Workshop 19

Nouveautés de PostgreSQL 19



19

Contents

1/ Introduction **1**

2/ Utilisation **3**

2.1 INSERT … ON CONFLICT … DO SELECT 4

2.2 IGNORE NULLS/RESPECT NULLS pour certaines fonctions de fenêtrage 6

2.3 Ajout du format JSON pour 'COPY … TO' 8

2.4 Autres nouvelles fonctionnalités de COPY 11

3/ Administration **15**

3.1 Autovacuum : parallélisation du nettoyage des index 16

3.2 Autovacuum : tri des tables par score 19

3.3 REPACK / REPACK USING INDEX 21

3.4 REPACK (CONCURRENTLY) 25

3.4.1 Fonctionnement 25

3.4.2 Limites 26

3.5 Améliorations de psql 29

3.6 Améliorations des autres outils 33

4/ Performances **35**

4.1 La question des hints 36

4.2 Que fait la concurrence ? 37

4.3 Que fait donc PostgreSQL v19 ? 38

4.4 Module pg_plan_advice 39

4.5 Module pg_stash_advice 44

4.6 `default_toast_compression = lz4` 48

4.7 Nouvelles fonctionnalités de EXPLAIN 49

4.7.1 Nouvelle option IO 49

4.7.2 Ajout du nombre d'octets pour les FPI 52

4.7.3 Ajout des estimations de cache et de recherche 52

5/ Optimiseur **55**

5.1 JIT désactivé par défaut 56

5.2 Statistiques sur les colonnes virtuelles 57

5.3 Optimisation sur NOT IN 62

5.4 Optimisation sur IS NOT DISTINCT FROM 64

6/ Supervision **67**
6.1 Nouvelle vue pg_stat_lock . . . 68
6.2 Nouvelle vue pg_stat_recovery . . . 70
6.3 Autres nouvelles colonnes des vues statistiques . . . 72
6.4 Nouvelle vue pg_stat_autovacuum_scores . . . 74
6.5 Nouvelles colonnes pour pg_stat_progress_vacuum et pg_stat_progress_analyze . . . 78
6.6 Nouveautés de pg_stat_statements . . . 79
6.6.1 Requêtes FETCH . . . 79
6.6.2 Décompte des plans génériques et personnalisés . . . 80
6.7 Configuration de log_min_messages par processus . . . 82
6.8 Nouveau paramètre log_autoanalyze_min_duration . . . 84
6.9 Activation de log_lock_waits par défaut . . . 85
6.10 VACUUM VERBOSE : ajout du nombre d'octets des FPI . . . 86
6.11 VACUUM VERBOSE : ajout de la consommation mémoire et de la parallélisation . . . 88
6.11.1 Utilisation de la mémoire . . . 88
6.11.2 Utilisation de la parallélisation . . . 91

7/ Réplication **93**
7.1 Réplication logique des séquences . . . 94
7.2 Autres nouveautés en réplication logique . . . 99
7.2.1 Exclusion de table . . . 99
7.2.2 Utilisation d'un SERVER comme source de publication . . . 99
7.2.3 Changement automatique de wal_level . . . 100
7.3 Nouvelle commande : WAIT FOR . . . 102

8/ Divers **105**
8.1 Nouveautés pour les modules contrib . . . 106
8.1.1 pg_buffercache . . . 106
8.1.2 file_fdw . . . 107
8.1.3 auto_explain . . . 107
8.1.4 oid2name . . . 107

9/ Ce qui ne sera pas là **109**
9.1 La prochaine fois peut-être ? . . . 110

10/ Questions **113**

Notes **115**

Notes **117**

Notes **119**

Nos autres publications **121**

Formations 122

Livres blancs 123

Téléchargement gratuit 124

11/ DALIBO, L'Expertise PostgreSQL **125**

1/ Introduction



- Développement depuis juin 2025
- Actuellement en 19 RC 1, RC 2 le 12 novembre
- Release prévue le 19 novembre 2026
- Des centaines de contributeurs

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

2/ Utilisation

2.1 INSERT … ON CONFLICT … DO SELECT



```
INSERT INTO employes (matricule, nom)
  VALUES ('P001', 'Guillaume'),
         ('P002', 'Christophe')
  ON CONFLICT (matricule) DO SELECT
  RETURNING *;
```

- Avant : `DO NOTHING`, `DO UPDATE`
- v19 : `DO SELECT` (clause `RETURNING` requise)
- Permet de récupérer les identifiants
  - des lignes insérées
  - et des lignes déjà présentes

Cette amélioration de l'instruction `INSERT … ON CONFLICT …` permet de récupérer le contenu des lignes insérées, mais aussi des lignes non insérées (car déjà présentes).

Commençons par créer une table et lui ajouter une ligne :

```
CREATE TABLE employes (
  id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  matricule text UNIQUE NOT NULL,
  nom text);

INSERT INTO employes (matricule, nom)
  VALUES ('P001', 'Guillaume')
  RETURNING * ;  -- affiche les lignes insérées
```

```
 id | matricule |    nom
----+-----------+-----------
  1 | P001      | Guillaume
(1 ligne)
```

Maintenant, tentons d'insérer le même matricule :

```
INSERT INTO employes (matricule, nom)
  VALUES ('P001', 'Guillaume')
  RETURNING * ;
```

Du fait de la présence d'une contrainte d'unicité sur la colonne `matricule` et que la valeur `P001` est déjà présente, une erreur est renvoyée :

```
ERROR:  duplicate key value violates unique constraint "employes_matricule_key"
DETAIL:  Key (matricule)=(P001) already exists.
```

Maintenant, faisons la même chose, mais en utilisant la clause `ON CONFLICT` :

```
INSERT INTO employes (matricule, nom)
  VALUES ('P001', 'Guillaume')
  ON CONFLICT (matricule) DO SELECT
  RETURNING *;
```

La valeur `P001` existe déjà, il y a donc conflit. De ce fait, il exécute le `SELECT` en utilisant les colonnes indiquées par la clause `RETURNING`, ce qui nous donne :

```
 id | matricule |    nom
----+-----------+-----------
  1 | P001      | Guillaume
(1 row)
```

Noter qu'aucune erreur n'est levée, comme pour `ON CONFLICT DO NOTHING`.

Maintenant, faisons-le pour deux lignes :

```
INSERT INTO employes (matricule, nom)
  VALUES ('P001', 'Guillaume'),
         ('P002', 'Christophe')
  ON CONFLICT (matricule) DO SELECT
  RETURNING *;
```

La première ligne existe déjà, donc il ne fait que le `SELECT` pour celle-ci. La deuxième ligne n'existe pas, il l'insère et la renvoit grâce à la clause `RETURNING`, ce qui nous donne ce résultat :

```
 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
(2 rows)
```

2.2 IGNORE NULLS/RESPECT NULLS POUR CERTAINES FONCTIONS DE FENÊTRAGE



- Ajout de la clause `IGNORE NULLS/RESPECT NULLS`
- Gérée par quelques fonctions de fenêtrage
  - `lead`
  - `lag`
  - `first_value`
  - `last_value`
  - `nth_value`

Il est maintenant possible d'indiquer à une fonction de fenêtrage si elle doit prendre en compte les valeurs NULL. Cela passe par la clause `IGNORE/RESPECT NULLS`. Par défaut, les valeurs NULL sont prises en compte (`RESPECT NULLS`). Cette nouveauté améliore le niveau de support du standard SQL.

Voici un exemple montrant cette fonctionnalité :

```
demo=# CREATE TABLE t1 (c1 integer);
CREATE TABLE
demo=# INSERT INTO t1 SELECT generate_series(1, 10);
INSERT 0 10
demo=# INSERT INTO t1 VALUES (NULL);
INSERT 0 1
demo=# \pset null '<NULL>'
Null display is "<NULL>".
demo=# SELECT c1, last_value(c1) OVER() FROM t1;
   c1   | last_value
--------+------------
      1 |     <NULL>
      2 |     <NULL>
      3 |     <NULL>
      4 |     <NULL>
      5 |     <NULL>
      6 |     <NULL>
      7 |     <NULL>
      8 |     <NULL>
      9 |     <NULL>
     10 |     <NULL>
 <NULL> |     <NULL>
(11 rows)
```

```
demo=# SELECT c1, last_value(c1) IGNORE NULLS OVER() FROM t1;
   c1   | last_value
--------+------------
      1 |         10
      2 |         10
      3 |         10
      4 |         10
      5 |         10
      6 |         10
      7 |         10
      8 |         10
      9 |         10
     10 |         10
 <NULL> |         10
(11 rows)
```

2.3 AJOUT DU FORMAT JSON POUR ‘COPY ... TO’



```
-- JSON
COPY matable TO 'fichier.json' (FORMAT json);
-- Tableau de JSON
COPY matable TO 'fichier.json' (FORMAT json, FORCE_ARRAY);
```

- Autrefois : contournement avec `row_to_json(table)`
- Pas de colonnes générées par défaut
  - contournement :

```
COPY (SELECT …) TO 'fichier.json'
```

L’instruction COPY supporte directement l’export au format JSON (enfin !).

Sur les versions précédentes, il fallait utiliser la fonction `row_to_json( ligne )` pour générer le JSON.

`FORMAT json` est incompatible avec des options comme HEADER, `DELIMITER` et `FORCE QUOTE` destinées aux CSV.

Par contre, il connaît une option intéressante : `FORCE_ARRAY`, qui transforme le JSON final en tableau (PostgreSQL ajoute juste les crochets et des virgules comme délimiteur).

Par défaut, COPY n’exporte pas les colonnes générées (virtuelles ou stockées). Si l’on tient à leur présence dans l’export, il est facile de contourner cette limitation en utilisant un `SELECT` en source du COPY.

Exemple :

```
CREATE TABLE demo (
    id      bigint,
    t       text,
    tabt    text[],
    j       jsonb,
    d       timestamptz,
    n       numeric(10,2),
    pi      float,
    p       boolean,
    gs      bigint GENERATED ALWAYS AS ((id-1)) STORED,
    gv      bigint GENERATED ALWAYS AS ((id-2)) STORED
);
INSERT INTO demo VALUES
```

```
 (1,
 'D''une part d''autre part',
 '{pim,pam, poum}',
 '{"pays":"France","capitale":"Paris","UE":true}'::jsonb,
 now(),
 pi(),
 pi(),
 true
) ;
INSERT INTO demo VALUES
 (9223372036854775806,
 'Ceci contient "des guillemets anglais"',
 null,
 '{"pays":"Allemagne","capitale":"Berlin","UE":true}'::jsonb,
 now()-interval '1 day',
 pi(),
 pi(),
 false
) ;
```

Utilisation de `row_to_json()` :

```
COPY (SELECT row_to_json(demo) FROM demo) TO '/tmp/json18.json';
\! cat /tmp/json18.json
```

```
{"id":1,"t":"D'une part d'autre part","tabt":["pim","pam","poum"],"j":{"UE": true,
 ↪  "pays": "France", "capitale": "Paris"},"d":"2026-07-
 ↪  09T14:53:44.862061+02:00","n":3.14,"pi":3.141592653589793,"p":true,"gs":0,"gv":-
 ↪  1}
{"id":9223372036854775806,"t":"Ceci contient \\"des guillemets
 ↪  anglais\\"","tabt":null,"j":{"UE": true, "pays": "Allemagne", "capitale":
 ↪  "Berlin"},"d":"2026-07-
 ↪  08T14:53:44.864768+02:00","n":3.14,"pi":3.141592653589793,"p":false,"gs":9223372
  036854775805,"gv":9223372036854775804}
```

Il existe même une version `row_to_json(demo, true)` avec des retours chariot, dont le résultat est plus esthétique.

