnement interne

Les statistiques d'IO dans `pg_stat_io` sont mises à zéro pour les *io workers*. En effet, mesurer un temps sur des *IO asynchrones* perd de son sens dans la mesure où il est difficile de différencier le temps de l'IO des temps d'attentes connexes.

La nouvelle vue `pg_aios` permet de voir les *IO handles* en cours d'utilisation ou, pour simplifier les IO en cours :

```
-[ RECORD 1 ]---+-------------------------------------
pid             | 1296095
io_id           | 192
io_generation   | 34929
state           | SUBMITTED
operation       | readv
off             | 224600064
length          | 8192
target          | smgr
handle_data_len | 1
raw_result      |
result          | UNKNOWN
target_desc     | block 27417 in file "base/16388/16404"
f_sync          | t
f_localmem      | f
f_buffered      | t
```

Pour réaliser une IO, un backend (ici `pg_aios.pid`) doit au préalable obtenir un *AIO handle* (identifié par `pg_aios.io_id`).

La colonne `pg_aios.state` permet de voir l'état du *handle*, différents états sont possible :

- `HANDED_OUT` : référencé dans le code mais pas utilisé ;
- `DEFINED` : toutes les infos nécessaires à l'exécution sont connues ;
- `STAGED` : prêt pour l'exécution ;
- `SUBMITTED` : soumis au noyau ;
- `COMPLETED_IO` : terminé, mais le résultat n'est pas encore traité ;
- `COMPLETED_SHARED` : traitement « partagé » de la terminaison réalisée ;
- `COMPLETED_LOCAL` : traitement par le backend de la terminaison réalisée.

Le statut `DEFINED` fait référence au fait que le *handle* a été associé à une opération (`pg_aios.operation`). Actuellement, seules les opérations `readv` et `writev` sont supportées et correspondent à des IO vectorisées mentionnées au début de cet article. En réalité, même si les écritures sont supportées dans une partie du code, seules les lectures sont utilisées. Les colonnes `pg_aios.off` et `pg_aios.length` indiquent l'offset et la longueur de l'opération.

Ces *handles* sont stockés en mémoire partagée et peuvent être réutilisés dès que l'io est terminée. Les zones mémoires allouées pour AIO sont visibles dans la vue `pg_shmem_allocations`. On peut voir que les zones mémoires créées varient en fonction de l'`io_method` configurée, mais dans tous les cas, des zones sont dédiées à la gestion des *handles* :

Nom de la zone mémoire	Valeur de `io_method`
AioWorkerSubmissionQueue	`worker`
AioWorkerControl	`worker`
AioUringContext	`io_uring`
AioHandleData	`worker`, `io_uring` et `sync`
AioHandleIOV	`worker`, `io_uring` et `sync`
AioHandle	`worker`, `io_uring` et `sync`
AioBackend	`worker`, `io_uring` et `sync`
AioCtl	`worker`, `io_uring` et `sync`

Chaque *handle* est associé à une cible (*AIO targets*) mais plusieurs *handles* peuvent avoir la même cible (le même bloc d'un fichier). Cette information est visible dans la colonne `pg_aios.target`. `smgr` désigne des IO sur des relations, c'est la seule valeur disponible pour le moment. Dans ce cas, la colonne `pg_aios.target_desc` identifie le *fork* de la relation et le bloc qui est lu.

Quand l'IO est réalisée depuis `shared_buffers`/`temp_buffers` (le seul choix actuellement), le nombre de blocs concernés est visible dans `pg_aios.handle_data_len`.

Les *handles* sont acquis et transmis entre différentes portions du code comme le *buffer manager* et le stockage. Elles n'ont pas les mêmes niveaux d'abstraction (exemples : l'un manipule des blocs, l'autre des descripteurs de fichiers et des offsets, certains ont besoin de données en mémoire partagée, d'autres de données dans la mémoire du backend), mais doivent tout de même pouvoir réagir au résultat d'une AIO (appelé : *AIO completion*). Pour cela, chaque *handle* peut être associé à plusieurs *callbacks*. Cela permet aux différents niveaux d'être notifiés de la fin d'une AIO, et ainsi, d'y réagir d'une manière appropriée. Ces callbacks ne sont pas visibles dans la vue. Au cours de leur exécution, `pg_aios.state` va passer de `COMPLETED_IO` à `COMPLETED_SHARED` puis `COMPLETED_LOCAL`.

Comme les *AIO handles* peuvent être réutilisés directement après qu'une IO a terminé, et qu'un backend peut être en train de faire autre chose le temps que l'AIO est réalisée, les backends ne peuvent pas utiliser les *AIO handles* directement pour attendre le résultat d'une IO. Il a donc fallu mettre en place un moyen pour distinguer des AIO différentes utilisant le même *handle* : un numéro de génération, et un moyen d'attendre la fin d'une IO : les *AIO wait references*.

Le numéro de génération est visible dans `pg_aios.io_generation` et indique le nombre de fois que le *handle* a été réutilisé. La vue `pg_stat_aio` ne montre que les *handles* en cours d'utilisation.

Avant la réutilisation d'un *AIO handle*, une structure décrivant le résultat de l'opération est copiée dans la mémoire locale du backend qui a demandé l'IO. L'acquittement du traitement de l'IO et la prise en compte du retour en lui même sont donc découplés.

5.4 CRÉATION PARALLÉLISÉE DES INDEX GIN

- Parallélisation de la construction des index GIN
  - enfin !
- `max_parallel_maintenance_workers` = 2 (défaut)
  - on peut augmenter un peu, gain non linéaire.

Après les index B-tree depuis PostgreSQL 11, les index BRIN depuis PostgreSQL 17, PostgreSQL 18 sait paralléliser la construction d'index GIN.

Comme pour les index B-tree, la mémoire utilisable `maintenance_work_mem` est partagée entre les workers (sinon il y a tri sur disque), et le nombre de processus annexes est géré par `max_parallel_maintenance_workers` (défaut : 2).

Augmenter le paramètre `max_parallel_maintenance_workers` implique souvent de monter `max_parallel_workers` (défaut 8), voire `max_worker_processes` (défaut 8), et, bien sûr, d'avoir une machine avec au moins autant de processus. `maintenance_work_mem` doit suffire à attribuer 32 Mo à chaque worker, ce qui n'est généralement pas un souci sur les instances correctement paramétrées.

Exemple : index GIN d'un JSON :

La base **personnes** pèse en version complète 613 Mo, pour 2 Go sur disque au final. Elle peut être installée comme suit :

```
curl -kL https://dali.bo/tp_personnes -o /tmp/personnes.dump
createdb --echo personnes
pg_restore -v -d personnes /tmp/personnes.dump
rm -- /tmp/personnes.dump
```

La construction de l'index de 284 Mo sur le champ JSON se fait ainsi :

```
SET maintenance_work_mem TO '1GB';
SET max_parallel_maintenance_workers TO 2 ;
DROP INDEX pers_json_idx ;
CREATE INDEX pers_json_idx ON json.personnes USING gin (personne jsonb_path_ops);
```

max_parallel_maintenance_workers	Temps de calcul
0 (pas de parallélisation)	27,3 s
2 (défaut)	17,2 s
4	15,1 s
6	15,1 s

Le gain est très appréciable, mais il n'est pas linéaire avec le nombre de workers, car toutes les étapes de création de l'index ne sont pas parallélisables. Monter `max_parallel_maintenance_workers` très haut n'a d'intérêt que pour les plus grosses tables.

Exemple : index GIN pour pg_trgm :

Dans certains cas, on hésite entre des index GIN et GiST (par exemple pour utiliser pg_trgm[9]). Les index GiST sont censés être plus légers à maintenir, mais la parallélisation fait pencher la balance un peu plus vers le GIN.

Cet exemple utilise la base **gutenberg** contenant de nombreux livres (21 millions de lignes pour 3 Go sur le disque) :

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
# version réduite :
# curl -kL https://dali.bo/tp_gutenberg10 -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
rm -- /tmp/gutenberg.dmp
```

La construction de l'index GIN (taille : 1,3 Go) pour l'indexation `pg_trgm` s'effectue ainsi :

```
SET max_parallel_maintenance_workers TO 2 ;
DROP INDEX idx_textes_trgm ;
CREATE INDEX idx_textes_trgm ON textes USING gin (contenu gin_trgm_ops);
\di+ idx_textes_trgm
```

max_parallel_maintenance_workers	Temps de calcul
0 (pas de parallélisation)	6 min 45 s
2 (défaut)	3 min 04 s
4	2 min 29 s

[9] https://dali.bo/x2_html#pg_trgm

max_parallel_maintenance_workers	Temps de calcul
6	2 min 06 s
8	2 min 05 s

On peut donc diviser le temps de calcul par 2 ou 3, mais ensuite un plafond est relativement vite atteint.

Pour comparaison, la création d'un index GiST, non parallélisable, prend 11 minutes !

5.5 AMÉLIORATION DU MÉCANISME FAST-PATH LOCKING

- Le mécanisme *fast-path locking* est amélioré
- Plus de verrous peuvent être posés via ce mécanisme
- Configurable via `max_locks_per_transaction` (64 par défaut)
- Limite dure à 16 384 verrous (contre 16 auparavant)
- Profite aux *workloads* de type OLTP avec tables partitionnées

Il est sans doute nécessaire, dans un premier temps, de décrire ce mécanisme relativement peu connu. Dans ce chapitre, sauf mention explicite, nous parlerons uniquement des verrous au niveau « table » décrits dans cette partie de la documentation[10]. Ceux-ci concernent les tables ainsi que leurs index éventuels.

Le mécanisme *fast-path locking* a été ajouté pour la version 9.2 de PostgreSQL, par Robert Haas, et n'a quasiment pas bougé jusqu'en version 17 incluse. Il permet d'éviter des contentions liées à l'accès d'une table de hachage en mémoire partagée, qui contient tous les verrous de tous les *backends* jusqu'en version 9.1. Celle-ci sera nommée « table des verrous principale » dans la suite de ce chapitre. À partir de la version 9.2, certains verrous peuvent être enregistrés dans la structure PGPROC de chaque *backend* (qui réside en mémoire partagée et permet de régir les interactions inter-processus en partageant des informations de verrouillage, synchronisation, etc.) plutôt que dans la table des verrous principale. Les verrous éligibles sont les verrous « faibles » qui n'entrent pas en conflit avec des verrous du même type (voir ce tableau[11]) : `ACCESS SHARE`, `ROW SHARE`, et `ROW EXCLUSIVE`.

Il est aussi possible de stocker un unique verrou de type `vxid` via ce mécanisme, ce qui est intéressant dans la mesure où les transactions prennent toujours un verrou sur leur propre *virtual transaction ID*, permettant ainsi à certaines actions d'attendre la fin de celles-ci ; `CREATE INDEX CONCURRENTLY` utilise notamment cette technique pour attendre la fin des transactions qui modifient la table, ou celles ayant un snapshot plus ancien, selon les différentes étapes de la création.

[10] https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-TABLES
[11] https://www.postgresql.org/docs/current/explicit-locking.html#TABLE-LOCK-COMPATIBILITY

Dans l'exemple ci-dessous, le processus 485913 exécute la requête
`CREATE INDEX CONCURRENTLY ON pgbench_accounts(bid);`
mais est bloqué par le processus 485921, qui a modifié la même table avec la requête
`INSERT INTO pgbench_accounts(aid) VALUES (-1);`
Sur l'avant-dernière ligne, on voit que le premier a tenté de prendre un verrou de type `virtualxid` en mode `ShareLock`, sans utiliser le mécanisme *fast-path* (champ `fastpath`), sur la *virtual transaction* 23/11 portée par le deuxième. Il sera intéressant de revenir à cet exemple après avoir lu la suite de ce chapitre.