En version 19, on peut simplement demander :

```
COPY demo TO '/tmp/json19.json' (FORMAT json);
\! cat /tmp/json19.json
```

```
{"id":1,"t":"D'une part d'autre part","tabt":["pim","pam","poum"],"j":{"UE": true,
 ↪  "pays": "France", "capitale": "Paris"},"d":"2026-07-
 ↪  09T14:53:44.862061+02:00","n":3.14,"pi":3.141592653589793,"p":true}
{"id":9223372036854775806,"t":"Ceci contient \"des guillemets
 ↪  anglais\"","tabt":null,"j":{"UE": true, "pays": "Allemagne", "capitale":
 ↪  "Berlin"},"d":"2026-07-
 ↪  08T14:53:44.864768+02:00","n":3.14,"pi":3.141592653589793,"p":false}
```

Pour avoir une table de JSON :

```
/* FORCE_ARRAY pour créer un tableau de JSON */
COPY demo TO '/tmp/json19fa.json' (FORMAT json, FORCE_ARRAY);
\! cat /tmp/json19fa.json
```

```
[
 {"id":1,"t":"D'une part d'autre part","tabt":["pim","pam","poum"],"j":{"UE": true,
  ↪  "pays": "France", "capitale": "Paris"},"d":"2026-07-
  ↪  09T14:53:44.862061+02:00","n":3.14,"pi":3.141592653589793,"p":true}
,{"id":9223372036854775806,"t":"Ceci contient \"des guillemets
 ↪  anglais\"","tabt":null,"j":{"UE": true, "pays": "Allemagne", "capitale":
 ↪  "Berlin"},"d":"2026-07-
 ↪  08T14:53:44.864768+02:00","n":3.14,"pi":3.141592653589793,"p":false}
]
```

Pour forcer l'export des colonnes générées :

```
COPY (SELECT *  FROM demo ) TO '/tmp/json19c.json' (FORMAT json);
\! cat /tmp/json19c.json
```

```
{"id":1,"t":"D'une part d'autre part","tabt":["pim","pam","poum"],"j":{"UE": true,
 ↪  "pays": "France", "capitale": "Paris"},"d":"2026-07-
 ↪  09T14:53:44.862061+02:00","n":3.14,"pi":3.141592653589793,"p":true,"gs":0,"gv":-
 ↪  1}
{"id":9223372036854775806,"t":"Ceci contient \"des guillemets
 ↪  anglais\"","tabt":null,"j":{"UE": true, "pays": "Allemagne", "capitale":
 ↪  "Berlin"},"d":"2026-07-
 ↪  08T14:53:44.864768+02:00","n":3.14,"pi":3.141592653589793,"p":false,
  "gs":9223372036854775805,"gv":9223372036854775804}
```

2.4 AUTRES NOUVELLES FONCTIONNALITÉS DE COPY



```
COPY fichier FROM 'x.csv' (FORMAT CSV, DELIMITER ',', HEADER 3, ON_ERROR
↪  SET_NULL);
```

- Ignorer plusieurs lignes d'entête
  - HEADER avec le nombre de lignes à ignorer
- Remplacer des valeurs invalides avec NULL
  - nouvelle valeur SET_NULL pour ON_ERROR
- Traiter les tables partitionnées

La commande COPY FROM accepte une option HEADER. Avant la version 19, cette option avait pour but d'ignorer la première ligne en entrée quand l'entrée commençait par un en-tête. À partir de la version 19, il est possible d'indiquer après cette option le nombre de lignes à ignorer. Par exemple, pour ce fichier :

```
première ligne de commentaire
la ligne suivante contient les entêtes de colonnes
nom,prenom
lelarge,guillaume
courtois,christophe
lobréau,benoît
```

Avec la version 19, on peut ignorer les trois premières lignes pour ne prendre en compte que les lignes de données. Cela se fait ainsi :

```
CREATE TABLE fichier(nom text, prenom text);
COPY fichier FROM '/tmp/fichier.csv' (FORMAT CSV, DELIMITER ',', HEADER 3);
SELECT * FROM fichier;
```

```
   nom    |   prenom
----------+------------
 lelarge  | guillaume
 courtois | christophe
 lobréau  | benoît
(3 rows)
```

Les développeurs de PostgreSQL ont aussi continué sur la gestion des erreurs dans le fichier en entrée. L'option ON_ERROR se voit ajouter une valeur SET_NULL permettant d'accepter une ligne dont certaines colonnes sont en erreur. Dans ce cas, la valeur de ces colonnes est remplacée par la valeur

NULL. Pour l'exemple, nous allons ajouter une colonne d'identifiant en première place. Voici le nouveau fichier en entrée :

```
première ligne de commentaire
la ligne suivante contient les entêtes de colonnes
id,nom,prenom
lelarge,guillaume,1
courtois,christophe,é
lobréau,benoît,3
```

Comme nous pouvons le voir, la cinquième ligne comporte une erreur : un é au lieu d'un entier (un 2 mal saisi). Ajoutons la nouvelle colonne à la table :

```
ALTER TABLE fichier ADD COLUMN id integer;
```

Voyons maintenant comment réagit PostgreSQL sans la nouvelle option :

```
TRUNCATE TABLE fichier ;
COPY FICHIER FROM '/tmp/fichier.csv' (FORMAT CSV, DELIMITER ',', HEADER 3);

ERROR:  invalid input syntax for type integer: "é"
CONTEXT:  COPY fichier, line 5, column id: "é"
```

Logiquement, le fichier n'est pas intégré du fait de l'erreur. Et voici ce qui se passe avec la nouvelle option :

```
COPY fichier FROM '/tmp/fichier.csv' (FORMAT CSV, DELIMITER ',', HEADER 3, ON_ERROR
↪  SET_NULL);

NOTICE:  in 1 row, columns were set to null due to data type incompatibility
COPY 3
```

PostgreSQL prévient que certaines colonnes ont été mises à NULL à cause d'une incompatibilité sur le type de données. Regardons le contenu de la table :

```
demo=# \pset null '<NULL>'
Null display is "<NULL>".
demo=# SELECT * FROM fichier;
   nom    |   prenom   |   id
----------+------------+--------
 lelarge  | guillaume  |      1
 courtois | christophe | <NULL>
 lobréau  | benoît     |      3
(3 rows)
```

Nous retrouvons bien nos trois lignes mais la colonne `id` de la deuxième ligne vaut NULL.

Enfin, dernier changement sur la commande COPY : la variante `COPY TO` est capable de lire les tables partitionnées.

3/ Administration

3.1 AUTOVACUUM : PARALLÉLISATION DU NETTOYAGE DES INDEX



- VACUUM savait déjà paralléliser le nettoyage des index
  - ex : `VACUUM (PARALLEL 4)`
- L'autovacuum sait à présent le faire
- `autovacuum_max_parallel_workers` encore à 0 par défaut
  - suppose `max_parallel_workers` suffisant
- Pour restreindre :

```
ALTER TABLE … SET (autovacuum_parallel_workers = 1) ;
```

Rappels :

Le VACUUM est optimisé pour ne nettoyer que les blocs avec des lignes mortes (grâce à la *visibility map*). Dans beaucoup de cas, le nettoyage des index reste alors la partie la plus longue du VACUUM, car ils sont intégralement parcourus.

Depuis PostgreSQL 13, ce nettoyage d'index peut être parallélisé, par exemple ainsi :

```
VACUUM (PARALLEL 4) ;
```

Quatre nouveaux processus (ou moins) se partagent alors les index à traiter, à condition que les paramètres `max_parallel_workers` et `max_worker_processes` soient plus élevés (leurs valeurs par défaut sont seulement 2 et 8), et que d'autres processus parallèles n'occupent pas tous les slots (y compris ceux associés à des requêtes).

Pour les détails :

- Voir le workshop de la version 13[1]
- Voir la documentation : commande VACUUM[2], et paragraphe Parallel Vacuum[3]

PostgreSQL 19 ouvre cette fonctionnalité à l'autovacuum, mais elle reste désactivée par défaut.

Il faudra donc monter `autovacuum_max_parallel_workers` au-delà de 0, et éventuellement positionner `autovacuum_parallel_workers` sur la table.

```
-- Ceci suppose que max_parallel_workers et max_worker_processes
```

[1] https://dali.bo/workshop13_html#vacuum-nouveaux-workers
[2] https://www.postgresql.org/docs/19/sql-vacuum.html
[3] https://www.postgresql.org/docs/19/routine-vacuuming.html#PARALLEL-VACUUM

```
-- sont déjà supérieurs (redémarrage nécessaire)
--
ALTER SYSTEM SET autovacuum_max_parallel_workers TO 8 ;
-- Pour tracer le vacuum
ALTER SYSTEM SET log_autovacuum_min_duration TO 0;

SELECT pg_reload_conf() ;

CREATE TABLE t1 AS
SELECT i AS c1, i AS c2 FROM generate_series (1,1_000_000) i;
CREATE INDEX ON t1 (c1);
CREATE INDEX ON t1 (c2);
CREATE INDEX ON t1 (c1+c2);
CREATE INDEX ON t1 (c1);
CREATE INDEX ON t1 (c1);
CREATE INDEX ON t1 (c1);
CREATE INDEX ON t1 (c1);
CREATE INDEX ON t1 (c1);
VACUUM t1 ;

-- 4 process au plus pour la table
ALTER TABLE t1 SET (autovacuum_parallel_workers = 4) ;
-- Effacer plus de 20% d'une table avec de nombreux index
DELETE FROM t1 WHERE random()> 0.75 ;
```

Si `log_autovacuum_min_duration` est assez bas, on verra ce qui suit dans `postgresql.log`. La ligne `parallel workers` n'apparaît que si le parallélisme a été utilisé.

```
LOG:  automatic vacuum of table "pgbench.public.t1": index scans: 1
pages: 0 removed, 4456 remain, 4456 scanned (100.00% of total), 0 eagerly scanned
tuples: 249696 removed, 750304 remain, 0 are dead but not yet removable
removable cutoff: 16254306, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 4425 pages set all-visible, 0 pages set all-frozen (0 were
 ↪  all-visible)
index scan needed: 4425 pages from table (99.30% of total) had 249696 dead item
 ↪  identifiers removed
parallel workers: index vacuum: 4 planned, 4 launched in total
index "t1_c1_idx": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
 ↪  reusable
index "t1_c2_idx": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
 ↪  reusable
index "t1_c1_idx1": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
 ↪  reusable
index "t1_c1_idx2": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
 ↪  reusable
index "t1_c1_idx3": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
 ↪  reusable
```

```
index "t1_c1_idx4": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
↪  reusable
index "t1_c1_idx5": pages: 2745 in total, 0 newly deleted, 0 currently deleted, 0
↪  reusable
avg read rate: 30.191 MB/s, avg write rate: 30.077 MB/s
buffer usage: 13613 hits, 19208 reads, 19136 dirtied
WAL usage: 27993 records, 5 full page images, 6007660 bytes, 40960 full page image
↪  bytes, 0 buffers full
memory usage: dead item storage 2.00 MB accumulated across 1 reset (limit 64.00 MB
↪  each)
system usage: CPU: user: 0.36 s, system: 0.08 s, elapsed: 4.97 s
```

L'autovacuum ne sera pas forcément beaucoup plus rapide si son bridage n'est pas levé (pour cela, baisser `autovacuum_vacuum_cost_delay` ou monter `autovacuum_vacuum_cost_limit`, et/ou monter `vacuum_buffer_usage_limit`, ou les paramètres associés de la commande utilisée). Sinon, les I/O deviendront le facteur limitant.



`autovacuum_max_parallel_workers` est une limite par worker de l'autovacuum, et par défaut il peut y en avoir trois simultanés (paramètre `autovacuum_max_workers`). Attention donc si vous montez cette limite et que vous avez de nombreuses tables avec beaucoup d'index.

3.2 AUTOVACUUM : TRI DES TABLES PAR SCORE



- Auparavant :
  - tri des bases par urgence
  - mais pas de tri des tables
- Système de score
- 5 nouveaux paramètres :
  - `autovacuum_freeze_score_weight`
  - `autovacuum_multixact_freeze_score_weight`
  - `autovacuum_vacuum_score_weight`
  - `autovacuum_vacuum_insert_score_weight`
  - `autovacuum_analyze_score_weight`

L'autovacuum est constitué de deux processus :

- l'*autovacuum launcher* qui récupère la liste des bases, trie les bases par ordre d'urgence de traitement, et lance un *autovacuum worker* sur une base ;
- l'*autovacuum worker* qui, une fois connecté à la base à traiter, récupère la liste des tables, sélectionne celles à traiter, et les traite les unes après les autres sans ordre prédéfini, sans priorité de traitement.

Le fait que l'ordre des tables soit aléatoire est problématique. Il peut être intéressant de traiter telle ou telle table en priorité. La version 19 améliore donc ce système en ajoutant le calcul d'un score pour le traitement d'une table, suivant cinq critères :

- le risque d'une réutilisation des identifiants de transactions ;
- le risque d'une réutilisation des identifiants MultiXact ;
- le besoin d'un *VACUUM* à cause des lignes mortes ;
- le besoin d'un *VACUUM* à cause des lignes insérées ;
- le risque de statistiques trop anciennes sur les données.

Un score est calculé pour chaque critère, et il est possible d'influer sur l'importance d'un score par rapport à un autre avec cinq nouveaux paramètres de poids :

- `autovacuum_freeze_score_weight`
- `autovacuum_multixact_freeze_score_weight`

- autovacuum_vacuum_score_weight
- autovacuum_vacuum_insert_score_weight
- autovacuum_analyze_score_weight

Ces paramètres ont tous une valeur de 1, ce qui fait qu'aucun critère n'est privilégié. Cependant, si le score sur le ANALYZE nous semble à prioriser, il faudrait augmenter le paramètre relatif, autovacuum_analyze_score_weight dans ce cas.

Si vous voulez revenir au comportement précédent (à savoir, sans score, sans priorisation), il faut configurer les cinq paramètres à 0.

Nous verrons plus loin la vue pg_stat_autovacuum_scores.

3.3 REPACK / REPACK USING INDEX



```
-- Avant
VACUUM FULL nomtable ;
CLUSTER nomtable USING nomindex ;
-- v19
REPACK nomtable ;
REPACK nomtable USING INDEX nomindex ;
```

- Renommage plus cohérent
- Même fonctionnement & impact
  - … du moins sans CONCURRENTLY
- Privilège : MAINTAIN
- Vue : pg_stat_progress_repack

PostgreSQL 19 supprime un souci de compréhension courant chez les débutants : la commande VACUUM FULL reconstruisait une table, mais n'avait aucun rapport avec le VACUUM simple qui libère les lignes mortes (entre autres) sans forcément réduire la taille de la table sur le disque.

Sans option, REPACK reprend le principe de VACUUM FULL avec les mêmes soucis :

- la table peut temporairement occuper jusqu'au double de la table d'origine (la nouvelle version de la table doit être reconstruite et réindexée à côté de l'ancienne avant de supprimer celle-ci) ;
- l'opération est lourde sur tous les plans : I/O, durée, journaux, voire CPU ;
- un verrou ACCESS EXCLUSIVE bloque totalement l'utilisation de la table, en lecture comme en écriture, pendant toute l'opération (voir REPACK (CONCURRENTLY) plus loin).

Ces deux commandes sont donc équivalentes :

```
-- jusque v18
VACUUM (FULL) pgbench_accounts ;
-- en v19
REPACK pgbench_accounts ;
```

Les vues matérialisées peuvent être aussi reconstruites comme les tables.

REPACK seul reconstruit toute la base en cours.

REPACK remplace aussi la commande CLUSTER, moins connue, qui reconstruit une table également, mais en la triant selon un index préexistant. Cela peut être intéressant pour améliorer la corrélation des données, et réduire les I/O des requêtes récupérant de nombreuses lignes via cet index. Cette

réorganisation n'est pas pérenne et les lignes peuvent se mélanger de nouveau ensuite en fonction de la vie de la table.

Cette opération est bien sûr encore plus lourde qu'un `VACUUM FULL` ou `REPACK` simple en raison du tri de la table.

Ces commandes sont équivalentes :

```
-- jusque v18
CLUSTER pgbench_accounts USING pgbench_accounts_pkey ;
-- en v19
REPACK pgbench_accounts  USING INDEX pgbench_accounts_pkey ;
```

Comme c'était le cas avec `CLUSTER`, si la table a déjà été soumise à un ordre `REPACK … USING INDEX …`, il n'y a plus besoin de préciser l'index au prochain `REPACK … USING INDEX`. En effet, PostgreSQL mémorise le dernier index de tri en tant qu'index `CLUSTER` :

```
\dt pgbench_accounts
…
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid) CLUSTER
```

Pour effacer cette information :

```
-- Annulation d'une commande `REPACK … USING INDEX …` antérieure
ALTER TABLE pgbench_accounts SET WITHOUT CLUSTER ;
-- Tri sans index, dans l'ordre physique
REPACK (VERBOSE) pgbench_accounts USING INDEX ;

ERROR:  there is no previously clustered index for table "pgbench_accounts"

-- Tri sans index, dans l'ordre physique
REPACK (VERBOSE) pgbench_accounts USING INDEX ;

INFO:  repacking "public.pgbench_accounts" in physical order
INFO:  "public.pgbench_accounts": found 570945 removable, 3000000 nonremovable row
 ↪  versions in 58541 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 1.64 s, system: 1.42 s, elapsed: 3.48 s.
```

L'index peut aussi être précisé dès la définition de la table avec `CLUSTER ON` :

```
-- Définition de l'index qui servira de tri
ALTER TABLE pgbench_accounts CLUSTER ON pgbench_accounts_pkey ;
-- Reconstruction dans l'ordre de l'index
REPACK (VERBOSE) pgbench_accounts USING INDEX;

INFO:  repacking "public.pgbench_accounts" using index scan on
 ↪  "pgbench_accounts_pkey"
```

```
INFO:  "public.pgbench_accounts": found 0 removable, 3000000 nonremovable row
 ↪  versions in 49181 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 1.65 s, system: 1.01 s, elapsed: 2.91 s.
```

L'index doit avoir les méthodes d'accès adéquates, ce sera donc presque toujours un B-tree, non partiel, éventuellement fonctionnel ; mais un GiST avec l'opérateur `btree_gist` est aussi utilisable. Attention : l'ordre final dans la table sera bien issu de l'ordre des valeurs stockées dans l'index, et pas de l'ordre des valeurs de la colonne. Ce peut être surprenant mais le but est bien de grouper des lignes voisines dans l'index.

Pour les tables partitionnées, l'index utilisé doit être précisé partition par partition, et n'est pas obligatoirement le même.

Si PostgreSQL considère qu'il vaut mieux lire la table et la trier, la première ligne d'un `REPACK ... USING INDEX` sera plutôt :

```
INFO:  repacking "public.pgbench_accounts" using sequential scan and sort
```

Il est donc conseillé de toujours ajouter `VERBOSE` pour voir la manière dont la table est réécrite.

Si `maintenance_work_mem` est insuffisant, les fichiers temporaires de tri (de la table ou pour la réindexation) apparaîtront dans le tablespace de tri (`./base/pgsql_tmp/` par défaut) et les noms de fichier contiendront le PID du processus concerné (par exemple `pgsql_tmp882137.13`).

Enfin, il est possible (et conseillé) de lancer un `ANALYZE` dans la foulée de la reconstruction :

```
REPACK (VERBOSE, ANALYZE) pgbench_accounts USING INDEX pgbench_accounts_pkey ;
```

```
INFO:  repacking "public.pgbench_accounts" using index scan on
 ↪  "pgbench_accounts_pkey"
INFO:  "public.pgbench_accounts": found 29 removable, 3000000 nonremovable row
 ↪  versions in 68848 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 3.19 s, system: 1.50 s, elapsed: 5.21 s.
INFO:  analyzing "public.pgbench_accounts"
INFO:  "pgbench_accounts": scanned 30000 of 49181 pages, containing 1829959 live
 ↪  rows and 0 dead rows; 30000 rows in sample, 2999974 estimated total rows
INFO:  finished analyzing table "postgres.public.pgbench_accounts"
avg read rate: 1010.338 MB/s, avg write rate: 0.101 MB/s
buffer usage: 34 hits, 30003 reads, 3 dirtied
WAL usage: 7 records, 3 full page images, 17962 bytes, 16440 full page image bytes, 0
 ↪  buffers full
system usage: CPU: user: 0.20 s, system: 0.02 s, elapsed: 0.23 s
```



Les commandes `VACUUM FULL` et `CLUSTER` sont toujours utilisables, pour la compatibilité avec les scripts existants.

En ce qui concerne les droits : pour lancer REPACK, il faut être propriétaire ou posséder le privilège MAINTAIN sur la table, ou faire partie du rôle pg_maintain.

Enfin, la vue pg_stat_progress_repack permet de suivre les REPACK en cours (il s'agit de pg_stat_progress_cluster renommée).

REPACK nomtable (sans les options ANALYZE ni CONCURRENTLY) peut être utilisé au sein d'une transaction.

3.4 REPACK (CONCURRENTLY)



```
REPACK (CONCURRENTLY) nomtable ;
```

- REPACK sans arrêt de production !
- consomme toujours 2× la volumétrie
- Utilise le décodage logique
  - `max_repack_replication_slots` (5)
  - `max_slot_wal_keep_size` à vérifier
- Restrictions
  - clé primaire (ou unique)
  - pas UNLOGGED
  - partition par partition
  - `wal_level` >= `replica`
  - éviter le DDL simultané

L'auteur de pg_squeeze[4] a porté son travail dans le cœur de PostgreSQL. REPACK possède donc une option CONCURRENTLY qui ne pose plus de verrou ACCESS EXCLUSIVE pendant toute l'opération, mais juste dans sa dernière phase. L'opération pourra donc être faite à chaud beaucoup plus souvent.

3.4.1 Fonctionnement

REPACK (CONCURRENTLY) utilise l'infrastructure de décodage logique pour capturer les lignes modifiées après le début de la copie de la table (*change data capture*).

Un *background worker* est lancé le temps de l'opération (visible dans `pg_stat_activity` avec `backend_type` à *REPACK decoding worker*). Il stocke les lignes modifiées par les requêtes pendant l'opération dans un fichier temporaire (répertoire `./base/pgsql_tmp/` par défaut) dans un sous-répertoire portant le PID de la session (par exemple `pgsql_tmp882137.1.fileset`).

Après copie des lignes vivantes des fichiers d'origine de la table, les index sont reconstruits. Les lignes de la table temporaire sont réinjectées dans la nouvelle table.

[4] https://github.com/cybertec-postgresql/pg_squeeze/blob/master/README.md

Enfin les tables système sont mises à jour pour pointer vers les nouveaux fichiers de tables et d'index. Le verrou ACCESS EXCLUSIVE reste nécessaire pendant cette dernière phase, mais pour une durée normalement très courte.

Les différentes phases qui peuvent apparaître dans pg_stat_progress_repack sont notamment : seq scanning heap, sorting tuples, writing new heap, rebuilding index, catch-up



Une session applicative pose très souvent un verrou sur la table (UPDATE, DELETE…, mais aussi SELECT ou même pg_dump) qui peut bloquer la dernière phase, et donc provoquer un blocage général des accès à la table. Le démarrage du *REPACK start_repack_decoding_worker* doit aussi attendre la fin des transactions démarrées avant lui. Il faut donc être sûr qu'aucune session ne va procéder à des transactions ou requêtes longues pendant le REPACK. De toute manière, vu l'impact d'un REPACK sur les I/O , il reste conseillé de procéder à l'opération dans la période la plus calme possible.

Un nouveau paramètre apparaît : max_repack_replication_slots. Il s'agit du nombre de slots de réplication logique utilisable par REPACK (CONCURRENTLY). Le défaut est 5, permettant d'avoir cinq opérations simultanées. Si le nombre de slots est insuffisant, ces messages apparaîtront :

```
ERROR:  out of background worker slots
HINT:  You might need to increase "max_worker_processes".

ERROR:  all replication slots are in use
HINT:  Free one or increase "max_repack_replication_slots".
```

Augmenter max_repack_replication_slots implique de redémarrer.

3.4.2 Limites

L'option CONCURRENTLY allonge forcément un peu l'opération : elle doit attendre que des verrous soient levés, et la phase catch-up n'est pas instantanée,

REPACK (CONCURRENTLY) résout en grande partie le souci lié à la durée du verrou. Mais le souci de la place nécessaire à la reconstruction reste présent (la place consommée est même un peu supérieure à cause du fichier temporaire des lignes modifiées, mais il peut être déporté vers un tablespace temporaire).

En raison du mécanisme sous-jacent, la table doit posséder une clé de réplication (REPLICA IDENTITY) avec index. En pratique, la clé primaire conviendra, et il est de toute façon conseillé d'en met-

tre une sur chaque table. Un index unique non partiel sur une colonne non nulle fonctionnera aussi si REPLICA IDENTITY est définie :

```
ALTER TABLE demo REPLICA IDENTITY USING INDEX demo_uq ;
```

Un REPLICA IDENTITY FULL n'est pas utilisable.