```
bench [488266] # SELECT locktype, relation, virtualxid as vxid, virtualtransaction as vtrans, l.pid,
  mode, granted, fastpath, substring(query from 1 for 12) as "truncated query"
  FROM pg_locks l left join pg_stat_activity s on (l.pid = s.pid and s.query NOT LIKE '%pg_locks%')
  WHERE l.pid != pg_backend_pid();

   locktype    | relation | vxid  | vtrans |  pid   |           mode            | granted | fastpath | truncated query
---------------+----------+-------+--------+--------+---------------------------+---------+----------+-----------------
 virtualxid    |          | 22/41 | 22/41  | 485913 | ExclusiveLock             | t       | t        | CREATE INDEX
 relation      |    16445 |       | 23/11  | 485921 | RowExclusiveLock          | t       | t        | INSERT INTO
 relation      |    16445 |       | 22/41  | 485913 | ShareUpdateExclusiveLock | t       | f        | CREATE INDEX
 transactionid |          |       | 23/11  | 485921 | ExclusiveLock             | t       | f        | INSERT INTO
 virtualxid    |          | 23/11 | 22/41  | 485913 | ShareLock                 | f       | f        | CREATE INDEX
 virtualxid    |          | 23/11 | 23/11  | 485921 | ExclusiveLock             | t       | f        | INSERT INTO
```

Même chose ci-dessous, mais cette fois-ci le processus 485921 porte une transaction *Repeatable Read* qui a juste fait un simple `select 1`, et qui bloque à nouveau la création d'index dans sa phase finale.

```
  locktype  | relation | vxid  | vtrans |  pid   |           mode            | granted | fastpath | truncated query
------------+----------+-------+--------+--------+---------------------------+---------+----------+-----------------
 virtualxid |          | 22/43 | 22/43  | 485913 | ExclusiveLock             | t       | t        | CREATE INDEX
 virtualxid |          | 23/12 | 23/12  | 485921 | ExclusiveLock             | t       | f        | select 1;
 virtualxid |          | 23/12 | 22/43  | 485913 | ShareLock                 | f       | f        | CREATE INDEX
 relation   |    16445 |       | 22/43  | 485913 | ShareUpdateExclusiveLock | t       | f        | CREATE INDEX
```

Le schéma ci-dessous représente les principales structures de données impliquées dans ce mécanisme.

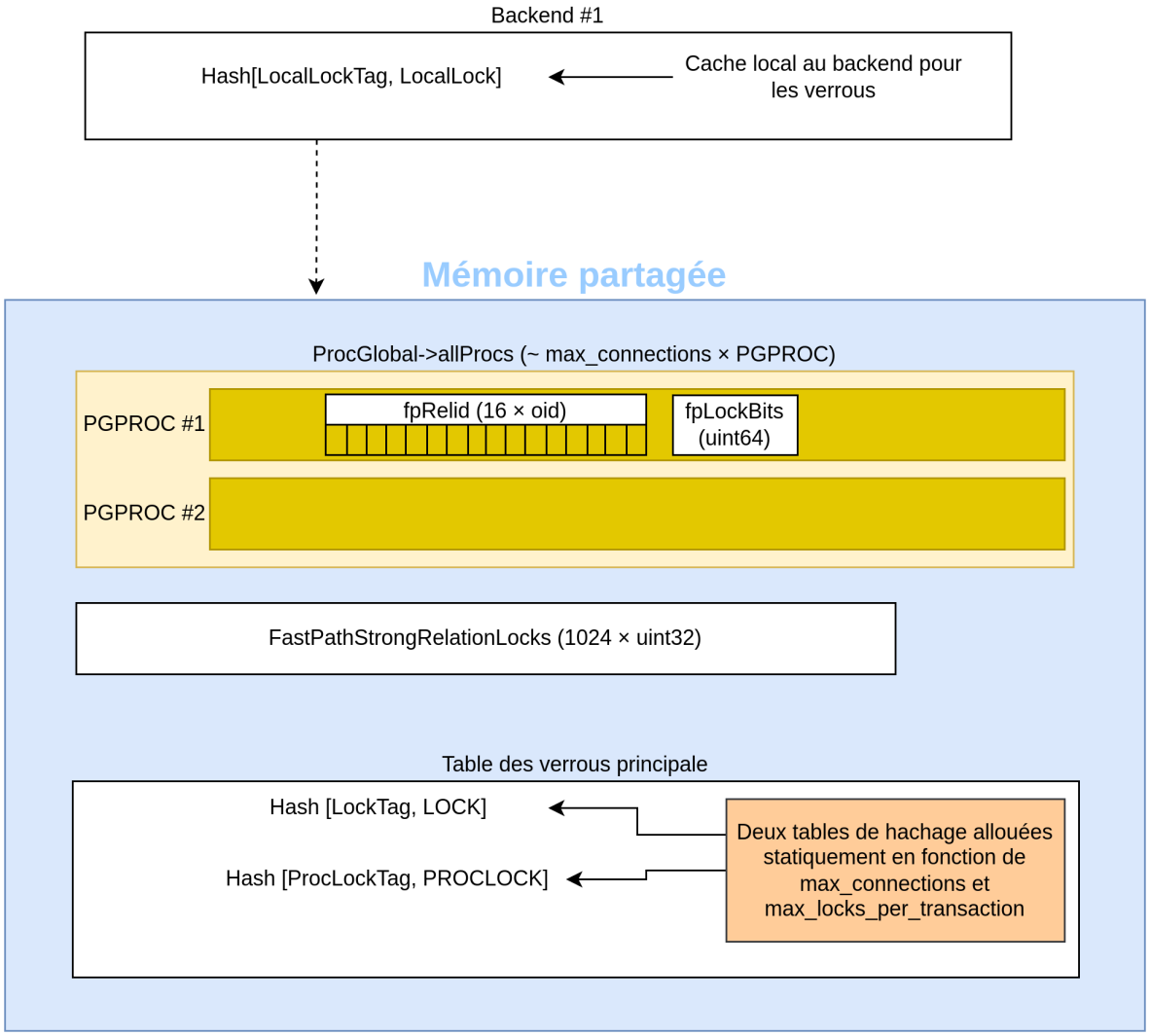

Figure 5/ .1: Principales structures de données

On peut se demander à juste titre quel est l'intérêt de stocker un verrou dans une structure de données propre à un *backend*, étant donné que celui-ci est utile pour la synchronisation entre les différents *backends*. Mais cette structure de données, comme indiquée plus haut, vit tout de même en mémoire partagée, et est donc accessible aux autres *backends*. Ainsi, lorsqu'un autre *backend* veut poser un verrou « fort » (SHARE ou au-dessus), il va pouvoir aller vérifier le PGPROC de tous les *backends* existants, pour voir s'ils contiennent un verrou faible sur la même *relation* (index ou table). Si c'est le cas, alors le verrou faible est migré dans la table des verrous principale. Dans tous

les cas, un compteur est incrémenté en mémoire partagée pour indiquer qu'un verrou fort est positionné pour la relation en question ; il existe un tableau statique de 1024 entiers en mémoire partagée (`FastPathStrongRelationLocks` dans le schéma), pour tous les verrous forts de toutes les relations de toutes les bases de l'instance, chaque compteur étant donc utilisé pour possiblement plus d'un verrou. Quand ce compteur est strictement supérieur à zéro, le mécanisme *fast-path* n'est pas disponible pour tous les couples `(base oid, relation oid)` dont le hachage correspond à son index dans le tableau. Ce dernier est protégé par un *spin lock*, uniquement en écriture, et ne pose donc pas de problèmes de contention dans le cas nominal du mécanisme *fast-path*. Pour résumer, le mécanisme *fast-path* est vraiment peu coûteux lorsqu'on a juste besoin de prendre un verrou « faible », mais ajoute un certain *overhead* lorsqu'il est nécessaire de prendre un verrou « fort ». Typiquement, il y a toujours beaucoup plus de verrous faibles que forts, et on y gagne donc beaucoup en moyenne.

Jusqu'en version 17, chaque *backend* peut stocker un maximum de 16 verrous via le mécanisme *fast-path*, mais cette limite est atteinte assez vite, il suffit d'une table avec 10 partitions et un index (un par partition), et d'une requête qui ne puisse pas faire de *partition pruning*. En pratique, on peut observer un effet de contention avec seulement 30 clients qui font des requêtes en lecture sur une telle table partitionnée.

La version 18 autorise un bien plus grand nombre de verrous, configurable avec le paramètre `max_locks_per_transaction` qui vaut 64 par défaut. On réutilise donc ici un paramètre déjà existant, afin de ne pas en rajouter encore un à la longue liste des paramètres PostgreSQL. Il est possible que cela change par la suite, car la taille de la table des verrous principale dépend du produit `max_locks_per_transaction × max_connections`, ce qui peut commencer à être non négligeable si on doit augmenter sensiblement `max_locks_per_transaction` et que `max_connections` a une valeur relativement haute. Afin de travailler au mieux avec le cache du CPU (principe de localité), on utilise toujours des tableaux de 16 identifiants de *relation* (32 bits chacun, 64 octets en tout, la taille d'un *CPU cache line*) répartis en plusieurs groupes. Avec `max_locks_per_transaction` positionné à 48, on a donc trois groupes comme sur le schéma ci-dessous. Il suffit de calculer un hachage de l'identifiant de *relation* pour déterminer le groupe à utiliser, et de procéder ensuite à une recherche linéaire dans le tableau, ce qui est efficace pour un petit tableau de 16 éléments. Le nombre maximal de groupes autorisés dans le code est de 1024, soit 16 384 verrous au total, pour chaque backend.

PGPROC étant une structure, sa taille est fixée à la compilation. Le tableau contenant les groupes d'oid est lui dimensionné en fonction de `max_locks_per_transaction`, qui est défini au démarrage de PostgreSQL. Il ne peut donc pas être stocké dans PGPROC. Cela explique pourquoi dans le schéma ci-dessous, le tableau est une entité séparée de PGPROC.

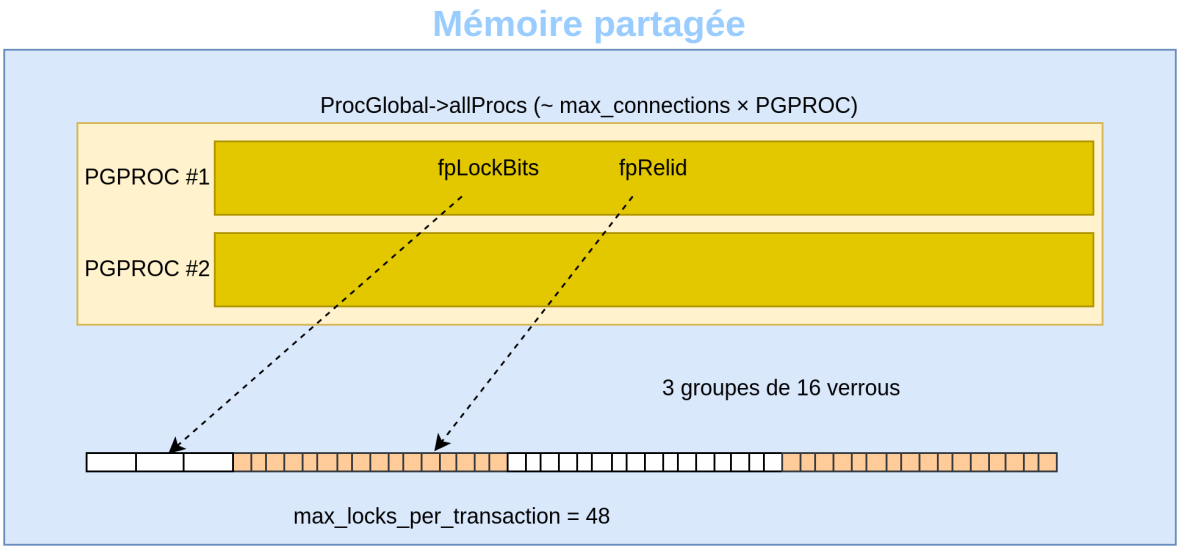

Figure 5/ .2: Changements en v18

Notons enfin que ce patch a fait sauter un goulet d'étranglement, mais son auteur, Tomas Vondra, en a identifié un autre[12] juste derrière, lié à l'allocation mémoire. Il a fallu le résoudre également afin de pouvoir vraiment bénéficier de plus de performances pour les cas d'usage concernés.

[12] https://vondra.me/posts/tuning-the-glibc-allocator-for-postgres/

5.6 AMÉLIORATIONS SUR LA COMMANDE EXPLAIN

- Affichage amélioré des nœuds désactivés
- Option BUFFERS
  - inclus par défaut
- Option WAL
  - ajout de l'information `buffers full`
- Ajout du nombre de recherches dans l'index
  - nouvelle ligne `Index searches`
- Affichage du nombre de lignes en fractionnel
- Ajout des informations d'utilisation mémoire et disque
- Ajout d'informations sur les arguments des fonctions de fenêtrage
- Ajout de statistiques sur le cache du worker parallélisé d'un `Bitmap Heap Scan`

Jeu de tests pour les évolutions autour de la commande EXPLAIN :

```
DROP TABLE IF EXISTS t1,t2;
CREATE TABLE t1 (c1 integer);
INSERT INTO t1 SELECT generate_series(1, 1000);
CREATE TABLE t2 (c1 integer);
INSERT INTO t2 SELECT random(1,g) FROM generate_series(1, 1000) g;
CREATE INDEX ON t2 (c1);
VACUUM ANALYZE;
```

Affichage amélioré des nœuds désactivés :

Lors de la désactivation d'un nœud avec l'un des paramètres `enable_*`,les coûts ne sont plus augmentés arbitrairement de 10 milliards comme auparavant. Cependant, si le nœud désactivé reste la seule solution possible, le nœud apparaît dans le plan avec une ligne `Disabled`.

Par exemple :

```
SET enable_seqscan TO off;
EXPLAIN SELECT * FROM t1;

                        QUERY PLAN
```

```
------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..30467.92 rows=2112192 width=4)
   Disabled: true
(2 rows)
```

Option BUFFERS :

Les informations sur les blocs lus ou écrits sont maintenant affichées par défaut avec `EXPLAIN (ANALYZE)` :

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM t1;
```

En voici le résultat :

```
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on t1 (actual rows=1000.00 loops=1)
   Buffers: shared hit=5
 Planning Time: 0.046 ms
 Execution Time: 0.162 ms
(4 rows)
```

L'effet de cache sera donc beaucoup plus évident lors de la comparaisons de plans à priori identiques mais de durées différentes. Il est toujours possible de désactiver l'option `BUFFERS` grâce à la valeur `OFF` :

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)
SELECT * FROM t1;
```

Option WAL :

Cette option existe depuis la version 13. En version 18, une information supplémentaire est disponible : le nombre de fois où le cache disque des journaux de transactions était rempli (et qu'il a donc fallu le vider sur disque).

Par exemple :

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, WAL)
INSERT INTO t1 SELECT generate_series(1, 1000000);
```

En voici le résultat :

```
                          QUERY PLAN
-------------------------------------------------------------
 Insert on t1 (actual rows=0.00 loops=1)
   Buffers: shared hit=1008860 dirtied=4430 written=4426
   WAL: records=1000003 fpi=4 bytes=59028234 buffers full=7007
   ->  ProjectSet (actual rows=1000000.00 loops=1)
         ->  Result (actual rows=1.00 loops=1)
 Planning Time: 0.123 ms
 Execution Time: 1150.033 ms
(7 rows)
```

La ligne WAL indique 1 million d'enregistrements, 4 *full page writes*, environ 56 Mio de journaux, et 7007 événements *buffer full*. Le nombre de ces derniers est lié au paramètre `wal_buffers` et peut permettre de détecter qu'il est trop petit, ce qui peut ralentir les écritures lourdes.

Ajout du nombre de recherches dans l'index :

Il était difficile auparavant de savoir si l'index était parcouru une seule fois ou plusieurs fois. À partir de la version 18, la ligne `Index Searches` indique le nombre de parcours dans l'index. Par exemple :

```
SET jit TO off ; SET enable_hashjoin TO off;

EXPLAIN (ANALYZE)
SELECT * FROM t1 JOIN t2 ON t1.c1<t2.c1;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.28..8370505.00 rows=295333333 width=8) (actual
↪  time=0.067..518.923 rows=506748.00 loops=1)
   Buffers: shared hit=2008367
   ->  Seq Scan on t1  (cost=0.00..13290.00 rows=886000 width=4) (actual
↪  time=0.018..34.857 rows=1001000.00 loops=1)
         Buffers: shared hit=4430
   ->  Index Only Scan using t2_c1_idx on t2  (cost=0.28..6.10 rows=333 width=4)
↪  (actual time=0.000..0.000 rows=0.51 loops=1001000)
         Index Cond: (c1 > t1.c1)
         Heap Fetches: 0
         Index Searches: 1001000
         Buffers: shared hit=2003937
 Planning Time: 0.185 ms
 Execution Time: 532.369 ms
```

Il devient donc beaucoup plus visible que PostgreSQL a fait plus d'un million d'accès à l'index, et non un seul. La mention `loops=1001000` était déjà visible dans les versions précédentes pour pointer

ce phénomène. À l'inverse, ce plan n'utilise que deux appels d'index pour récupérer 5 lignes d'une part, et 48 d'autre part :

```
RESET ALL ;
EXPLAIN (ANALYZE,BUFFERS OFF, COSTS OFF,SETTINGS)
SELECT * FROM t2 WHERE c1 > 900 OR c1 <10 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t2 (actual time=0.032..0.045 rows=53.00 loops=1)
   Recheck Cond: ((c1 > 900) OR (c1 < 10))
   Heap Blocks: exact=5
   ->  BitmapOr (actual time=0.012..0.013 rows=0.00 loops=1)
         ->  Bitmap Index Scan on t2_c1_idx (actual time=0.005..0.005 rows=5.00
 ↪  loops=1)
               Index Cond: (c1 > 900)
               Index Searches: 1
         ->  Bitmap Index Scan on t2_c1_idx (actual time=0.006..0.006 rows=48.00
 ↪  loops=1)
               Index Cond: (c1 < 10)
               Index Searches: 1
 Planning Time: 0.258 ms
 Execution Time: 0.075 ms
```

Affichage du nombre de lignes :

L'affichage du nombre de lignes a un peu changé. Il n'est plus affiché sous la forme d'un entier, mais sous la forme d'un nombre décimal. Ceci a pour but d'éviter d'arrondir à 0 quand il y a quelques lignes renvoyées et que le nombre de boucles est supérieur au nombre de lignes renvoyées.

Dans un exemple avec la jointure ci-dessus, on voit :

```
Index Only Scan using t2_c1_idx on t2  (cost=0.28..6.10 rows=333 width=4) (actual
 ↪  time=0.000..0.000 rows=0.51 loops=1001000)
```

Ce parcours d'index a été réalisé 1 million de fois et n'a pas souvent ramené une ligne (total final à 333 seulement). Avant la version 18, nous aurions vu 0 ligne par passage (`rows=0`), donc 0 ligne après les deux millions de passages. Là, nous comprenons que 0,51 × 1001000 lignes sont récupérées (donc environ 0,5 million), ce qui correspond bien au nombre de lignes indiqué au niveau du nœud `Nested Loop`.

Ajout des informations d'utilisation mémoire et disque

Sur les nœuds `CTE Scan`, `Materialize` et `Window Aggregate`, la quantité de mémoire ou d'espace disque est affichée. Par exemple :

```
EXPLAIN (ANALYZE)
SET enable_hashjoin TO  'off' ;
SET max_parallel_workers_per_gather TO '0' ;
SET jit TO 'off';
SET work_mem TO '4MB';

WITH cte1 AS MATERIALIZED
 (SELECT * FROM t1 WHERE c1 >0)
SELECT * FROM cte1
INNER JOIN t2 ON cte1.c1=t2.c1;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=150403.89..150454.92 rows=1007 width=8) (actual
↪  time=195.970..196.545 rows=2000.00 loops=1)
   Merge Cond: (t2.c1 = cte1.c1)
   Buffers: shared hit=4435, temp read=510 written=3187
   CTE cte1
     ->  Seq Scan on t1  (cost=0.00..16942.50 rows=1000900 width=4) (actual
↪  time=0.017..47.152 rows=1001000.00 loops=1)
           Filter: (c1 > 0)
           Buffers: shared hit=4430
   ->  Index Only Scan using t2_c1_idx on t2  (cost=0.28..35.27 rows=1000 width=4)
↪  (actual time=0.021..0.074 rows=1000.00 loops=1)
         Heap Fetches: 0
         Index Searches: 1
         Buffers: shared hit=5
   ->  Materialize  (cost=133457.03..138461.53 rows=1000900 width=4) (actual
↪  time=195.943..196.224 rows=2895.00 loops=1)
         Storage: Memory  Maximum Storage: 17kB
         Buffers: shared hit=4430, temp read=510 written=3187
         ->  Sort  (cost=133457.03..135959.28 rows=1000900 width=4) (actual
↪  time=195.939..196.032 rows=1901.00 loops=1)
               Sort Key: cte1.c1
               Sort Method: external merge  Disk: 11776kB
               Buffers: shared hit=4430, temp read=510 written=3187
               ->  CTE Scan on cte1  (cost=0.00..20018.00 rows=1000900 width=4)
↪  (actual time=0.020..139.627 rows=1001000.00 loops=1)
                     Storage: Disk  Maximum Storage: 13680kB
                     Buffers: shared hit=4430, temp written=1710
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.276 ms
 Execution Time: 198.743 ms
```

Ce plan montre deux clauses `Storage` : la première, tout en bas, indique que la CTE matérialisée a pris environ 13 Mo sur le disque ; la deuxième, au milieu, indique que le résultat du nœud `Sort` a été matérialisé en mémoire, dans seulement 17 ko.

6/ Optimiseur

6.1 SKIP SCAN POUR INDEX B-TREE

```
-- Index sur 2 colonnes
CREATE INDEX ON matable (c1 , c2);
-- critère sur la 2è colonne
SELECT * FROM matable WHERE c2 = …
```

- Avant v18 : lecture de tout l'index… voire *Seq Scan*
- v18 : pour chaque `c1`, cherche la bonne valeur de `c2`
- Suppose : `c1` de faible cardinalité
- Économie d'index

Un index B-tree multicolonne n'est optimal que si la première colonne fait systématiquement partie des critères de recherche. En effet, les données sont triées dans l'index d'abord selon le premier champ (ici `c1`), puis selon le second (`c2`), etc.