Enfin, une conséquence de l'utilisation d'un slot de réplication est que REPACK (CONCURRENTLY) est sensible à max_slot_wal_keep_size. Ce paramètre est souvent placé pour protéger le primaire d'un secondaire trop en retard. Si des transactions longues ou un verrou gênent la fin du REPACK, il peut échouer avec ce message :

```
ERROR:  terminating background worker "REPACK decoding worker" due to administrator
↪  command
CONTEXTE : waiting for ShareLock on transaction 16928891
REPACK decoding worker
```

avec dans postgresql.log :

```
LOG:  terminating process 3914256 to release replication slot "pg_repack_3914256"
DETAIL:  The slot's restart_lsn 3E/F399ADA8 exceeds the limit by 359027288 bytes.
HINT:  You might need to increase "max_slot_wal_keep_size".
```

Un REPACK (CONCURRENTLY) n'est pas possible pour des tables UNLOGGED, car elles ne sont pas journalisées. Vu les cas d'utilisation de ces tables, c'est rarement utile.

```
REPACK (CONCURRENTLY, VERBOSE) demo ;

ERROR:  cannot execute REPACK (CONCURRENTLY) on relation "demo"
ASTUCE : REPACK (CONCURRENTLY) is only allowed for permanent relations.
```

Pour une table partitionnée, il faudra procéder partition par partition :

```
REPACK (CONCURRENTLY, VERBOSE)  pgbench_accounts;

ERROR:  REPACK (CONCURRENTLY) is not supported for partitioned tables
ASTUCE : Consider running the command on individual partitions.

REPACK (CONCURRENTLY, VERBOSE)  pgbench_accounts_3;

INFO:  repacking "public.pgbench_accounts_3" in physical order
…
```

Là non plus, ce ne devrait pas être gênant puisque le but premier du partitionnement est de réduire la volumétrie à manier lors des maintenances.

Les tables systèmes ne sont pas gérées par REPACK (CONCURRENTLY), mais peuvent être reconstruites avec REPACK.

`REPACK (CONCURRENTLY)` ne peut être exécuté dans une transaction.

`REPACK (CONCURRENTLY)` utilise l'infrastructure de décodage logique, qui impose `wal_level = logical`. Mais PostgreSQL 19 sait passer automatiquement `wal_level` de `replica` à `logical`, et juste le temps de l'opération (la valeur de `effective_wal_level` passe à `logical` transitoirement). Par contre, avec `wal_level = minimal`, `REPACK (CONCURRENTLY)` ne fonctionne plus :

```
ERROR:  cannot execute REPACK (CONCURRENTLY) in this configuration
DÉTAIL : REPACK (CONCURRENTLY) requires "wal_level" to be set to "replica" or higher.
```

Là encore, ce devrait être rarement gênant, car `replica` est le défaut depuis PostgreSQL 10.

Un autre souci théorique est que la commande n'est pas *MVCC-safe*, c'est-à-dire que la table pourrait apparaître vide à une autre transaction. Cette transaction doit avoir pris un *snapshot* avant le début du `REPACK (CONCURRENTLY)`, mais en pratique elle prend aussi un verrou suffisant pour bloquer le verrou `ACCESS EXCLUSIVE` le temps de finir son opération.

Pour les détails : https://www.postgresql.org/docs/19/mvcc-caveats.html

3.5 AMÉLIORATIONS DE PSQL



```
-- Prompt : primary/standby  et search_path
\set PROMPT1 '%i %S > '
-- Booléens plus lisibles
\pset display_true '✓'
\pset display_false '×'
```

- Affichage du commentaire pour les publications, souscriptions et statistiques étendues

Le prompt dispose de deux nouveaux caractères joker. Le premier est %S, qui renvoie le `search_path` en cours (le serveur doit être de version 18 au moins). En voici un exemple d'utilisation :

```
postgres=# \set PROMPT1 'mon search_path "%S" > '
mon search_path ""$user", public" > set search_path to s1,public;
SET
mon search_path "s1, public" > set search_path to s1;
SET
mon search_path "s1" > reset search_path;
RESET
mon search_path ""$user", public" >
```

Le second est `%i` qui indique si le serveur est un primaire (`primary`) ou un secondaire (`secondary`). Le serveur doit être au moins en version 14. Par exemple :

```
postgres=# \set PROMPT1 '%i | '

primary | select 1;
 ?column?
----------
        1
(1 row)
```

Il devient aussi possible de modifier la façon dont les valeurs booléennes sont affichées. Pour cela, deux nouvelles variables sont disponibles via la méta-commande `\pset` : `display_true` et `display_false`. Voici un nouvel exemple.

Commençons par créer une table :

```
CREATE TABLE t1 (c1 integer, c2 boolean);
INSERT INTO t1 VALUES (1,true), (2,true), (3, false), (4,true), (5,false);
```

Si on interroge son contenu avec un simple SELECT, par défaut, nous obtenons cet affichage :

```
 c1 | c2
----+----
  1 | t
  2 | t
  3 | f
  4 | t
  5 | f
(5 rows)
```

Maintenant, configurons les deux nouvelles variables :

```
postgres=# \pset display_true '✓'
Boolean true display is "✓".
postgres=# \pset display_false '×'
Boolean false display is "×".
```

Ré-exécutons le SELECT, voici son résultat :

```
 c1 | c2
----+----
  1 | ✓
  2 | ✓
  3 | ×
  4 | ✓
  5 | ×
(5 rows)
```

Cela peut paraître anecdotique, mais cela permet de plus rapidement trouver les colonnes à `true` ou à `false` quand elles sont nombreuses. Prenons comme exemple cette requête sur la vue `pg_roles` :

```
SELECT rolname, rolsuper, rolinherit, rolcanlogin FROM pg_roles;
```

En voici le résultat. Saurez-vous trouver au premier coup d'œil les rôles superuser ?

```
           rolname            | rolsuper | rolinherit | rolcanlogin
------------------------------+----------+------------+-------------
 pg_database_owner            | f        | t          | f
 pg_read_all_data             | f        | t          | f
 pg_write_all_data            | f        | t          | f
 pg_monitor                   | f        | t          | f
 pg_read_all_settings         | f        | t          | f
 pg_read_all_stats            | f        | t          | f
 pg_stat_scan_tables          | f        | t          | f
 pg_read_server_files         | f        | t          | f
 pg_write_server_files        | f        | t          | f
 pg_execute_server_program    | f        | t          | f
```

```
 pg_signal_backend           | f        | t          | f
 pg_checkpoint               | f        | t          | f
 pg_maintain                 | f        | t          | f
 pg_use_reserved_connections | f        | t          | f
 pg_create_subscription      | f        | t          | f
 pg_signal_autovacuum_worker | f        | t          | f
 a2c1                        | f        | t          | t
 adminbdd                    | t        | t          | t
 grafana                     | f        | t          | t
 guillaume                   | f        | t          | t
 postgres                    | t        | t          | t
 scp_testpv                  | f        | t          | t
 toto                        | f        | t          | t
 u1                          | f        | t          | t
(24 rows)
```

Mais avec cette nouvelle fonctionnalité, c'est beaucoup plus rapide :

```
           rolname           | rolsuper | rolinherit | rolcanlogin
-----------------------------+----------+------------+-------------
 pg_database_owner           | ×        | ✓          | ×
 pg_read_all_data            | ×        | ✓          | ×
 pg_write_all_data           | ×        | ✓          | ×
 pg_monitor                  | ×        | ✓          | ×
 pg_read_all_settings        | ×        | ✓          | ×
 pg_read_all_stats           | ×        | ✓          | ×
 pg_stat_scan_tables         | ×        | ✓          | ×
 pg_read_server_files        | ×        | ✓          | ×
 pg_write_server_files       | ×        | ✓          | ×
 pg_execute_server_program   | ×        | ✓          | ×
 pg_signal_backend           | ×        | ✓          | ×
 pg_checkpoint               | ×        | ✓          | ×
 pg_maintain                 | ×        | ✓          | ×
 pg_use_reserved_connections | ×        | ✓          | ×
 pg_create_subscription      | ×        | ✓          | ×
 pg_signal_autovacuum_worker | ×        | ✓          | ×
 a2c1                        | ×        | ✓          | ✓
 adminbdd                    | ✓        | ✓          | ✓
 grafana                     | ×        | ✓          | ✓
 guillaume                   | ×        | ✓          | ✓
 postgres                    | ✓        | ✓          | ✓
 scp_testpv                  | ×        | ✓          | ✓
 toto                        | ×        | ✓          | ✓
 u1                          | ×        | ✓          | ✓
(24 rows)
```



Avec un `psql` client de version 19, ces nouveautés fonctionnent aussi souvent sur des serveurs de versions antérieures.

3.6 AMÉLIORATIONS DES AUTRES OUTILS



- `vacuumdb`
  - `--dry-run`
  - `--analyze-only` et `--analyze-in-stages` analysent les tables partitionnées
- `pg_verifybackup` et `pg_waldump`
  - peuvent lire les fichiers WAL compris dans des archives tar
- `pgbench`
  - nouvelle option `--continue-on-error`
- `pg_upgrade`
  - meilleure gestion des *large objects*, meilleures performances
  - traite les tablespaces stockés dans le répertoire principal

Les différents outils de la distribution PostgreSQL ont bénéficié eux aussi d'attentions. Parmi les plus marquants, notons l'option `--dry-run` de `vacuumdb` qui permet de savoir ce qu'il compte exécuter sur la base. Par exemple :

```
postgres> vacuumdb --dry-run demo

vacuumdb: executing in dry-run mode
vacuumdb: detail: No commands will be sent to the server.
vacuumdb: vacuuming database "demo"
VACUUM (SKIP_DATABASE_STATS) public.t1;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_collation;
[...]
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_foreign_server;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_replication_origin;
VACUUM (ONLY_DATABASE_STATS);
```

Notons aussi que `pg_upgrade` peut maintenant traiter des instances dont les tablespaces ont été placés dans le répertoire principal des données. Auparavant, une telle configuration renvoyait une erreur et il était nécessaire de déplacer les tablespaces avant de mettre à jour.

4/ Performances

4.1 LA QUESTION DES HINTS



- Forte opposition à leur intégration
- Beaucoup d'arguments contre
- Mais
  - leur absence est problématique dans certains cas
  - les enable_* sont utilisés alors qu'ils ne sont à l'origine qu'un outil de débogage
  - les astuces style OFFSET 0 sont des hints cachés
- Il faut trouver un meilleur moyen

Il y a toujours eu une forte opposition de la part des développeurs de PostgreSQL à l'intégration des « hints » dans PostgreSQL. Même certains DBA Oracle critiquent les hints. Parmi les arguments contre leur utilisation, notons :

- leur maintenance est très complexe ;
- les mises à jour du moteur de bases de données nécessitent de les revoir intégralement ;
- ils découragent fortement une analyse en profondeur du problème ;
- ils ralentissent des améliorations sur le planificateur parce que les utilisateurs ne remontent plus ce type de problème aux développeurs.

Toutes ces raisons ont influé pour que le projet se prononce contre leur intégration.

Cependant, au fil des années, nous nous apercevons que nous avons quand même des hints. Ils sont un peu différents dans leur utilisation (ce sont plutôt des paramètres) comme dans leur efficacité. Les paramètres `enable_*` sont parfois utilisés pour contraindre le planificateur à utiliser d'autres types de nœuds. Ils n'ont pas été faits pour cela, et ce contournement de leur utilisation peut être très négatif. D'autres astuces vont aussi contraindre le planificateur, comme l'utilisation du `OFFSET 0` ou des CTE matérialisées.

De plus, dans certains cas, aucune réponse simple n'est apportée à une requête lente. Et dans le cas d'un problème dans le planificateur (optimisation absente), il est difficile de soutenir qu'il faudra attendre plusieurs mois, voire un an, avant la sortie d'une version corrigeant ce problème.

Il s'est fait de plus en plus évident qu'un système de proposition d'un meilleur plan devrait être disponible.

4.2 QUE FAIT LA CONCURRENCE ?



- Hints dans la requête
  - problématique car la requête n'est pas modifiable facilement
- Profiles
  - hints dans une table de référence

Parmi les concurrents de PostgreSQL, nous voyons plusieurs réponses apportées à cette demande.

La première réponse fut les hints proposées par Oracle, à intégrer dans une requête. Cette solution n'est pas satisfaisante. Si le hint est problématique, il faut modifier la requête alors que les utilisateurs n'ont pas forcément cette possibilité (une application fournie par un éditeur par exemple). C'est justement cette réponse que les développeurs de PostgreSQL voulaient éviter.

Une autre réponse est venue après, pour des moteurs comme Oracle et SQL Server. Il s'agit de la possibilité d'enregistrer des hints, voire des plans, à utiliser pour certaines requêtes. Les requêtes se voient attribuer un identifiant et l'optimiseur va chercher dans une table de référence si des hints ont été préconisées pour un identifiant particulier. C'est cette solution que les développeurs de PostgreSQL ont choisie.