Si le critère de recherche ne porte que sur le second champ `c2`, l'index est peut-être en partie utilisable. PostgreSQL 17 et précédents ne peuvent accéder aux valeurs de `c2` directement, mais l'index peut être lu intégralement pour y trouver les diverses valeurs de `c2` qui y sont dispersées. C'est souvent mieux que parcourir toute la table, mais pas optimal. Plus l'index est gros par rapport à la table, plus l'optimiseur aura tendance à se rabattre sur un parcours complet de la table.

PostgreSQL 18 connaît les *index skip scans*. Il change l'accès habituel à l'index en autant d'accès qu'il y a de valeurs de `c1`, comme s'il y avait autant d'index différents (« sous-index logiques ») ; puis il va dans chacun chercher les valeurs correspondant au critère sur `c2`. Il n'y a pas de nœud *Skip Scan* dédié. Le degré de découpage est en pratique géré lors de l'exécution. S'il y a trop de valeurs de `c1` (des milliers), PostgreSQL peut revenir à un parcours intégral de l'index ou de la table. Avec les *skip scan*, l'accès à l'index n'est pas aussi optimal qu'un index commençant par `c2`, mais on peut beaucoup s'en rapprocher.

Cette optimisation n'a vraiment d'intérêt que si les valeurs distinctes de `c1` sont assez peu nombreuses (des années, des statuts…). Au contraire, si les valeurs de `c1` sont trop diverses (champ très discriminant), l'intérêt est limité, car en pratique PostgreSQL parcourra tout l'index ou presque.

Cette nouvelle fonctionnalité permet donc d'économiser des index. Des requêtes non critiques peuvent se satisfaire d'un index dont elles ne filtrent pas le premier champ. On peut éviter de créer le même index multicolonnes en deux versions dans des ordres différents pour satisfaire à des requêtes différentes : celui commençant par la colonne la moins discriminante sera peut-être suffisant pour toutes les requêtes. De même, on a couramment le dilemme suivant : créer un index pour optimiser

un ORDER BY ou en créer un autre pour optimiser le critère de sélection ? À présent, l'index pour le tri suffira peut-être à la sélection des lignes.

Cette amélioration ne concerne que les index B-tree, qui sont les plus courants. Il arrivait que des index GIN, voire bloom, soient utilisés pour cette indexation multicolonne[1], mais ils sont beaucoup plus lourds ou ont des limites.

Exemple :

Cette table contient des ID de commandes, avec un index sur `(annee, num_client)`. Elle est triée pour améliorer la corrélation physique et rendre plus évident les changements d'utilisation des blocs.

```
DROP TABLE IF EXISTS demo ;
CREATE TABLE demo (annee int NOT NULL,
                   num_client varchar (5) NOT NULL,
                   id_commande serial PRIMARY KEY,
                   mois int , z int, filler char(10) default ' ');
INSERT INTO demo (annee, num_client, mois)
SELECT a, trim(to_char(cl,'00000') ), (cl+commandes)%12
FROM generate_series (2016,2025) a
CROSS JOIN generate_series (30000,99999) cl
CROSS JOIN generate_series (10,20) commandes
ORDER BY a,cl,commandes;

CREATE INDEX demo_annee_num_client_idx ON demo (annee, num_client) ;

-- Ne pas oublier les statistiques
VACUUM (ANALYZE) demo ;
```

La table pèse 442 Mo, l'index sur les deux champs 165 Mo.

Ce qui suit suppose des bases PostgreSQL 17 et 18 avec le paramétrage par défaut, à part ceci pour simplifier les plans :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
```

À cause de l'effet de cache en cas de répétition des requêtes, les plans peuvent différer de vos propres tests.

Dans le cas idéal, avec une année et un numéro de client, récupérer une ligne via l'index ne nécessite que 4 blocs :

```
EXPLAIN (ANALYZE,BUFFERS,SETTINGS)  SELECT * FROM demo
WHERE num_client = '50000' AND annee = 2024 ;
```

[1] https://dali.bo/j5_html#indexation-multicolonne-gin-gist-bloom

```
 Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..27.92 rows=12
↪  width=33) (actual time=0.040..0.045 rows=11.00 loops=1)
   Index Cond: ((annee = 2024) AND ((num_client)::text = '50000'::text))
   Index Searches: 1
   Buffers: shared hit=4
 Planning Time: 0.124 ms
 Execution Time: 0.068 ms
```

Dans le cas où nous cherchons les lignes d'un client pour toutes les années, PostgreSQL 17 utilise l'index, mais on constate qu'il le lit intégralement (l'index pèse 9640 blocs selon `pg_class.relpages`). On ne peut malheureusement distinguer ici les blocs lus dans l'index de ceux lus dans la table.

```
EXPLAIN (ANALYZE,BUFFERS,SETTINGS)  SELECT * FROM demo
WHERE num_client = '50000' ;
```

```
 Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..96520.72 rows=118
↪  width=33) (actual time=1.709..21.606 rows=110 loops=1)
   Index Cond: ((num_client)::text = '50000'::text)
   Buffers: shared hit=9603
 Settings: search_path = '"$user", public, topology', jit = 'off',
↪  max_parallel_workers_per_gather = '0'
 Planning Time: 0.134 ms
 Execution Time: 21.641 ms
```

PostgreSQL 18 sait mieux utiliser l'index pour trouver la commande : seuls 47 blocs sont lus :

```
 Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..257.84 rows=117
↪  width=33) (actual time=0.031..0.108 rows=110.00 loops=1)
   Index Cond: ((num_client)::text = '50000'::text)
   Index Searches: 12
   Buffers: shared hit=47
 Planning Time: 0.068 ms
 Execution Time: 0.128 ms
```

La ligne `Index Searches` est une amélioration de PostgreSQL 18, qui indique le nombre de recherches indépendantes dans l'index, c'est-à-dire démarrant de la racine, et pouvant potentiellement lire de nombreuses feuilles. Pour l'efficacité, il vaut mieux qu'il y en ait le moins possible. Mais ici, 12 recherches permettent d'éviter de lire tout l'index. Il y a une recherche pour chacune des 10 années.

S'il y a deux valeurs à chercher, `Index Searches` trahit ici un accès à chaque année pour chacune :

```
EXPLAIN (ANALYZE,BUFFERS,SETTINGS,SUMMARY OFF)  SELECT * FROM demo
WHERE num_client = '30000' OR num_client = '90001' ;
```

```
 Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..513.98 rows=234
↪  width=33) (actual time=0.036..0.283 rows=220.00 loops=1)
   Index Cond: (num_client = ANY ('{30000,90001}'::text[]))
   Index Searches: 21
   Buffers: shared hit=83
```

Si les valeurs des clients sont très proches, une autre optimisation entre en jeu et le nombre de recherches et de blocs chute :

```
 Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..513.98 rows=234
↪  width=33) (actual time=0.047..0.302 rows=220.00 loops=1)
   Index Cond: (num_client = ANY ('{30000,30009}'::text[]))
   Index Searches: 12
   Buffers: shared hit=49 read=4
```

En effet, PostgreSQL repère la proximité de valeurs et préfère parcourir plus de feuilles finales dans l'index pour éviter les quelques blocs d'une recherche complète de la deuxième valeur. La recherche d'un algorithme qui évite une régression (à quel point parcourt-on trop de feuilles inutiles ?) a fait partie des travaux de la version 18[2].

Cette requête calcule l'évolution des commandes par années et mois pour un client :

```
EXPLAIN (ANALYZE,BUFFERS,SETTINGS)
SELECT annee, mois, count(*) FROM demo
WHERE num_client = '50000'
GROUP BY annee, mois
ORDER BY annee, mois ;
```

```
 GroupAggregate  (cost=26.38..263.20 rows=75 width=16) (actual time=0.097..0.254
↪  rows=110.00 loops=1)
   Group Key: annee, mois
   Buffers: shared hit=47
   ->  Incremental Sort  (cost=26.38..261.57 rows=117 width=8) (actual
↪  time=0.089..0.196 rows=110.00 loops=1)
         Sort Key: annee, mois
         Presorted Key: annee
         Full-sort Groups: 4  Sort Method: quicksort  Average Memory: 25kB  Peak
↪  Memory: 25kB
         Buffers: shared hit=47
         ->  Index Scan using demo_annee_num_client_idx on demo  (cost=0.43..257.84
↪  rows=117 width=8) (actual time=0.030..0.137 rows=110.00 loops=1)
               Index Cond: ((num_client)::text = '50000'::text)
               Index Searches: 12
               Buffers: shared hit=47
 Planning:
   Buffers: shared hit=8
```

[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=8a510275d

```
 Planning Time: 0.201 ms
 Execution Time: 0.293 ms
```

La sélection des donnée se fait par l'index `(annee, num_client)` avec 12 recherches. Comme l'index est déjà trié par années, PostgreSQL peut utiliser l'`Incremental Sort`, une optimisation de PostgreSQL 13, pour ne pas tout trier.

Un seul index à priori non optimal permet donc à la requête d'être très rapide. Avec PostgreSQL 17, la requête aurait dû parcourir tout l'index et aurait duré 20 ms (60 fois plus longtemps).

Exemple de recherche multicritère :

Une recherche multicritère est notoirement problématique pour un index B-tree quand aucun champ n'est obligatoire.

```
CREATE TABLE demo_multi  (n int, i int, j int, k int, l int,
                          filler char(50) default ' ') ;

SELECT * FROM demo_multi WHERE j=1 AND l=17 ;
SELECT * FROM demo_multi WHERE k=3 AND l=1 ;
SELECT * FROM demo_multi WHERE i=0 AND k=33 ;
```

Il est alors impossible de choisir une première colonne. Un index B-tree portant sur les différents critères est inutilisable par PostgreSQL 17 dans le cas général :

```
CREATE INDEX demo_multi_idx ON demo_multi USING btree (i,j,k,l) ;
```

L'optimiseur utilise alors souvent un *Seq Scan*. On peut bien sûr définir un index sur chaque combinaison de critères, mais cela peut faire beaucoup d'index.

Cet exemple d'une de nos formations[3] montre l'intérêt d'index GIN, GiST ou bloom pour une telle recherche multicritère.

Avec PostgreSQL 18, le B-tree peut redevenir intéressant dans certains cas favorables : très peu de valeurs différentes sur les premières colonnes de l'index, et peu de colonnes.

```
SELECT setseed (0.0); -- pour la reproductibilité
INSERT INTO demo_multi
SELECT n, random(0,2) AS i, random(0,2) AS j, random(0,2) AS k, n AS l
FROM generate_series (1,1000000) n
ORDER BY i,j,k,l;
VACUUM ANALYZE demo_multi ;

EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM demo_multi WHERE k=0 AND l=200002 ;
```

La dernière colonne est trouvée avec 13 recherches différentes :

[3] https://dali.bo/j5_html#indexation-multicolonne-gin-gist-bloom

```
 Index Scan using demo_multi_idx on demo_multi (actual time=0.096..0.116 rows=1.00
↪  loops=1)
   Index Cond: ((k = 0) AND (l = 200002))
   Index Searches: 13
   Buffers: shared hit=40
 Planning Time: 0.126 ms
 Execution Time: 0.140 ms
```

Cette recherche fonctionne aussi et donne lieu cette fois à un Bitmap Scan :

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM demo_multi WHERE j=0 AND k=0 ;
```

```
 Bitmap Heap Scan on demo_multi (actual time=7.341..20.521 rows=111392.00 loops=1)
   Recheck Cond: ((j = 0) AND (k = 0))
   Heap Blocks: exact=1378
   Buffers: shared hit=1677 read=141
   ->  Bitmap Index Scan on demo_multi_idx (actual time=7.025..7.026 rows=111392.00
↪  loops=1)
         Index Cond: ((j = 0) AND (k = 0))
         Index Searches: 4
         Buffers: shared hit=299 read=141
 Planning Time: 0.135 ms
 Execution Time: 26.388 ms
```

Cela reste plus intéressant qu'un *Seq Scan*.

Au final, le B-tree redevient une option de plus à tester pour les recherches multicritères, mais tout dépend du nombre de champs, de leur sélectivité, etc.

6.2 SUPPRESSION AUTOMATIQUE DE JOINTURES INUTILES

- Jointure d'une table avec elle-même
  - automatiquement supprimée
- Cette problématique arrive facilement
- 6 ans de discussion pour le patch !

Il semble absurde de joindre une table avec elle-même sur sa clé primaire, mais cela arrive plus souvent que l'on ne croit, surtout involontairement.

Il n'est pas rare que des utilisateurs n'aient accès qu'à des vues « de présentation », et qu'ils ajoutent une jointure sur une table déjà présente dans cette vue pour ajouter un champ manquant. Il arrive même qu'on le fasse en connaissance de cause quand le schéma n'est pas aisément modifiable C'est parfois en connaissance quand n'est pas aisément modifiable (pas les droits, pas le temps de valider, pas de contrôle sur l'ORM, la vue est fournie par un éditeur, etc.)

PostgreSQL sait à présent repérer ce cas et supprime une auto-jointure quand il peut prouver que pour chaque ligne :

- il y a au plus une ligne en face ;
- les deux lignes des deux côtés de la jointure seraient physiquement les mêmes (concrètement : même `ctid`) ;

Seules les tables, et avec une clé unique ou primaire, peuvent être optimisées. L'optimisation ne fonctionne pas pour les vues matérialisées, par exemple.

PostgreSQL 9.0 savait déjà éliminer des *left joins* inutiles. Pour la *self-join elimination*, plus complexe qu'elle n'en a l'air, il aura fallu une vingtaine de contributeurs expérimentés[4] et presque sept ans de discussions[5] ! Une première version avait même été intégrée à PostgreSQL 17 mais retirée avant sa parution à cause de bugs. Parmi les problèmes rencontrés : la complexité du code de l'optimiseur, l'habituelle crainte que la recherche de l'optimisation coûte cher en temps de planification pour toutes les requêtes, la quantité assez importante de code nécessaire pour un cas rare, l'endroit exact où placer l'optimisation, l'implémentation et l'optimisation du code, l'inhibition du traitement des *self-joins* lors d'UPDATE, ou avec `TABLESAMPLE`…, la recherche de cas tordus, un changement d'auteur en cours de route, l'impact sur l'extension `pg_hint_plan`…

[4] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=fc069a3a6319b5bf40d2f0f1efceae1c9b7a68a8
[5] https://www.postgresql.org/message-id/flat/64486b0b-0404-e39e-322d-0801154901f3%40postgrespro.ru

Un paramètre `enable_self_join_elimination` apparaît pour inhiber la fonctionnalité, ce qui ne devrait pas être nécessaire.

Exemple :

Partons d'une base générée par `pgbench`. Chaque intervenant utilise une vue pour rajouter une information qui lui manque dans la vue précédente :

```
DROP VIEW IF EXISTS pgbench_accounts_balance_v CASCADE ;

-- Première vue montrant juste l'ID et le compte courant
CREATE VIEW pgbench_accounts_balance_v
AS
SELECT aid, abalance
FROM pgbench_accounts a1 ;

-- L'utilisateur suivant ajoute le champ texte
CREATE VIEW pgbench_accounts_v2
AS
SELECT v.*, a2.filler AS libelle
FROM pgbench_accounts a2
INNER JOIN pgbench_accounts_balance_v v USING (aid) ;

-- On a besoin des infos sur la branche (pgbench_branches)
-- mais il faut trouver la clé dans pgbench_accounts
CREATE VIEW pgbench_accounts_v3
AS
SELECT v2.aid, v2.abalance, a3.bid, b3.bbalance
FROM pgbench_accounts_v2 v2
INNER JOIN pgbench_accounts a3 USING (aid)
INNER JOIN pgbench_branches b3 ON (a3.bid = b3.bid) ;

-- On re-rajoute le libellé oublié dans la vue précédente
-- et on teste que la ligne fait partie des lignes valides
CREATE VIEW pgbench_accounts_v4
AS
SELECT v3.*, a4.filler AS libelle_bis
FROM pgbench_accounts_v3 v3
INNER JOIN pgbench_accounts a4 USING (aid)
WHERE EXISTS (
     SELECT 'existe' FROM pgbench_accounts x
     WHERE filler IS NOT NULL
     AND x.aid = a4.aid
) ;

-- L'utilisateur rajoute le libellé côté branche
-- plus divers tests
EXPLAIN
```

```
SELECT v4.*, b5.filler AS libelle_branche
FROM pgbench_accounts_v4 v4
INNER JOIN pgbench_branches b5 USING (bid)
WHERE v4.aid IS NOT NULL AND v4.bbalance > 0 AND libelle_bis IS NOT NULL ;
```

Le plan de cette dernière requête avec PostgreSQL 18 ne comprend qu'une simple jointure par *hash join* entre les deux tables concernées, avec les divers filtres, les redondances en moins. Même le EXISTS est interprété comme une jointure et donc éliminé aussi.

```
 Hash Join  (cost=2.26..291299.76 rows=100000 width=457)
   Hash Cond: (x.bid = b3.bid)
   ->  Seq Scan on pgbench_accounts x  (cost=0.00..263935.00 rows=10000000 width=97)
         Filter: (filler IS NOT NULL)
   ->  Hash  (cost=2.25..2.25 rows=1 width=364)
         ->  Seq Scan on pgbench_branches b3  (cost=0.00..2.25 rows=1 width=364)
               Filter: (bbalance > 0)
```

Comparons avec le plan sous PostgreSQL 17 : il contient 18 nœuds[6], dont quatre *Seq Scan* et un *Index Scan* sur `pgbench_accounts`, et deux *Seq Scan* sur `pgbench_branches` !

Cette fonctionnalité est très pratique mais n'est pas infaillible. La requête suivante ajoute un filtre sur `aid` qui est propagé sur toutes les jointures, ce qui désactive la *self-join elimination*.

```
EXPLAIN (COSTS OFF)
SELECT * FROM pgbench_accounts_v4
WHERE libelle_bis IS NOT NULL
AND aid = 10000 ;
```

```
Nested Loop Semi Join
->  Nested Loop
        ->  Nested Loop
            ->  Nested Loop
                    ->  Hash Join
                        Hash Cond: (b3.bid = a3.bid)
                        ->  Seq Scan on pgbench_branches b3
                        ->  Hash
                                ->  Index Scan using pgbench_accounts_pkey on
 ↪  pgbench_accounts a3
                                    Index Cond: (aid = 10000)
                    ->  Index Only Scan using pgbench_accounts_pkey on
 ↪  pgbench_accounts a2
                        Index Cond: (aid = 10000)
            ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a1
                    Index Cond: (aid = 10000)
        ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a4
            Index Cond: (aid = 10000)
```

[6] https://explain.dalibo.com/plan/70699d4d5fdcd676

```
            Filter: (filler IS NOT NULL)
->  Index Scan using pgbench_accounts_pkey on pgbench_accounts x
        Index Cond: (aid = 10000)
        Filter: (filler IS NOT NULL)
```

Ce n'est pas ici un souci, car l'accès par les index est alors réellement rapide.

Voir aussi cet article de Guillaume Lelarge sur le blog Dalibo : PostgreSQL 18 - Suppression de jointures inutiles[7].

[7] https://blog.dalibo.com/2025/03/07/postgresql-18-self_join_elimination.html

6.3 CONVERSION DE CLAUSES OR EN TABLEAU

- OR converti si possible en ANY et tableau
- Ces clauses donnent enfin le même plan :

```
WHERE a = 1 OR a = 2 OR a = 3
WHERE a IN (1,2,3)
WHERE a = ANY ('{1,2,3}')
```

- Évite des accès et des *BitmapOR*

PostgreSQL sait à présent regrouper des clauses OR qui portent sur la même colonne pour faire un seul accès à l'index. C'était déjà le cas quand on utilisait `IN (…)` ou `= ANY (…)`. Les champs OR reconnus comme identiques sont regroupés en tableau.

Ces trois formulations donnent donc le même plan :

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo WHERE a = 1 OR a = 20 OR a = 300 ;
```

```
                    QUERY PLAN
--------------------------------------------------
 Index Scan using demo_a_idx on demo
   Index Cond: (a = ANY ('{1,20,300}'::integer[]))
```

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo WHERE a IN (1, 20, 300) ;
```

```
                    QUERY PLAN
--------------------------------------------------
 Index Scan using demo_a_idx on demo
   Index Cond: (a = ANY ('{1,20,300}'::integer[]))
```

```
EXPLAIN (COSTS OFF) SELECT * FROM demo
WHERE a = ANY (ARRAY[1,20,300]) ;
```

```
                    QUERY PLAN
--------------------------------------------------
 Index Scan using demo_a_idx on demo
   Index Cond: (a = ANY ('{1,20,300}'::integer[]))
```

Avec PostgreSQL 17, celle avec les OR aurait donné trois accès différents, que PostgreSQL aurait regroupés avec un *BitmapOR*, ce qui n'est généralement pas mauvais, mais pas optimal.

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo WHERE a = 1 OR a = 20 OR a = 300 ;
```

```
 Bitmap Heap Scan on demo
   Recheck Cond: ((a = 1) OR (a = 20) OR (a = 300))
   ->  BitmapOr
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 1)
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 20)
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 300)
```

Exemple :

La table pour l'exemple est la suivante, elle pèse presque 500 Mo :

```
SET jit TO off ;
DROP TABLE IF EXISTS demo ;
CREATE UNLOGGED TABLE demo (id bigint PRIMARY KEY,
                            a int, b int,
                            filler char (5) default ' ') ;
INSERT INTO demo SELECT i, i, i%1000000
FROM generate_series (1,10_000_000) i ;

CREATE INDEX demo_a_idx ON demo (a) ;
VACUUM ANALYZE demo ;
```

Une requête de ce genre avec de nombreux OR peut se rencontrer :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM demo
WHERE
 a=0 OR a=100000 OR a=200000 OR a=300000 OR a=400000 OR a=500000 OR a=600000 OR
 ↪  a=700000 OR a=800000 OR a=900000 OR a=1000000 OR a=1100000 OR a=1200000 OR
 ↪  a=1300000 OR a=1400000 OR a=1500000 OR a=1600000 OR a=1700000 OR a=1800000 OR
 ↪  a=1900000 OR a=2000000 OR a=2100000 OR a=2200000 OR a=2300000 OR a=2400000 OR
 ↪  a=2500000 OR a=2600000 OR a=2700000 OR a=2800000 OR a=2900000 OR a=3000000 OR
 ↪  a=3100000 OR a=3200000 OR a=3300000 OR a=3400000 OR a=3500000 OR a=3600000 OR
 ↪  a=3700000 OR a=3800000 OR a=3900000 OR a=4000000 OR a=4100000 OR a=4200000 OR
 ↪  a=4300000 OR a=4400000 OR a=4500000 OR a=4600000 OR a=4700000 OR a=4800000 OR
 ↪  a=4900000 OR a=5000000 OR a=5100000 OR a=5200000 OR a=5300000 OR a=5400000 OR
 ↪  a=5500000 OR a=5600000 OR a=5700000 OR a=5800000 OR a=5900000 OR a=6000000 OR
 ↪  a=6100000 OR a=6200000 OR a=6300000 OR a=6400000 OR a=6500000 OR a=6600000 OR
 ↪  a=6700000 OR a=6800000 OR a=6900000 OR a=7000000 OR a=7100000 OR a=7200000 OR
 ↪  a=7300000 OR a=7400000 OR a=7500000 OR a=7600000 OR a=7700000 OR a=7800000 OR
 ↪  a=7900000 OR a=8000000 OR a=8100000 OR a=8200000 OR a=8300000 OR a=8400000 OR
 ↪  a=8500000 OR a=8600000 OR a=8700000 OR a=8800000 OR a=8900000 OR a=9000000 OR
 ↪  a=9100000 OR a=9200000 OR a=9300000 OR a=9400000 OR a=9500000 OR a=9600000 OR
 ↪  a=9700000 OR a=9800000 OR a=9900000 OR a=10000000 ;
```