4.3 QUE FAIT DONC POSTGRESQL V19 ?



- Module pg_plan_advice
  - pas de contrainte sur le planificateur
  - juste un conseil pour prioriser certains types de noeud
- Module pg_stash_plan
  - enregistrement d'un advice pour un queryid
  - exécution automatique de cet advice

Le système se base sur deux modules : `pg_plan_advice` qui permet de récupérer et de configurer un *advice* pour le planificateur, et `pg_stash_advice` qui permet de stocker un *advice* pour une requête précise dans une table de référence, table consultée par le planificateur.

Dans tous les cas, il s'agit de conseiller le planificateur, pas de le contraindre. Nous aurons beau demander un parcours d'index, il n'aura pas lieu si l'index n'est pas utilisable. Il s'agit donc uniquement de conseils.

4.4 MODULE PG_PLAN_ADVICE



- Librairie
  - donc à charger avec LOAD ou `*_preload_libraries`
- Option PLAN_ADVICE pour EXPLAIN
- Paramètres
  - `pg_plan_advice.advice`
- Balises d'advice
  - parcours (SEQ_SCAN, INDEX_SCAN, INDEX_ONLY_SCAN, …)
  - jointures (ordre avec JOIN_ORDER, méthode avec MERGE_JOIN_PLAIN, MERGE_JOIN_MATERIALIZE, NESTED_LOOP_PLAIN, HASH_JOIN, etc)
  - parallélisation (GATHER, GATHER_MERGE, NO_GATHER)
  - etc

Le module `pg_plan_advice` est une librairie. Cette dernière doit être chargée pour être fonctionnelle. Cela se fait dans une session en utilisant l'instruction LOAD. Cela se fait de façon plus globale ou plus automatiquement en utilisant les paramètres `local_preload_libraries`, `session_preload_libraries` et `shared_preload_libraries`. Cette dernière sera certainement utilisée majoritairement mais, pour l'exemple, nous en resterons à un LOAD.

Créons une table avec quelques données :

```
CREATE TABLE t1(c1 integer PRIMARY KEY, c2 text);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1_000_000) i;
ANALYZE t1;
```

Regardons le plan d'exécution avec cette requête :

```
EXPLAIN (COSTS OFF) SELECT * FROM t1 WHERE c1=50;
```

En voici le résultat :

```
                          QUERY PLAN
---------------------------------------------------------------------
 Index Scan using t1_pkey on t1
   Index Cond: (c1 = 50)
(2 rows)
```

Rien de bien nouveau. Le planificateur estime récupérer une ligne (et il a raison). De ce fait, l'utilisation de l'index nous sera bénéfique en durée d'exécution. Nous avons donc un *index scan*.

Maintenant, chargeons le module `pg_plan_advice` :

```
LOAD 'pg_plan_advice';
```

L'option `plan_advice` de la commande `EXPLAIN` va nous donner quelques informations supplémentaires. Très exactement, elle va nous donner l'*advice* du plan généré :

```
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=50;
```

```
           QUERY PLAN
---------------------------------
 Index Scan using t1_pkey on t1
   Index Cond: (c1 = 50)
 Generated Plan Advice:
   INDEX_SCAN(t1 public.t1_pkey)
   NO_GATHER(t1)
(5 rows)
```

L'*advice* `INDEX_SCAN` indique que la table `t1` doit être parcourue via un parcours sur l'index de la clé primaire. Quant à l'*advice* `NO_GATHER`, il indique qu'il n'y a pas de parallélisation souhaitée pour ce parcours.

Nous allons maintenant conseiller un parcours séquentiel plutôt qu'un parcours d'index. Cela se fait en configurant le paramètre `pg_plan_advice.advice` :

```
SET pg_plan_advice.advice TO 'SEQ_SCAN (t1)';
```

Regardons maintenant le plan généré :

```
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=50;
```

```
           QUERY PLAN
-------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on t1
         Filter: (c1 = 50)
 Supplied Plan Advice:
   SEQ_SCAN(t1) /* matched */
 Generated Plan Advice:
   SEQ_SCAN(t1)
   GATHER(t1)
(9 rows)
```

La ligne `Supplied Plan Advice` indique l'*advice* préconisé par l'utilisateur, et la ligne `Generated Plan Advice` l'*advice* du plan généré. Le parcours séquentiel a bien été respecté comme

l'indique le mot clé `matched`. Comme il a été demandé au planificateur de prioriser un parcours séquentiel, mais sans indication sur l'utilisation ou non de la parallélisation, il a décidé de l'activer dans ce cas. Mais nous pouvons lui déconseiller cela aussi :

```
SET pg_plan_advice.advice TO 'SEQ_SCAN (t1) NO_GATHER(t1)';
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=50;
```

```
          QUERY PLAN
------------------------------
 Seq Scan on t1
   Filter: (c1 = 50)
 Supplied Plan Advice:
   SEQ_SCAN(t1) /* matched */
   NO_GATHER(t1) /* matched */
 Generated Plan Advice:
   SEQ_SCAN(t1)
   NO_GATHER(t1)
(8 rows)
```

Ici, nous voyons bien que PostgreSQL a accepté à la fois le conseil sur le parcours séquentiel et celui sur la parallélisation.

Notons bien qu'il ne suit pas forcément les conseils donnés. Ajoutons par exemple un index sur la colonne `c2` et ajoutons un *advice* pour un parcours d'index sur ce nouvel index :

```
CREATE INDEX ON t1(c2);
SET pg_plan_advice.advice TO 'INDEX_SCAN (t1 t1_c2_idx)';
```

Voici le plan généré pour cette requête :

```
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=50;
```

```
                    QUERY PLAN
--------------------------------------------------
 Index Scan using t1_pkey on t1
   Disabled: true
   Index Cond: (c1 = 50)
 Supplied Plan Advice:
   INDEX_SCAN(t1 t1_c2_idx) /* matched, failed */
 Generated Plan Advice:
   INDEX_SCAN(t1 public.t1_pkey)
   NO_GATHER(t1)
(8 rows)
```

Bien que nous ayons conseillé un parcours d'index et que ce dernier correspondait à la requête (`matched`), il n'était pas utilisable pour cette requête (`failed`). De ce fait, nous avons un parcours d'index en utilisant celui de la clé primaire (d'où le `Disabled` à `true`).

Maintenant, dans cet exemple, nous notons le cas d'un *advice* inapplicable (tentative d'utilisation d'un index sur une autre table que celle qui est lue) :

```
CREATE TABLE t2(c1 integer);
CREATE INDEX ON t2(c1);
SET pg_plan_advice.advice TO 'INDEX_SCAN (t1 t2_c1_idx)';
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=50;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using t1_pkey on t1
   Index Cond: (c1 = 50)
 Supplied Plan Advice:
   INDEX_SCAN(t1 t2_c1_idx) /* matched, inapplicable, failed */
 Generated Plan Advice:
   INDEX_SCAN(t1 public.t1_pkey)
   NO_GATHER(t1)
(7 rows)
```

Le mot-clé `inapplicable` indique bien que l'*advice* n'est pas applicable dans le contexte de cette requête.

Il est à noter qu'à partir du moment où le paramètre `pg_plan_advice.advice` est configuré, il est utilisé pour toutes les requêtes exécutées, avec ou sans `EXPLAIN` (voir `pg_stash_advice` plus loin pour limiter l'utilisation à une requête). En voici une démonstration :

```
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM pg_namespace;
```

```
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on pg_namespace
 Supplied Plan Advice:
   INDEX_SCAN(t1 t2_c1_idx) /* not matched */
 Generated Plan Advice:
   SEQ_SCAN(pg_namespace)
   NO_GATHER(pg_namespace)
(6 rows)
```

L'*advice* a bien été fourni au planificateur mais ce dernier a compris qu'il ne correspondait pas à la requête exécutée (pas de table `t1` dans `SELECT * FROM pg_namespace`). Cela se fait aussi sans l'`EXPLAIN` et son option `PLAN_ADVICE` même si cela ne se voit pas :

```
demo=# SELECT * FROM pg_namespace;
```

```
  oid  |      nspname       | nspowner |                       nspacl
-------+--------------------+----------+---------------------------------------------
    99 | pg_toast           |       10 |
    11 | pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
```

```
  2200 | public             |     6171 |
↪  {pg_database_owner=UC/pg_database_owner,=U/pg_database_owner}
 14381 | information_schema |       10 | {postgres=UC/postgres,=U/postgres}
(4 rows)
```

L'option `pg_plan_advice.feedback_warnings` permet l'affichage d'un message dans ce cas :

```
SET pg_plan_advice.feedback_warnings TO on;
SELECT * FROM pg_namespace;
```

```
WARNING:  supplied plan advice was not enforced
DETAIL:  advice INDEX_SCAN(t1 t2_c1_idx) feedback is "not matched"
  oid  |      nspname       | nspowner |                        nspacl
-------+--------------------+----------+-----------------------------------------
    99 | pg_toast           |       10 |
    11 | pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
  2200 | public             |     6171 |
↪  {pg_database_owner=UC/pg_database_owner,=U/pg_database_owner}
 14381 | information_schema |       10 | {postgres=UC/postgres,=U/postgres}
(4 rows)
```

Les lignes `WARNING` et `DETAIL` indiquent bien que le module `pg_plan_advice` a proposé un `advice` au planificateur mais que ce dernier n'a pas pu l'utiliser.

Nous n'avons vu ici que quelques balises pour les `advices` mais il en existe beaucoup d'autres. La documentation de ce module les liste exhaustivement.

4.5 MODULE PG_STASH_ADVICE



- Librairie et extension
  - donc `CREATE EXTENSION` nécessaire
- Fonctions
  - `pg_create_advice_stash()`
  - `pg_set_stashed_advice()`
  - `pg_drop_advice_stash()`
  - `pg_set_stashed_advice()`
  - etc

Le module `pg_plan_advice` permet de conseiller le planificateur. Cependant, il est nécessaire de configurer le paramètre `pg_plan_advice.advice` avant d'exécuter la requête, ce qui nécessite dans la plupart des cas d'avoir à modifier l'application qui utilise cette requête.

Le module `pg_stash_advice` a pour but de stocker les `advices` en les associant à une requête précise. Ce stockage est accessible par toutes les sessions et, de ce fait, il n'est plus nécessaire de modifier l'application pour qu'elle en profite.

Pour que ce module fonctionne, il faut tout d'abord configurer le paramètre `shared_preload_libraries` ou le paramètre `session_preload_libraries`. Il est préférable d'utiliser le premier paramètre mais cela ne sera pas toujours possible car la prise en compte de sa modification nécessite un redémarrage de l'instance.

Voici la configuration dans notre cas :

```
shared_preload_libraries = 'pg_plan_advice,pg_stash_advice'
compute_query_id = on
```

Supposons que nous voulions toujours passer par un parcours séquentiel pour la requête suivante :

```
SELECT * FROM t1 WHERE c1=?
```

Pour cela, nous avons besoin de récupérer son identifiant. Cela peut se faire via l'extension `pg_stat_statements` ou via la commande `EXPLAIN` et son option `VERBOSE` :

```
EXPLAIN (COSTS OFF, VERBOSE) SELECT * FROM t1 WHERE c1=10;

               QUERY PLAN
----------------------------------------
```

```
 Index Scan using t1_pkey on public.t1
   Output: c1, c2
   Index Cond: (t1.c1 = 10)
 Query Identifier: -5932685914642725193
(4 rows)
```

La ligne `Query Identifier` nous donne cette information.