Le plan complet avec PostgreSQL 17 est sur explain.dalibo.com[8], il est assez long (103 nœuds) :

```
Bitmap Heap Scan on demo  (cost=451.24..869.69 rows=101 width=22) (actual
↪  time=0.214..0.265 rows=100 loops=1)
  Recheck Cond: ((a = 0) OR (a = 100000) OR (a = 200000) OR (a = 300000) OR (a =
↪  400000) OR (a = 500000) OR (a = 600000) OR (a = 700000) OR (a = 800000) OR (a =
↪  900000) OR (a = 1000000) OR (a = 1100000) OR (a = 1200000) OR (a = 1300000) OR (a
↪  = 1400000) OR (a = 1500000) OR (a = 1600000) OR (a = 1700000) OR (a = 1800000) OR
↪  (a = 1900000) OR (a = 2000000) OR (a = 2100000) OR (a = 2200000) OR (a = 2300000)
↪  OR (a = 2400000) OR (a = 2500000) OR (a = 2600000) OR (a = 2700000) OR (a =
↪  2800000) OR (a = 2900000) OR (a = 3000000) OR (a = 3100000) OR (a = 3200000) OR
↪  (a = 3300000) OR (a = 3400000) OR (a = 3500000) OR (a = 3600000) OR (a = 3700000)
↪  OR (a = 3800000) OR (a = 3900000) OR (a = 4000000) OR (a = 4100000) OR (a =
↪  4200000) OR (a = 4300000) OR (a = 4400000) OR (a = 4500000) OR (a = 4600000) OR
↪  (a = 4700000) OR (a = 4800000) OR (a = 4900000) OR (a = 5000000) OR (a = 5100000)
↪  OR (a = 5200000) OR (a = 5300000) OR (a = 5400000) OR (a = 5500000) OR (a =
↪  5600000) OR (a = 5700000) OR (a = 5800000) OR (a = 5900000) OR (a = 6000000) OR
↪  (a = 6100000) OR (a = 6200000) OR (a = 6300000) OR (a = 6400000) OR (a = 6500000)
↪  OR (a = 6600000) OR (a = 6700000) OR (a = 6800000) OR (a = 6900000) OR (a =
↪  7000000) OR (a = 7100000) OR (a = 7200000) OR (a = 7300000) OR (a = 7400000) OR
↪  (a = 7500000) OR (a = 7600000) OR (a = 7700000) OR (a = 7800000) OR (a = 7900000)
↪  OR (a = 8000000) OR (a = 8100000) OR (a = 8200000) OR (a = 8300000) OR (a =
↪  8400000) OR (a = 8500000) OR (a = 8600000) OR (a = 8700000) OR (a = 8800000) OR
↪  (a = 8900000) OR (a = 9000000) OR (a = 9100000) OR (a = 9200000) OR (a = 9300000)
↪  OR (a = 9400000) OR (a = 9500000) OR (a = 9600000) OR (a = 9700000) OR (a =
↪  9800000) OR (a = 9900000) OR (a = 10000000))
  Heap Blocks: exact=100
  Buffers: shared hit=403
  ->  BitmapOr  (cost=451.24..451.24 rows=101 width=0) (actual time=0.201..0.207
↪  rows=0 loops=1)
        Buffers: shared hit=303
        ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.008..0.008 rows=0 loops=1)
              Index Cond: (a = 0)
              Buffers: shared hit=3
        ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.003..0.003 rows=1 loops=1)
              Index Cond: (a = 100000)
              Buffers: shared hit=3
        ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.003..0.003 rows=1 loops=1)
              Index Cond: (a = 200000)
              Buffers: shared hit=3
        ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.002..0.002 rows=1 loops=1)
              Index Cond: (a = 300000)
```

[8] https://explain.dalibo.com/plan/4dd8eeedg9ce6d6c

```
               Buffers: shared hit=3
         …
         …
         ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.002..0.002 rows=1 loops=1)
               Index Cond: (a = 9900000)
               Buffers: shared hit=3
         ->  Bitmap Index Scan on demo_a_idx  (cost=0.00..4.44 rows=1 width=0)
↪  (actual time=0.002..0.002 rows=1 loops=1)
               Index Cond: (a = 10000000)
               Buffers: shared hit=3
 Planning Time: 0.222 ms
 Execution Time: 0.372 ms
```

Cela reste bon, car l'essentiel du temps est perdu à aller chercher des blocs dans la table. Toujours avec PostgreSQL 17, le plan équivalent avec `IN` prend un tiers de temps en moins (une fois tout effet de cache supprimé) et n'utilise qu'un nœud :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM demo
WHERE a IN (
 0,100000,200000,300000,400000,500000,600000,700000,800000,900000,
 1000000,1100000,1200000,1300000,1400000,1500000,1600000,1700000,1800000,1900000,
 2000000,2100000,2200000,2300000,2400000,2500000,2600000,2700000,2800000,2900000,
 3000000,3100000,3200000,3300000,3400000,3500000,3600000,3700000,3800000,3900000,
 4000000,4100000,4200000,4300000,4400000,4500000,4600000,4700000,4800000,4900000,
 5000000,5100000,5200000,5300000,5400000,5500000,5600000,5700000,5800000,5900000,
 6000000,6100000,6200000,6300000,6400000,6500000,6600000,6700000,6800000,6900000,
 7000000,7100000,7200000,7300000,7400000,7500000,7600000,7700000,7800000,7900000,
 8000000,8100000,8200000,8300000,8400000,8500000,8600000,8700000,8800000,8900000,
 9000000,9100000,9200000,9300000,9400000,9500000,9600000,9700000,9800000,9900000,
 10000000
 ) ;
```

```
 Index Scan using demo_a_idx on demo  (cost=0.43..453.70 rows=101 width=22) (actual
↪  time=0.011..0.193 rows=100 loops=1)
   Index Cond: (a = ANY
↪  ('{0,100000,200000,300000,400000,500000,600000,700000,800000,900000,
   1000000,1100000,1200000,1300000,1400000,1500000,1600000,1700000,1800000,1900000,
   2000000,2100000,2200000,2300000,2400000,2500000,2600000,2700000,2800000,2900000,
   3000000,3100000,3200000,3300000,3400000,3500000,3600000,3700000,3800000,3900000,
   4000000,4100000,4200000,4300000,4400000,4500000,4600000,4700000,4800000,4900000,
   5000000,5100000,5200000,5300000,5400000,5500000,5600000,5700000,5800000,5900000,
   6000000,6100000,6200000,6300000,6400000,6500000,6600000,6700000,6800000,6900000,
   7000000,7100000,7200000,7300000,7400000,7500000,7600000,7700000,7800000,7900000,
   8000000,8100000,8200000,8300000,8400000,8500000,8600000,8700000,8800000,8900000,
   9000000,9100000,9200000,9300000,9400000,9500000,9600000,9700000,9800000,9900000,
   10000000}'::integer[]))
```

```
   Buffers: shared hit=403
 Planning Time: 0.061 ms
 Execution Time: 0.202 ms
```

(version graphique[9])

Avec PostgreSQL 18, la première requête avec OR se retrouve avec ce dernier plan et donc des performances améliorées (et, au passage, un plan nettement plus lisible).

De plus, comme ce plan n'utilise qu'un *Index Scan*, il a accès à l'optimisation des recherches dans ce type de nœud :

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT * FROM demo
WHERE a = 1 OR a = 10 OR a = 20 OR a = 10_000_000 ;
```

```
 Index Scan using demo_a_idx on demo (actual time=0.037..0.061 rows=4.00 loops=1)
   Index Cond: (a = ANY ('{1,10,20,10000000}'::integer[]))
   Index Searches: 2
   Buffers: shared hit=8
 Planning Time: 0.143 ms
 Execution Time: 0.083 ms
```

Les trois premières valeurs sont assez proches pour être récupérées ensemble, il n'y a donc que deux recherches (`Index Searches: 2`), à raison de 4 blocs par recherche ici.

Le critère regroupé dans OR est ici basique. Mais il peut être beaucoup plus complexe. Ceci utilise un index fonctionnel :

```
CREATE INDEX ON demo ((a+b)) ;
VACUUM ANALYZE demo ;

EXPLAIN (COSTS OFF)
SELECT * FROM demo
WHERE (a+b) = 1 OR (a+b) = 2 ;
```

```
 Index Scan using demo_expr_idx on demo
   Index Cond: ((a + b) = ANY ('{1,2}'::integer[]))
```

Limites :

L'algorithme n'est pas infaillible. Par exemple, il ne sait pas fusionner ces clauses pourtant équivalentes :

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo
WHERE a = ANY (ARRAY[1,2]) OR a IN (3,4) OR a = 5;
```

[9] https://explain.dalibo.com/plan/76651974844759ag

```
 Bitmap Heap Scan on demo
   Recheck Cond: ((a = ANY ('{1,2}'::integer[])) OR (a = ANY ('{3,4}'::integer[])) OR
↪   (a = 5))
   ->  BitmapOr
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = ANY ('{1,2}'::integer[]))
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = ANY ('{3,4}'::integer[]))
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 5)
```

Ceci n'est pas non plus réécrit comme un ANY, il faut deux *Bitmap Scan* et le *Recheck* est nécessaire :

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo  WHERE a=5 OR (a=10 and b=2) ;
```

```
 Bitmap Heap Scan on demo
   Recheck Cond: ((a = 5) OR (a = 10))
   Filter: ((a = 5) OR ((a = 10) AND (b = 2)))
   ->  BitmapOr
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 5)
         ->  Bitmap Index Scan on demo_a_idx
               Index Cond: (a = 10)
```

Par contre, si la clause supplémentaire b=2 est « factorisable », PostgreSQL 18 sait à nouveau regrouper les OR :

```
EXPLAIN (COSTS OFF)
SELECT * FROM demo
WHERE ((a=5 OR a=6) and b=2)
    OR (a =10 AND b=2)
    OR (b=2 AND a=20) ;
```

```
 Index Scan using demo_ab_idx on demo
   Index Cond: ((b = 2) AND (a = ANY ('{5,6,10,20}'::integer[])))
```

7/ Supervision

7.1 VUES DE SUPERVISION

- `pg_stat_database`
  - nouvelles colonnes `parallel_workers_to_launch` et `parallel_workers_launched`
- `pg_stat_all_tables`
  - plusieurs colonnes indiquant le temps passé sur les opérations VACUUM et ANALYZE
- `pg_stat_io`
  - indique l'activité en octets (et non en blocs)
  - indique l'activité sur les journaux de transactions
  - `track_wal_io_timing` à activer pour avoir les durées
- `pg_stat_wal`
  - suppressions des colonnes `read` et `sync`
- `pg_stat_checkpointer`
  - nouvelles colonnes `num_done` et `slru_written`

Vue pg_stat_database

La vue `pg_stat_database` comporte deux nouvelles colonnes. `parallel_workers_to_launch` indique le nombre total de *workers* de parallélisation à lancer, et `parallel_workers_launched` le nombre total de *workers* de parallélisation lancés. Cela devrait permettre de savoir si la parallélisation est fréquemment demandée (première colonne avec une valeur haute) et si la configuration est suffisante (première et deuxième colonnes de valeurs équivalentes). Sinon il faut peut-être ajuster les paramètres du parallélisme comme `max_parallel_workers_per_gather` ou `max_parallel_workers` (voir plus loin).

Vue pg_stat_all_tables

La table `pg_stat_all_tables` (et par conséquent ses deux variantes `pg_stat_sys_tables` et `pg_stat_user_tables`) se voit ajouter quatre nouvelles colonnes :

- `total_vacuum_time`, durée totale des opérations VACUUM manuelles sur cette table ;

- `total_autovacuum_time`, durée totale des opérations VACUUM automatiques sur cette table ;
- `total_analyze_time`, durée totale des opérations ANALYZE manuelles sur cette table ;
- `total_autoanalyze_time`, durée totale des opérations ANALYZE automatiques sur cette table.

Il est à noter qu'il n'est pas nécessaire d'activer `track_io_timing` pour que cette durée soit comptabilisée.

Voici un exemple concernant uniquement la colonne `total_vacuum_time` :

```
create table t1(c1 integer);
alter table t1 set (autovacuum_enabled=off);
insert into t1 select generate_series(1, 10_000_000);

select relname, total_vacuum_time from pg_stat_all_tables where relname='t1';
```

```
┌──────────┬────────────────────┐
│ relname  │ total_vacuum_time  │
├──────────┼────────────────────┤
│ t1       │                  0 │
└──────────┴────────────────────┘
(1 row)

vacuum t1;
VACUUM
Time: 2225.883 ms (00:02.226)

select relname, total_vacuum_time from pg_stat_all_tables where relname='t1';
```

```
┌──────────┬────────────────────┐
│ relname  │ total_vacuum_time  │
├──────────┼────────────────────┤
│ t1       │               2225 │
└──────────┴────────────────────┘
(1 row)
```

Comme la table n'a jamais été traitée, elle doit être entièrement lue, ce qui cause un long traitement. Les 2225 ms chronométrées par `psql` sont bien visibles dans la vue statistique. Un deuxième VACUUM devrait être bien plus rapide.

```
vacuum t1;
```

```
VACUUM
Time: 0.975 ms

select relname, total_vacuum_time from pg_stat_all_tables where relname='t1';
┌─────────┬────────────────────┐
│ relname │ total_vacuum_time  │
├─────────┼────────────────────┤
│ t1      │               2226 │
└─────────┴────────────────────┘
(1 row)
```

C'est bien le cas, il n'a pris qu'une seule milliseconde, qui se trouve bien comptabilisée dans la vue statistique.

Supprimons un bon lot de lignes pour qu'il ait du travail :

```
delete from t1 where c1<100000;

vacuum t1;
VACUUM
Time: 5.012 ms

select relname, total_vacuum_time from pg_stat_all_tables where relname='t1';
┌─────────┬────────────────────┐
│ relname │ total_vacuum_time  │
├─────────┼────────────────────┤
│ t1      │               2231 │
└─────────┴────────────────────┘
(1 row)
```

Là aussi, nous pouvons constater que ce nouveau traitement a été pris en compte dans la vue statistique.

Vues pg_stat_io et pg_stat_wal

La vue `pg_stat_io` intègre maintenant des statistiques sur les entrées/sorties disque des processus *WAL writer*, *WAL receiver* et *WAL summarizer*, ainsi que les processus de communication avec les clients.

Avec la requête suivante :

```
select backend_type, object, count(*)
from pg_stat_io
where object='wal' or backend_type like 'wal%'
group by 1, 2
order by 1, 2;
```

Voici les statistiques en version 17 :

backend_type	object	count
walsender	relation	4
walsender	temp relation	1

Et celles en version 18 :

backend_type	object	count
autovacuum launcher	wal	2
autovacuum worker	wal	2
background worker	wal	2
background writer	wal	2
checkpointer	wal	2
client backend	wal	2
io worker	wal	2
slotsync worker	wal	2
standalone backend	wal	2
startup	wal	2
walreceiver	wal	2
walsender	relation	5
walsender	temp relation	1
walsender	wal	2
walsummarizer	wal	2
walwriter	wal	2

Le nombre d'informations est bien plus important. Certaines informations étant disponibles sur

`pg_stat_io` et `pg_stat_wal`, elles ont été supprimées de cette dernière. Il s'agit des colonnes `read` et `sync`.

Pour terminer sur la vue `pg_stat_io`, l'activité est maintenant indiqué en octets et non plus en blocs.

Vue pg_stat_checkpointer

Enfin, la vue `pg_stat_checkpointer` dispose de deux nouvelles colonnes. `num_done` indique le nombre de *checkpoints* réalisés, et `slru_written` indique le nombre de blocs écrits à partir des caches *SLRU*.

7.2 INFORMATIONS SUR LE VACUUM ET L'ANALYZE

- Délai suite à un dépassement de coût
  - nouvelle colonne `delay_time` dans `pg_stat_progress_vacuum`
  - nouvelle colonne `delay_time` dans `pg_stat_progress_analyze`
  - trace supplémentaire avec l'option VERBOSE
- Trace supplémentaire sur les WAL
  - *buffers full*

Un VACUUM peut se voir appliquer une limitation des IO. C'est d'ailleurs le cas par défaut pour un VACUUM lancé par l'`autovacuum` Dans ce cas, le VACUUM se met en pause un certain temps (`[auto]vacuum_cost_delay`) à chaque fois que le coût de l'opération dépasse une certaine limite (`[auto]vacuum_cost_limit`). Un VACUUM manuel n'est par défaut pas soumis à ce bridage. Pour les détails, voir notre formation DBA2[1].

Cette pause ralentit l'opération de VACUUM et il est intéressant de savoir le temps qu'a pris ces pauses par rapport au temps total de l'opération. C'est ce que propose la nouvelle version. Cette information est disponible à plusieurs endroits.

Elle apparaît dans la colonne `delay_time` des vues de progression `pg_stat_progress_vacuum` et `pg_stat_progress_analyze`.

Le nouveau paramètre `track_cost_delay_timing` doit être passé à on pour que ce délai soit calculé et affiché.

En voici un exemple. Après voir chargé une table `demo`, l'autovacuum finit par se déclencher :

```
SELECT * FROM pg_stat_progress_vacuum;
```

```
┌─[ RECORD 1 ]──────┬──────────────┐
│ pid               │ 264181       │
│ datid             │ 114910       │
│ datname           │ demo         │
│ relid             │ 230753       │
│ phase             │ scanning heap │
```

[1] https://dali.bo/m5_html#param%C3%A9trage-de-vacuum-autovacuum

```
│ heap_blks_total       │ 44248          │
│ heap_blks_scanned     │ 37305          │
│ heap_blks_vacuumed    │ 0              │
│ index_vacuum_count    │ 0              │
│ max_dead_tuple_bytes  │ 67108864       │
│ dead_tuple_bytes      │ 0              │
│ num_dead_item_ids     │ 0              │
│ indexes_total         │ 0              │
│ indexes_processed     │ 0              │
│ delay_time            │ 7688.594373    │
```

Nous voyons ici que ce VACUUM a passé plus de 7 secondes en pause. Le temps total du VACUUM doit se calculer autrement, par exemple, en récupérant le temps d'exécution du VACUUM sur la vue `pg_stat_activity`. Cela nous donnerait la requête suivante :

```
SELECT a.pid, c.relname AS "table",
  round(pv.delay_time) AS delay_time,
  round((extract(epoch from now()) - extract(epoch from a.query_start)) * 1000) AS
  ↪  total_time
FROM pg_stat_activity a
JOIN pg_stat_progress_vacuum pv ON a.pid=pv.pid
JOIN pg_class c ON pv.relid=c.oid
ORDER BY 1, 2;
```

Et le résultat suivant :

```
┌───────┬───────┬────────────┬────────────┐
│  pid  │ table │ delay_time │ total_time │
├───────┼───────┼────────────┼────────────┤
│ 10740 │ t1    │      31487 │      33245 │
└───────┴───────┴────────────┴────────────┘
```

Avec de bons disques (notamment SSD) où les lectures et écritures sont très rapides, il est normal que le VACUUM passe l'essentiel de son temps en pause.

Toujours si `track_cost_delay_timing` est activé, ce délai apparaît dans la sortie de VACUUM (VERBOSE) ou dans les traces (ligne *delay time*), comme ici :

```
INFO:  vacuuming "demo.public.t1"
INFO:  finished vacuuming "demo.public.t1": index scans: 0
```

```
pages: 0 removed, 44248 remain, 444 scanned (1.00% of total), 0 eagerly scanned
tuples: 99999 removed, 9999977 remain, 0 are dead but not yet removable
removable cutoff: 55363, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 443 pages set all-visible, 443 pages set all-
frozen (0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifier
delay time: 40.413 ms
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.170 MB/s
buffer usage: 910 hits, 0 reads, 1 dirtied
WAL usage: 887 records, 2 full page images, 266495 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.04 s
```

À noter aussi dans cette sortie le nombre de *buffers full* pour les journaux de transactions. Cet ajout permet de connaître le nombre de fois où le cache des journaux de transactions était rempli à cause des écritures des opérations VACUUM et ANALYZE.

7.3 STATISTIQUES SUR LES I/O PAR BACKEND

- Deux nouvelles fonctions
  - `pg_stat_get_backend_io()`
  - `pg_stat_reset_backend_stats()`
- Expose des données similiaires à `pg_stat_io`
- Pour les backends clients, processus auxilaires, le *WAL writer*, le *WAL receiver* et le *WAL summarizer*

Il est désormais possible de suivre les IO générées par un backend spécifique, grâce à la fonction `pg_stat_get_backend_io()`[2] qui prend en paramètre le PID du backend dont on souhaite observer l'activité.

```
-[ RECORD 1 ]-------+-------------------------------------------------------------------------------
Schema              | pg_catalog
Name                | pg_stat_get_backend_io
Result data type    | SETOF record
Argument data types | backend_pid integer, OUT backend_type text, OUT object text, OUT context text,
                    | OUT reads bigint, OUT read_bytes numeric, OUT read_time double precision,
                    | OUT writes bigint, OUT write_bytes numeric, OUT write_time double precision,
                    | OUT writebacks bigint, OUT writeback_time double precision, OUT extends bigint,
                    | OUT extend_bytes numeric, OUT extend_time double precision, OUT hits bigint,
                    | OUT evictions bigint, OUT reuses bigint, OUT fsyncs bigint,
                    | OUT fsync_time double precision, OUT stats_reset timestamp with time zone
Type                | func
```

Les informations retournées par la fonction sont les mêmes que celles renvoyées par la vue `pg_stat_io`[3] mais pour un backend spécifique. Seules les statistiques des backends clients, des processus auxilaires, du *WAL writer*, du *WAL receiver* et du *WAL summarizer* sont accessibles avec cette fonction. Les autres types de backends ne sont pas supportés. Ils sont pour la plupart visibles dans les données aggrégées par la vue `pg_stat_io`.