Pour aller plus loin, nous avons besoin d'ajouter l'extension `pg_stash_advice` :

```
CREATE EXTENSION pg_stash_advice;
```

Cette extension fournit un ensemble de fonctions que nous allons utiliser :

```
demo=# \dx+ pg_stash_advice
      Objects in extension "pg_stash_advice"
                Object description
---------------------------------------------------
 function pg_create_advice_stash(text)
 function pg_drop_advice_stash(text)
 function pg_get_advice_stash_contents(text)
 function pg_get_advice_stashes()
 function pg_set_stashed_advice(text,bigint,text)
 function pg_start_stash_advice_worker()
(6 rows)
```

Il nous faut tout d'abord créer un dépôt :

```
SELECT pg_create_advice_stash('ma_demo');
```

Ensuite, je vais pouvoir associer un `advice` à une requête dans mon dépôt :

```
SELECT pg_set_stashed_advice('ma_demo', -5932685914642725193, 'SEQ_SCAN(t1)');
```

La fonction `pg_get_advice_stashes()` indique bien qu'il y a une entrée dans le dépôt `ma_demo` :

```
SELECT * FROM pg_get_advice_stashes();
```

```
 stash_name | num_entries
------------+-------------
 ma_demo    |           1
(1 row)
```

Quant à la fonction `pg_get_advice_stash_contents()`, elle donne le contenu d'un dépôt particulier :

```
SELECT * FROM pg_get_advice_stash_contents('ma_demo');
```

```
 stash_name |       query_id       | advice_string
------------+----------------------+---------------
 ma_demo    | -5932685914642725193 | SEQ_SCAN(t1)
(1 row)
```

Actuellement, aucun dépôt n'a été sélectionné, donc il n'y a aucune modification du comportement du planificateur :

```
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=10;
```

```
           QUERY PLAN
--------------------------------
 Index Scan using t1_pkey on t1
   Index Cond: (c1 = 10)
 Generated Plan Advice:
   INDEX_SCAN(t1 public.t1_pkey)
   NO_GATHER(t1)
(5 rows)
```

En configurant le paramètre `pg_stash_advice.stash_name` à `ma_demo`, le planificateur va prendre en compte les `advices` configurés :

```
SET pg_stash_advice.stash_name TO 'ma_demo';
EXPLAIN (COSTS OFF, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=10;
```

```
               QUERY PLAN
----------------------------------------
 Gather
   Output: c1, c2
   Workers Planned: 2
   ->  Parallel Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = 10)
 Query Identifier: -5932685914642725193
 Supplied Plan Advice:
   SEQ_SCAN(t1) /* matched */
 Generated Plan Advice:
   SEQ_SCAN(t1)
   GATHER(t1)
(12 rows)
```

Si la valeur testée change, cela ne change pas le *query identifier* et de ce fait, l'*advice* est toujours pris en compte :

```
SET pg_stash_advice.stash_name TO 'ma_demo';
EXPLAIN (COSTS OFF, VERBOSE, PLAN_ADVICE) SELECT * FROM t1 WHERE c1=15;
```

```
               QUERY PLAN
----------------------------------------
```

```
 Gather
   Output: c1, c2
   Workers Planned: 2
   ->  Parallel Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = 15)
 Query Identifier: -5932685914642725193
 Supplied Plan Advice:
   SEQ_SCAN(t1) /* matched */
 Generated Plan Advice:
   SEQ_SCAN(t1)
   GATHER(t1)
(12 rows)
```

Par contre, si la requête change, la nouvelle requête ne prendra pas en compte l'*advice* :

```
EXPLAIN (COSTS OFF, VERBOSE, PLAN_ADVICE) SELECT * FROM t1 WHERE c1<15;

               QUERY PLAN
----------------------------------------
 Index Scan using t1_pkey on public.t1
   Output: c1, c2
   Index Cond: (t1.c1 < 15)
 Query Identifier: -5542344203702659193
 Generated Plan Advice:
   INDEX_SCAN(t1 public.t1_pkey)
   NO_GATHER(t1)
(7 rows)
```

Ce paramètre est configurable au niveau d'une base ou d'un utilisateur. De ce fait, on peut configurer des *advices* par application par exemple.

4.6 DEFAULT_TOAST_COMPRESSION = LZ4



- Compression `lz4` plus efficace et plus rapide que `pglz`
- Conseillé déjà sur versions précédentes

Avec le mécanisme TOAST, PostgreSQL sait compresser certains champs, notamment les longs textes, les binaires, les JSON, les grands tableaux…

L'algorithme par défaut pour cette compression devient `lz4` et non plus l'antique `pglz`. Selon les données, `lz4` peut mieux compresser, et il est surtout plus rapide.

Il était déjà disponible depuis PostgreSQL 14 et la fonctionnalité est donc stable. Si la librairie nécessaire n'est pas disponible lors de la compilation, `pglz` est la solution de repli. LZ4 est bien sûr intégré dans les distributions habituelles.



Sur les versions précédentes, nous conseillons déjà de positionner `default_toast_compression = lz4`.

En cas de migration, les champs compressés existants ne sont pas modifiés. Les données ne sont décompressées et recompressées avec le nouvel algorithme qu'en cas de modification de la valeur du champ.

Pour les détails sur le stockage TOAST et un benchmark, voir ce billet du blog Dalibo[1] ou cet extrait de la formation DBA2[2].

[1] https://blog.dalibo.com/2024/02/27/toast.html
[2] https://dali.bo/m4_html#toast

4.7 NOUVELLES FONCTIONNALITÉS DE EXPLAIN



- `IO` : informations sur les IO asynchrones

```
EXPLAIN (ANALYZE, IO) SELECT …

…
Prefetch: avg=8.49 max=16 capacity=99
I/O: count=794 waits=0 size=15.92 in-progress=1.00
…
```

- WAL : nombre d'octets pour les FPI
- Nœud `Memoize` : estimations de cache et de recherche

```
Estimates: capacity=5 distinct keys=5 lookups=100000 hit percent=100.00%
```

Comme à l'occasion de toute nouvelle version, la commande `EXPLAIN` se voit améliorée. Cela concerne trois axes pour cette version 19.

4.7.1 Nouvelle option IO

Cette nouvelle option permet de récupérer des informations sur le nouveau système d'entrées/sorties disques asynchrones. Elle donne des détails sur la distance des lectures en avance et sur les requêtes d'entrées/sorties disque.

Pour l'instant, cela ne concerne que les parcours séquentiels et les parcours bitmap.

Elle ne fonctionne que si l'option `ANALYZE` est elle-aussi utilisée.

Voici notre jeu de tests :

```
CREATE TABLE t1(c1 integer, c2 text);
INSERT INTO t1
  SELECT i, 'Ligne '||i FROM generate_series(1, 2_000_000) i;

SELECT pg_buffercache_evict_relation('t1');
```

Et voici le résultat d'un `EXPLAIN (ANALYZE, IO)` en lecture de cette table :

```
 Seq Scan on t1  (cost=0.00..32643.00 rows=2000000 width=17)
       (actual time=0.094..2927.873 rows=2000000.00 loops=1)
   Prefetch: avg=8.49 max=16 capacity=99
   I/O: count=794 waits=0 size=15.92 in-progress=1.00
```

```
   Buffers: shared read=12643
 Planning Time: 0.068 ms
 Execution Time: 5724.011 ms
```

Les deux nouvelles lignes sont les lignes `Prefetch` et `I/O`.

La ligne `Prefetch` indique la distance moyenne de lecture en avance, la distance maximale et la capacité du système (calculées à partir de différents facteurs dont le paramètre `effective_io_concurrency`). Ici, 8 blocs sont lus en avance en moyenne, le maximum était 16, et il reste théoriquement de la marge si la latence devait augmenter.

La ligne `I/O` précise le nombre de demandes de lecture (`count`) et la taille moyenne d'une lecture (sur `size`). Si les deux sont multipliés, on obtient sur cet exemple 12640, soit pratiquement les 12643 blocs indiqués dans la partie `read` de la ligne `Buffers`. En fait, 12643 blocs divisé par 794 donne 15,9231…, arrondi à 15,92. Cette taille moyenne montre que les lectures sont bien groupées, et qu'elles respectent la configuration du paramètre `io_combine_limit` qui est à 16. Le nombre `waits` indique le nombre d'attentes pour des I/O. Ici, aucune attente. En moyenne, il y avait une seule lecture en cours (`in-progress`), car le disque SSD de la machine de test a une très faible latence.

Les I/O asynchrones sont intéressantes quand la latence du stockage est importante, mais que celui-ci est capable de traiter plusieurs demandes de lecture en parallèle, ce qui est le cas dans un environnement *cloud*. Si nous ajoutons artificiellement de la latence à notre disque SSD, cela donne le résultat suivant :

```
 Seq Scan on t1  (cost=0.00..32643.00 rows=2000000 width=17)
      (actual time=56.144..5554.588 rows=2000000.00 loops=1)
   Prefetch: avg=87.54 max=96 capacity=99
   I/O: count=794 waits=534 size=15.92 in-progress=5.92
   Buffers: shared read=12643
 Planning Time: 0.039 ms
 Execution Time: 8430.891 ms
```

On observe alors que la distance moyenne de lecture en avance a beaucoup augmenté et s'approche de la capacité maximale. Il y a de l'attente sur 534 des 794 opérations de lecture, et il y avait en moyenne 5,92 lectures en cours. Ce dernier nombre peut être limité par `effective_io_concurrency`, qui vaut 16 par défaut :

```
SET effective_io_concurrency TO 4;
```

```
 Seq Scan on t1  (cost=0.00..32643.00 rows=2000000 width=17)
      (actual time=52.557..8520.951 rows=2000000.00 loops=1)
   Prefetch: avg=70.82 max=79 capacity=80
   I/O: count=794 waits=535 size=15.92 in-progress=3.96
   Buffers: shared read=12643
 Settings: effective_io_concurrency = '4'
 Planning Time: 0.034 ms
```

```
 Execution Time: 11450.589 ms
```

On remarque au passage que le temps d'exécution a augmenté en proportion de la baisse de ce nombre de lectures simultanées.

Ce qui n'a pas changé est le nombre de blocs lus par appel système, qui est resté à (presque) 16 en moyenne. Ce nombre peut baisser si une partie de la table est en cache. Avec les requêtes suivantes, on met en cache deux blocs sur huit, et on a des suites de six blocs contigus hors cache :

```
SELECT pg_buffercache_evict_relation('t1');

SELECT i FROM generate_series(1, 12640, 8) AS T(i)
  CROSS JOIN LATERAL pg_prewarm('t1', 'buffer', 'main', i, i+1);
```

On observe alors que le compteur `size` est passé de 16 à 6 :

```
 Seq Scan on t1  (cost=0.00..32643.00 rows=2000000 width=17) (actual
↪  time=55.787..19431.393 rows=2000000.00 loops=1)
   Prefetch: avg=33.33 max=37 capacity=80
   I/O: count=1582 waits=731 size=5.99 in-progress=3.98
   Buffers: shared hit=3160 read=9483
 Settings: effective_io_concurrency = '4'
 Planning Time: 0.055 ms
 Execution Time: 22483.690 ms
```

Le temps d'exécution a lui presque doublé, mais l'impact est généralement bien plus modéré, et s'explique ici par la façon dont la latence a été artificiellement augmentée. Si on met plutôt la première moitié des blocs en cache, on peut remarquer que le mécanisme de lecture asynchrone est désactivé pour lire cette première moitié de table :

```
SELECT pg_prewarm('t1', 'buffer', 'main', 0, 6300);

 Seq Scan on t1  (cost=0.00..32643.00 rows=2000000 width=17)
       (actual time=0.017..5771.420 rows=2000000.00 loops=1)
   Prefetch: avg=35.52 max=79 capacity=80
   I/O: count=400 waits=283 size=15.86 in-progress=3.90
   Buffers: shared hit=6301 read=6342
 Settings: effective_io_concurrency = '4'
 Planning Time: 0.036 ms
 Execution Time: 8638.707 ms
```

Le nombre d'opérations de lecture était 794 initialement, il est passé à 400, mais `size` est remonté à 15,86, presque la valeur maximale possible, puisque les blocs hors cache sont contigus sur la deuxième moitié.

4.7.2 Ajout du nombre d'octets pour les FPI

Les journaux de transaction contiennent des entrées unitaires, mais aussi des copies de blocs entiers de 8 ko (*Full Page Images*[3] ou FPI) pour des raisons de fiabilité.

L'option `WAL`de `EXPLAIN` est plus loquace sur les écritures dans les journaux de transactions. Nous avions déjà le nombre d'enregistrements FPI mais pas leur taille. C'est chose faite en version 19 :

Voici la requête étudiée :

```
EXPLAIN (ANALYZE, WAL, COSTS OFF, TIMING OFF, BUFFERS OFF)
  INSERT INTO t1 SELECT i, 'Ligne '||i from generate_series(1, 1_000_000) i;
```

Et voici son résultat :

```
Insert on t1 (actual rows=0.00 loops=1)
  WAL: records=1000001 fpi=2 bytes=71905218 fpi bytes=16268 buffers full=6795
  ->  Function Scan on generate_series i (actual rows=1000000.00 loops=1)
Planning Time: 0.154 ms
Execution Time: 1428.113 ms
```

La nouveauté est la partie `fpi bytes=N` de la ligne `WAL:`. Elle indique le nombre d'octets des deux enregistrements FPI.

4.7.3 Ajout des estimations de cache et de recherche

Le nœud `Memoize` affiche quelques estimations supplémentaires pouvant expliquer le fait qu'il a été sélectionné.

Prenons un jeu de tests :

```
CREATE TABLE t2 AS
  SELECT i, i % 5 AS j FROM generate_series(1, 100000) AS i;

CREATE TABLE t3 AS
  SELECT i, i % 20000 as j FROM generate_series(1, 100000) AS i;

CREATE INDEX ON t3(j);
```

Et récupérons le plan de la requête suivante :

```
SELECT * FROM t2 JOIN t3 ON t2.j = t3.j;
```

Voici le plan :

[3] https://docs.postgresql.fr/current/runtime-config-wal.html#GUC-FULL-PAGE-WRITES

```
Nested Loop  (cost=0.30..8950.55 rows=501731 width=16)
  ->  Seq Scan on t2  (cost=0.00..1448.00 rows=100000 width=8)
  ->  Memoize  (cost=0.30..0.44 rows=5 width=8)
        Cache Key: t2.j
        Cache Mode: logical
        Estimates: capacity=5 distinct keys=5 lookups=100000 hit percent=100.00%
        ->  Index Scan using t3_j_idx on t3  (cost=0.29..0.43 rows=5 width=8)
              Index Cond: (j = t2.j)
Settings: enable_mergejoin = 'off', random_page_cost = '8'
```

La ligne `Settings` indique que j'ai triché un peu en désactivant la jointure par *Merge Join* que l'optimiseur aurait plutôt choisi, en rendant plus coûteux les parcours d'index.

La nouvelle information est fournie par la ligne `Estimates`. Les nombres sont des estimations : 5 clés distinctes, 100 000 recherches, un pourcentage de lecture du cache de 100 %.

Pourquoi 5 clés distinctes ? L'INSERT est assez parlant mais on peut aussi récupérer cette information de deux façons. D'abord, via les statistiques sur les données :

```
SELECT n_distinct
FROM pg_stats
WHERE tablename='t2'
  AND attname='j';
```

```
 n_distinct
------------
          5
(1 row)
```

Ou bien en interrogeant directement la table :

```
SELECT j, count(*) FROM t2 GROUP BY j;
```

```
 j | count
---+-------
 0 | 20000
 1 | 20000
 3 | 20000
 4 | 20000
 2 | 20000
(5 rows)
```

La colonne `j` contient bien 5 valeurs distinctes. Il y aura 100 000 recherches dans le cache car il estime le nombre de lignes de `t2` à 100 000. Comme toutes les valeurs distinctes tiennent dans le cache, toutes les lectures iront dans le cache, d'où un ratio de 100 %. Il est la raison pour laquelle le nœud `Memoize` est intéressant dans ce cas précis.

Pour finir, `capacity` représente la capacité maximale du cache en termes de nombre maximal de clés possibles. Elle est calculée en fonction de `work_mem` et de `hash_mem_multiplier` (le

cache étant une table de hachage), ainsi que de la taille estimée des clés. Elle est aussi bornée par le `n_distinct` de la table, car il n'y pas d'intérêt à provisionner plus de mémoire que nécessaire, et cela explique pourquoi, dans cet exemple, `capacity` vaut seulement 5, alors qu'on a 4 Mo de `work_mem`, soit une capacité mémoire pour environ 29 000 clés et les lignes associées.

5/ Optimiseur

5.1 JIT DÉSACTIVÉ PAR DÉFAUT



- `jit = off` nouveau défaut
- Trop souvent contre-productif
- À réactiver pour de l'analytique, après tests

Depuis la version 11, une des fonctionnalités les plus techniquement pointues de PostgreSQL est la « compilation à la volée » (Just In Time compilation, ou JIT) de certaines expressions dans les requêtes SQL. Dans certaines requêtes, l'essentiel du temps est passé à décoder des enregistrements (*tuple deforming*), à analyser des clauses WHERE, à effectuer des calculs. En conséquence, l'idée du JIT est de transformer tout ou partie de la requête en un programme natif directement exécuté par le processeur. Le JIT de PostgreSQL s'appuie actuellement sur la chaîne de compilation LLVM.

Elle était activée par défaut depuis la version 12, la version 19 revient en arrière sur ce point.

En effet, de nombreuses contre-performances dues au JIT ont été observées, notamment sur des requêtes très complexes, donc au coût élevé. La compilation et l'optimisation du code généré sont en effet des opérations lourdes. Le partitionnement et l'alourdissement de LLVM au fil des versions ont aussi un impact.

De l'avis même de son auteur, l'algorithme de déclenchement du JIT est « naïf ». Se baser sur le coût de la requête est trop simple.

Il était souvent conseillé d'augmenter d'un facteur 10 les seuils des différents niveaux d'optimisation `jit_above_cost`, `jit_inline_above_cost`, `jit_optimize_above_cost`.



Nous conseillons même de désactiver `jit` sur les versions précédentes, pour les applications de type OLTP.

Le JIT garde un intérêt s'il y a des dizaines de millions de lignes, ou plus, à analyser, donc plutôt dans les requêtes analytiques. Dans ce cas, `jit` peut être repassé à on, dans la session ou globalement, si l'intérêt en terme de performances a été testé.

5.2 STATISTIQUES SUR LES COLONNES VIRTUELLES



- Les colonnes virtuelles existent depuis PostgreSQL 18
- Les statistiques n'étaient pas calculées
- C'est à présent possible sur demande explicite :

```
CREATE STATISTICS nomstat ON col_v FROM nomtable ;
ANALYZE nomtable ;
```

Les colonnes virtuelles apparues dans PostgreSQL 18 ne possédaient pas de statistiques. Si un critère portait sur cette colonne, l'optimiseur n'avait aucun moyen d'estimer la sélectivité et appliquait des forfaits. Cela pouvait entraîner des sous-estimations massives.

PostgreSQL 19 ne crée toujours pas de statistiques sur ces colonnes par défaut. En effet, leur calcul est souvent coûteux (toutes les valeurs de l'échantillon sont recalculées), alors que ce sont souvent des colonnes de présentation. De plus, il était déjà possible de créer un index fonctionnel avec la même formule, qui vient avec ses statistiques si la colonne générée sert souvent à la sélection.

Il est nécessaire de passer par la syntaxe des statistiques étendues.

```
CREATE STATISTICS nomstat ON colonne_virtuelle FROM nomtable ;
```

Exemple :

```
-- Une table avec des paquets,
-- un champ calculé volume, et un champ calculé annee
CREATE UNLOGGED TABLE paquets (
    id int PRIMARY KEY,
    longueur    int,
    hauteur     int,
    largeur     int,
    t           timestamptz,
    filler      char(50) DEFAULT ' ',
    volume      int
        GENERATED ALWAYS AS (longueur*hauteur*largeur)
        VIRTUAL
        CHECK (volume > 0),
    annee       int
        GENERATED ALWAYS AS (extract(year FROM t AT TIME ZONE 'Europe/Paris'))
        VIRTUAL
    ) ;

INSERT INTO paquets
```

```
SELECT i, 1+mod(i,89), 1+mod(i,97), 1+mod(i,83),
'2025-12-31'::timestamptz+mod (i,370)*interval '23h'
FROM generate_series (1, 1_000_000) i ;
```

Avec les statistiques classiques, une erreur d'estimation peut être énorme :

```
-- Pour réduire la volumétrie à afficher
SET default_statistics_target TO 20 ;

VACUUM (ANALYZE) paquets ;

-- Ceci ne renvoie rien
-- car ce sont des colonnes générées
SELECT * FROM pg_stats
WHERE tablename = 'paquets' AND attname in ( 'volume', 'annee');

-- Estimation totalement fausse
-- (estimation forfaitaire de 0,5%, au lieu de 95 % réel)
EXPLAIN (ANALYZE, BUFFERS OFF)
SELECT * FROM paquets
WHERE annee = 2026 ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on paquets  (cost=0.00..33397.50 rows=5000 width=83) (actual
 ↪  time=0.024..1198.749 rows=994595.00 loops=1)
   Filter: ((EXTRACT(year FROM (t AT TIME ZONE 'Europe/Paris'::text)))::integer =
 ↪  2026)
   Rows Removed by Filter: 5405
```

Création des statistiques étendues :

```
CREATE STATISTICS paquets_volume_stat ON volume FROM paquets ;
CREATE STATISTICS paquets_annee_stat ON annee  FROM paquets ;
ANALYZE paquets ;
```

Pour les consulter, ce n'est pas dans `pg_stats`, mais avec les autres statistiques étendues :

```
SELECT * FROM pg_stats_ext WHERE tablename LIKE '%paquets%' \gx
```

```
-[ RECORD 1 ]----------+--------------------
schemaname             | public
tablename              | paquets
tableid                | 25495
statistics_schemaname  | public
statistics_name        | paquets_volume_stat
statistics_id          | 25505
statistics_owner       | postgres
attnames               | ø
exprs                  | {volume}
```

```
kinds                  | {e}
inherited              | f
n_distinct             | ø
dependencies           | ø
most_common_vals       | ø
most_common_val_nulls  | ø
most_common_freqs      | ø
most_common_base_freqs | ø
-[ RECORD 2 ]----------+--------------------
schemaname             | public
tablename              | paquets
tableid                | 25495
statistics_schemaname  | public
statistics_name        | paquets_annee_stat
statistics_id          | 25506
statistics_owner       | postgres
attnames               | ø
exprs                  | {annee}
kinds                  | {e}
inherited              | f
n_distinct             | ø
dependencies           | ø
most_common_vals       | ø
most_common_val_nulls  | ø
most_common_freqs      | ø
most_common_base_freqs | ø
```

```
SELECT * FROM pg_stats_ext_exprs WHERE tablename LIKE '%paquets%' \gx
```

```
-[ RECORD 1 ]----------+--------------------
schemaname             | public
tablename              | paquets
tableid                | 25495
statistics_schemaname  | public
statistics_name        | paquets_volume_stat
statistics_id          | 25505
statistics_owner       | postgres
expr                   | volume
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -0.814
most_common_vals       |
 ↪  {3780,14400,18000,18720,8640,18480,20160,1680,3840,7656,13440,20250,20400,21840,26208,
31752,55680,420,2160,2754}
most_common_freqs      |
 ↪  {0.0011666666,0.0011666666,0.0011666666,0.0011666666,0.001,0.001,0.001,0.00083333335,
0.00083333335,0.00083333335,0.00083333335,0.00083333335,0.00083333335,0.00083333335,
```

```
0.00083333335,0.00083333335,0.00083333335,0.00066666666,0.00066666666,0.00066666666}
histogram_bounds       |
 ↪  {15,2196,4800,7968,11700,16376,22032,28782,36120,44892,55593,67782,81120,97020,115736,
136188,162279,199810,243360,316728,655040}
correlation            | -0.018437287
most_common_elems      | ø
most_common_elem_freqs | ø
elem_count_histogram   | ø
range_length_histogram | ø
range_empty_frac       | ø

-[ RECORD 2 ]----------+--------------------
schemaname             | public
tablename              | paquets
tableid                | 25495
statistics_schemaname  | public
statistics_name        | paquets_annee_stat
statistics_id          | 25506
statistics_owner       | postgres
expr                   | annee
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | 2
most_common_vals       | {2026,2025}
most_common_freqs      | {0.9925,0.0075}
histogram_bounds       | ø
correlation            | 0.985847
most_common_elems      | ø
most_common_elem_freqs | ø
elem_count_histogram   | ø
range_length_histogram | ø
range_empty_frac       | ø
range_bounds_histogram | ø
```

L'estimation est bien meilleure :

```
EXPLAIN SELECT * FROM paquets
WHERE annee = 2025 ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on paquets  (cost=0.00..33418.34 rows=6667 width=83)
   Filter: ((EXTRACT(year FROM (t AT TIME ZONE 'Europe/Paris'::text)))::integer =
 ↪  2025)
```

Si l'on change l'expression d'une colonne virtuelle, ses statistiques disparaissent aussitôt. (Noter au passage que PostgreSQL 19 a levé la restriction sur le changement d'expression quand la colonne

virtuelle portait une contrainte CHECK.)

```
ALTER TABLE paquets
ALTER COLUMN volume
SET EXPRESSION AS ((longueur*hauteur*largeur*1.50+1)::int) ;

-- Ceci ne renvoie plus rien
SELECT * FROM pg_stats_ext_exprs
WHERE statistics_name ='paquets_volume_stat' \gx
```

Hélas, cette syntaxe ne fonctionne pas pour les colonnes virtuelles :

```
-- ne marche pas
ANALYZE (VERBOSE) paquets (volume) ;
```

Il faut regénérer totalement les statistiques de la table pour recalculer les statistiques étendues :

```
ANALYZE (VERBOSE) paquets ;
```

Elles sont alors à jour avec les nouvelles valeurs.