Cette fonction permet donc de suivre les IO sur les relations dans et hors de la mémoire partagée, ainsi que l'activité dans les fichiers temporaires et les WAL. Voici les différentes rubriques disponibles, elles ne le sont pas pour tous les backends, le *wal writer* n'a par exemple que les objets WAL.

```
    object      |  context
----------------+-----------
 relation       | bulkread
```

[2] https://www.postgresql.org/docs/18/monitoring-stats.html#PG-STAT-GET-BACKEND-IO
[3] https://www.postgresql.org/docs/18/monitoring-stats.html#MONITORING-PG-STAT-IO-VIEW

```
 relation      | bulkwrite
 relation      | init
 relation      | normal
 relation      | vacuum
 temp relation | normal
 wal           | init
 wal           | normal
```

Les statistiques existent pour la durée de vie du backend. La fonction `pg_stat_reset_backend_stats()` qui prend aussi en paramètre le PID d'un backend permet de réinitialiser ces statistiques.

7.4 PG_STAT_STATEMENTS

- Normalisation des commandes SET
- Traçage des sous-requêtes de DECLARE CURSOR et CREATE TABLE AS
- Nouvelles colonnes dans pg_stat_statements
  - parallel_workers_to_launch : workers demandés par le planificateur
  - parallel_workers_launched : workers effectivement lancés
  - wal_buffers_full : nombre de fois que le WAL buffer s'est rempli

Traçage des sous-requêtes de DECLARE CURSOR et CREATE TABLE AS :

PostgreSQL permet désormais de calculer l'identifiant des sous-requêtes exécutées par les commandes DECLARE CURSOR et CREATE TABLE AS, et met à disposition des hooks pour que les extensions puissent récupérer ces ID. De plus, EXPLAIN expose désormais l'ID de la sous-requête.

pg_stat_statements exploite ces nouvelles informations et affiche deux lignes. Pour que l'ID de la sous-requête soit enregistré, il faut configurer pg_stat_statements.track à all.

Voici un exemple avec CREATE TABLE AS :

```
DROP TABLE matable;
SELECT pg_stat_statements_reset();
SET pg_stat_statements.track TO 'all';
-- Requête 1
CREATE TABLE matable AS
   SELECT relname, relpages
     FROM pg_class
    WHERE relname LIKE '%class%';
DROP TABLE matable;
-- Requete 2
EXPLAIN (ANALYZE, VERBOSE, TIMING OFF, COSTS OFF)
   CREATE TABLE matable AS
      SELECT relname, relpages
        FROM pg_class
       WHERE relname LIKE '%class%';
SELECT ROW_NUMBER() OVER (), substr(query, 1, 100) AS  query, toplevel, queryid,
↪  calls
  FROM pg_stat_statements;

                          QUERY PLAN
------------------------------------------------------------
```

```
 Seq Scan on pg_catalog.pg_class (actual rows=7.00 loops=1)
   Output: relname, relpages
   Filter: (pg_class.relname ~~ '%class%'::text)
   Rows Removed by Filter: 416
   Buffers: shared hit=14
 Query Identifier: -1352204053524932295
 Planning Time: 0.036 ms
 Execution Time: 0.260 ms
(8 rows)
```

Dans le tableau suivant :

- L'entrée 1 correspond à la requête 1.
- L'entrée 4 (avec `toplevel` à `false`) correspond à la sous-requête commune aux requêtes 1 et 2.
- L'entrée 5 correspond à la requête 2.

```
 row_number |                      query                       | toplevel |       queryid        | calls
------------+--------------------------------------------------+----------+----------------------+-------
          1 | CREATE TABLE matable AS                         +| t        |  5601689128997531384 |     1
            |    SELECT relname, relpages                     +|          |                      |
            |      FROM pg_class                              +|          |                      |
            |     WHERE relname LIKE $1                        |          |                      |
          2 | SELECT pg_stat_statements_reset()                | t        |  5619685836067787454 |     1
          3 | DROP TABLE matable                               | t        |  4889846160454277914 |     1
          4 | CREATE TABLE matable AS                         +| f        | -1352204053524932295 |     2
            |    SELECT relname, relpages                     +|          |                      |
            |      FROM pg_class                              +|          |                      |
            |     WHERE relname LIKE $1;                       |          |                      |
          5 | EXPLAIN (ANALYZE, VERBOSE, TIMING OFF, COSTS OFF)+| t        |  4695920809708743906 |     1
            |    CREATE TABLE matable AS                      +|          |                      |
            |       SELECT relname, r                          |          |                      |
(5 rows)
```

Normalisation des commandes SET :

Le nombre d'entrées dans `pg_stat_statements` est, par défaut, limité à 5000 (`pg_stat_statements.max`). Lorsqu'il n'y a plus d'entrées libres pour enregistrer les statistiques d'une nouvelle requête, elle prend la place des statistiques de la requête la moins exécutée. La version 18 de PostgreSQL normalise les commandes SET ce qui permet d'éviter de polluer la vue avec de multiples exécutions de la même commande avec des valeurs de paramètres différentes.

```
SELECT pg_stat_statements_reset();
SET work_mem = '64MB';   -- entrée 1
SET work_mem = '128MB';  -- entrée 2
SET work_mem = '256MB';  -- entrée 3
SELECT query, calls FROM pg_stat_statements;
```

```
               query                | calls
------------------------------------+-------
 SELECT pg_stat_statements_reset() |     1
 SET work_mem = $1                 |     3
(2 rows)
```

Observabilité du parallélisme :

PostgreSQL dispose de plusieurs paramètes pour déterminer si le parallélisme doit être exécuté et combien de processus auxiliaires il doit/peut utiliser :

- `min_parallel_index_scan_size` et `min_parallel_table_scan_size` : déterminent la taille minimale de la relation (index ou table) et participent au calcul du nombre de workers à utiliser ;
- `max_parallel_workers_per_gather` : définit le nombre maximal de workers que peut lancer un nœud Gather.
- `parallel_setup_cost` et `parallel_tuple_cost` permettent au planificateur d'estimer le coût de ces opérations.

PostgreSQL dispose aussi de paramétrage pour limiter l'usage des processus auxiliaires de parallélisation :

- `max_parallel_workers` : le nombre de processus utilisables simultanément pour le parallélisme ;
- `max_parallel_maintenance_workers` : le nombre de processus utilisables simultanément par les tâches de maintenances (inclus dans le précédent) ;
- `max_worker_processes` : le nombre processus auxiliaires utilisables simultanément par le parallélisme, la réplication logique et les extensions.

Il est difficile de savoir si le parallélisme est correctement utilisé. Il est tout à fait possible que PostgreSQL estime pouvoir tirer parti du parallélisme, génère un plan adapté, mais ne dispose pas d'assez de *background workers* disponibles pour l'exécuter comme il l'avait planifié. La requête utilise alors moins de *workers* que prévu, voire, dans le pire des cas, paye le coût de la mise en place du parallélisme sans en profiter du tout.

La version 18 ajoute deux colonnes à `pg_stat_statements` :

- `parallel_workers_to_launch` : nombre cumulé de workers de parallélisation planifiés ;
- `parallel_workers_launched` : nombre cumulé de workers de parallélisation réellement lancés.

Ces statistiques ne concernent que les nœuds `Gather` et `Gather merge` des requêtes utilisateurs (le `VACUUM` et les créations d'index parallélisés sont donc exclus). Elles permettent de déterminer quelles requêtes présentes dans `pg_stat_statements` utilisent le parallélisme, et lesquelles devaient l'utiliser mais n'ont pas obtenu les processus auxiliaires nécessaires. En historisant ces données, on peut avoir une idée de l'évolution de ces informations dans le temps et adapter la configuration.

Ces informations viennent compléter les nouvelles colonnes `parallel_workers_to_launch` et `parallel_workers_launched` de la vue `pg_stat_database`.

Remplissage des WAL buffers :

Les journaux de transaction de PostgreSQL permettent de garantir qu'en cas de crash de l'instance les modifications validées par les utilisateurs de la base de données sont préservées. Ces modifications sont écrites dans les WAL au fur et à mesure, par oppositions aux fichiers de données qui sont mis à jour en différé par le processus *checkpointer*, *background writer* et éventuellement les *backends clients*. Afin de regrouper les écritures, les enregistrements de WAL sont écrits dans un *buffer* circulaire, le *wal buffer*, qui est vidé par le processus *wal writer* lorsqu'il est plein, que `wal_writer_delay` (200 ms) s'est écoulé ou qu'un utilisateur a fait un commit (on assume ici que `synchronous_commit` est a on).

Lorsque le *buffer* est plein, il n'est plus possible de faire des écritures tant qu'il n'a pas été vidé. Ce n'est pas un problème avec des charges de travail composées de petites transactions sans trop de contentions. Mais parfois c'est un facteur limitant.

La taille du *wal buffer* est par défaut calculée par PostgreSQL et vaut 3% de `shared_buffers` avec un maximum de 16 Mo, la taille d'un WAL (quand `wal_buffers = -1`). Il est possible de la définir explicitement et d'aller plus haut.

Depuis la version 14, il est possible d'obtenir des statistiques sur l'utilisation des WAL via la vue `pg_stat_wal` dont le nombre de fois que le *wal writer* s'est déclenché car le *buffer* était plein. C'est une information intéressante qui peut mener à une augmentation du paramètre `wal_buffers`.

Avec la version 18, il est possible de savoir le nombre de fois que cet événement s'est produit par requête via la nouvelle colonne `wal_buffers_full` de `pg_stat_statements`.

Cet exemple compare deux insertions légèrement différentes, autour de 4 Mo (qui est la taille allouée quand `shared_buffers = '128MB'`, la configuration par défaut) :

```
DROP TABLE t1;
CREATE table t1 (id int ) ;
SELECT  pg_stat_statements_reset() ; -- purge de pg_stat_statements
EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT i   FROM generate_series (1,60000) i;
EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT i+1 FROM generate_series (1,70000) i ;
SELECT
    substring(query, 1, 50) as query,
    calls, round(mean_exec_time::numeric,1) AS "temps ms",
    wal_records,                                    -- nombre de journaux générés
    pg_size_pretty(wal_bytes::bigint) as wal_size, -- leur taille
    wal_buffers_full,
    current_setting('wal_buffers') AS wal_buffers,
    round(100.0 * wal_buffers_full / wal_records, 2) as pct_full
FROM  pg_stat_statements
```

```
WHERE wal_records > 0
ORDER BY wal_buffers_full DESC
LIMIT 10 \gx
```

Pour la première requête, 0,05 % des écritures d'enregistrements de journaux sont en attente, alors qu'il n'y a que peu de lignes à insérer en plus. (L'impact sur les performances n'est pas significatif sur un si petit exemple.)

```
-[ RECORD 1 ]----+--------------------------------------------------
query            | EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT i+$1
calls            | 1
temps ms         | 24.0
wal_records      | 70000
wal_size         | 4033 kB
wal_buffers_full | 37
wal_buffers      | 4MB
pct_full         | 0.05
-[ RECORD 2 ]----+--------------------------------------------------
query            | EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT i   F
calls            | 1
temps ms         | 18.9
wal_records      | 60000
wal_size         | 3457 kB
wal_buffers_full | 0
wal_buffers      | 4MB
pct_full         | 0.00
```

Attention : sur un système normalement sollicité, rien ne nous garantit que la requête pour qui `wal_buffers_full` a été incrémenté est à l'origine de toutes les écritures. Cependant, en historisant ces données, cela permet d'expliquer certaines instabilités dans les temps d'exécutions.

7.5

8/ Questions

Merci de votre écoute !
Nouveautés de la version 18 :
https://dali.bo/workshop18_html
https://dali.bo/workshop18_pdf

Notes

Notes

Notes

Nos autres publications

FORMATIONS

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

LIVRES BLANCS

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

9/ DALIBO, L'Expertise PostgreSQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.