5.3 OPTIMISATION SUR NOT IN



- Les clauses `NOT IN (...)` sont converties en anti-jointures
  - uniquement si `NOT NULL`

En version 19, l'optimiseur teste si les colonnes sont `NOT NULL` pour un test `NOT IN`.

Voici un jeu de données pour tester cette optimisation :

```
DROP TABLE IF EXISTS t1;
CREATE UNLOGGED TABLE t1 (c1 integer GENERATED ALWAYS AS IDENTITY, c2 integer NOT
↪  NULL);
INSERT INTO t1 (c2) SELECT random(1, 5) FROM generate_series(1,1000000);
DROP TABLE IF EXISTS t2;
CREATE UNLOGGED TABLE t2 (c1 integer NOT NULL, c2 text);
INSERT INTO t2 SELECT g, 'Ligne '||g FROM generate_series(1,5) g;
VACUUM ANALYZE t1;
VACUUM ANALYZE t2;
CHECKPOINT;
```

La requête de test est la suivante :

```
SELECT * FROM t1
WHERE t1.c2 NOT IN (SELECT t2.c1 FROM t2 WHERE t2.c2='Ligne 3');
```

Voici son résultat sur une version 18 :

```
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on t1 (actual rows=799184.00 loops=1)
   Filter: (NOT (ANY (c2 = (hashed SubPlan 1).col1)))
   Rows Removed by Filter: 200816
   Buffers: shared hit=4426
   SubPlan 1
     ->  Seq Scan on t2 (actual rows=1.00 loops=1)
           Filter: (c2 = 'Ligne 3'::text)
           Rows Removed by Filter: 4
           Buffers: shared hit=1
 Planning:
   Buffers: shared hit=24
 Planning Time: 0.068 ms
 Execution Time: 98.942 ms
(13 rows)
```

Et voici son résultat sur une version 19 :

```
                        QUERY PLAN
-----------------------------------------------------------
 Hash Anti Join (actual rows=799727.00 loops=1)
   Hash Cond: (t1.c2 = t2.c1)
   Buffers: shared hit=4426
   ->  Seq Scan on t1 (actual rows=1000000.00 loops=1)
         Buffers: shared hit=4425
   ->  Hash (actual rows=1.00 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         Buffers: shared hit=1
         ->  Seq Scan on t2 (actual rows=1.00 loops=1)
               Filter: (c2 = 'Ligne 3'::text)
               Rows Removed by Filter: 4
               Buffers: shared hit=1
 Planning:
   Buffers: shared hit=83
 Planning Time: 0.181 ms
 Execution Time: 87.672 ms
(16 rows)
```

Nous voyons ici que le sous-plan n'existe plus en version 19. Il a été remplacé par une simple jointure. Le gain (en durée d'exécution) est limité dans cet exemple.

5.4 OPTIMISATION SUR IS NOT DISTINCT FROM



- Les clauses `IS NOT DISTINCT FROM` sont optimisées
  - uniquement sur les colonnes avec une contrainte NOT NULL

La clause `IS DISTINCT FROM` est trop peu connue. Elle permet de revenir à une logique binaire dans le cas où de NULL peuvent se glisser dans les valeurs, pour éviter de rajouter des clauses du genre `OR … IS NULL`.

```
CREATE TABLE test1 (id, solde) AS
VALUES (1,0), (2,0), (3,17), (4, null) ;

SELECT * FROM test1 WHERE solde != 0 ;

 id | solde
----+-------
  3 |    17

SELECT * FROM test1 WHERE solde IS DISTINCT FROM 0 ;

 id | solde
----+-------
  3 |    17
  4 |
```

`IS NOT DISTINCT` est même utilisable dans une clause de jointure, ce qui n'est pas forcément facile à gérer pour l'optimiseur, qui utilisera souvent un `Nested Loop`.

En version 19, l'optimiseur optimise le cas où des colonnes `NOT NULL` font partie d'un test `IS NOT DISTINCT FROM`, ce qui se ramène en fait à une simple égalité entre valeurs non nulles.

Voici un jeu de tests :

```
DROP TABLE IF EXISTS t1;
CREATE UNLOGGED TABLE t1 (c1 integer NOT NULL);
INSERT INTO t1 SELECT generate_series(1,1000000);
DROP TABLE IF EXISTS t2;
CREATE UNLOGGED TABLE t2 (c1 integer NOT NULL);
INSERT INTO t2 SELECT generate_series(1,1000000);
VACUUM ANALYZE t1;
VACUUM ANALYZE t2;
CHECKPOINT;
```

Et voici la requête de test :

```
SELECT * FROM t1
JOIN t2 ON t1.c1 IS NOT DISTINCT FROM t2.c1;
```

En version 18, il n'a pas été possible d'obtenir le résultat. Nous avons dû arrêter son exécution au bout de dix minutes. Le plan a effectivement un coût effarant (ici parallélisme désactivé) :

```
EXPLAIN SELECT * FROM t1
JOIN t2 ON t1.c1 IS NOT DISTINCT FROM t2.c1;
```default
                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=0.00..18907031350.00 rows=1000000 width=8)
   Join Filter: (NOT (t1.c1 IS DISTINCT FROM t2.c1))
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
   ->  Materialize  (cost=0.00..23332.00 rows=1000000 width=4)
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
```

Mais voici son résultat sur une version 19 :

```
EXPLAIN (ANALYZE) SELECT * FROM t1
JOIN t2 ON t1.c1 IS NOT DISTINCT FROM t2.c1;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Hash Join  (cost=30832.00..70728.00 rows=1000000 width=8)
               (actual time=2480.295..6475.466 rows=1000000.00 loops=1)
   Hash Cond: (t1.id = t2.id)
   Buffers: shared hit=8850, temp read=5134 written=5134
   I/O Timings: temp read=13.974 write=42.817
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
                  (actual time=0.018..1192.456 rows=1000000.00 loops=1)
         Buffers: shared hit=4425
   ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
                (actual time=2479.872..2479.876 rows=1000000.00 loops=1)
         Buckets: 262144  Batches: 8  Memory Usage: 6446kB
         Buffers: shared hit=4425, temp written=2560
         I/O Timings: temp write=21.726
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                  (actual time=0.009..1176.287 rows=1000000.00 loops=1)
               Buffers: shared hit=4425
 Planning Time: 0.141 ms
 Execution Time: 7671.215 ms
```

Le plan est le même que pour une jointure `ON t1.c1=t2.c1`.

La version 19 diminue donc fortement la durée d'exécution sur ce type de cas.

```

6/ Supervision

6.1 NOUVELLE VUE PG_STAT_LOCK



- Informations sur les verrous en attente
  - Si durée > `deadlock_timeout` (1s)
- 5 colonnes
  - `locktype` : *relation*, *transaction_id*…
  - `waits` : nombre
  - `wait_time` : durée
  - `fastpath_exceeded` : sans *fast path*
  - `stats_reset` : depuis quand ?

La vue `pg_stat_lock` donne une vision globale des verrous rencontrés au niveau de l'instance. Elle est cumulative.

La colonne `locktype` indique le type de verrous (par exemple `relation`, `transactionid`, etc). Les informations numériques qui suivent correspondent à ce type de verrous.

La colonne `waits` indique le nombre de fois où une requête était en attente de ce type de verrous. La colonne `wait_time` précise la durée totale d'attente. La colonne `fastpath_exceeded` indique le nombre de fois où le verrou n'a pas été obtenu avec un *fast path*, la limite de la structure mémoire associée ayant été atteinte (augmenter la valeur du paramètre `max_locks_per_transaction` permet d'éviter ce souci).

Enfin, la colonne `stats_reset` indique depuis quand les statistiques sont cumulées.

Voici un exemple d'utilisation. Commençons par créer une table `t1` :

```
CREATE TABLE t1(id integer);
```

Regardons le contenu de la vue `pg_stat_lock` :

```
SELECT * FROM pg_stat_lock WHERE locktype='relation';
```

```
 locktype | waits | wait_time | fastpath_exceeded |          stats_reset
----------+-------+-----------+-------------------+-------------------------------
 relation |     2 |     38251 |             40363 | 2026-06-18 13:55:14.399003+02
(1 row)
```

Le type de verrous `relation` a été attendu deux fois, pour une durée de 38 secondes depuis le 18 juin 13h55.

Maintenant, commençons une transaction et lançons une lecture de `t1` :

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

Sur une autre session, nous lançons une suppression de `t1` :

```
DROP TABLE t1;
```

Cette suppression est bloquée en attente de la fin de la première transaction. Au bout de 10 secondes, nous fermons cette transaction. Regardons le contenu de la nouvelle vue :

```
ROLLBACK;
SELECT * FROM pg_stat_lock WHERE locktype='relation';
```

```
 locktype | waits | wait_time | fastpath_exceeded |          stats_reset
----------+-------+-----------+-------------------+-------------------------------
 relation |     3 |     48861 |             40363 | 2026-06-18 13:55:14.399003+02
(1 row)
```

La colonne `waits` a été incrémentée de un, et la colonne `wait_time` a bien environ 10 secondes de plus.

La fonction `pg_stat_get_lock()` est appelée pour récupérer les informations affichées par la vue.

6.2 NOUVELLE VUE PG_STAT_RECOVERY



- Sur un secondaire en réplication physique
- Une seule ligne indiquant la progression de la restauration

```
-[ RECORD 1 ]------------+------------------------------
promote_triggered        | f
last_replayed_read_lsn   | 0/17FFD370
last_replayed_end_lsn    | 0/17FFED08
last_replayed_tli        | 1
replay_end_lsn           | 0/17FFED08
replay_end_tli           | 1
recovery_last_xact_time  | 2026-07-03 11:11:19.930406+02
current_chunk_start_time | 2026-07-03 11:11:23.842326+02
pause_state              | not paused
```

La nouvelle vue système `pg_stats_recovery` permet de suivre la progression de la restauration une fois que l'état de cohérence a été atteint.



Cette vue est utilisable aussi bien pour suivre une restauration que la réplication du côté du réplica. C'est un complément sur le secondaire de la vue `pg_stat_replication` sur le primaire, qui ne fonctionne que si le *streaming* est en place.

Voici par exemple son contenu lors de l'exécution de la phase d'initialisation d'un `pgbench`, qui crée et remplit quelques tables :

```
-[ RECORD 1 ]------------+------------------------------
promote_triggered        | f
last_replayed_read_lsn   | 0/17FFD370
last_replayed_end_lsn    | 0/17FFED08
last_replayed_tli        | 1
replay_end_lsn           | 0/17FFED08
replay_end_tli           | 1
recovery_last_xact_time  | 2026-07-03 11:11:19.930406+02
current_chunk_start_time | 2026-07-03 11:11:23.842326+02
pause_state              | not paused
```

La colonne `promote_triggered` permet de savoir si une demande de promotion a été reçue. De même, la colonne `pause_state` permet de savoir s'il n'y a pas de pause (`not paused`), si une demande de pause a été reçue (`pause requested`) ou si une pause est actuellement en place (`paused`).

Les colonnes `last_replayed_read_lsn` et `last_replayed_end_lsn` indiquent respectivement les emplacements de début et de fin du dernier enregistrement rejoué avec succès à partir des journaux de transactions. La colonne `replay_end_lsn` indique l'emplacement de l'enregistrement en cours de rejeu. Si jamais cet emplacement est identique à `last_replayed_end_lsn`, cela sous-entend qu'aucun enregistrement n'est en cours de rejeu.

La timeline du dernier rejeu est précisée par la colonne `last_replayed_tli` alors que celle du rejeu en cours est visible dans la colonne `replay_end_tli`.

Enfin, il existe deux colonnes « date et heure ». La colonne `recovery_last_xact_time` indique la date et l'heure de la dernière transaction rejouée, et la colonne `current_chunk_start_time` donne le moment où le processus `startup` a rattrapé son retard avec le dernier morceau reçu des journaux de transactions.

Pendant le temps de l'écriture de ce paragraphe, `pgbench` a terminé son écriture, et le contenu de la vue indique bien maintenant qu'il n'y a pas de rejeu en cours :

```
-[ RECORD 1 ]------------+------------------------------
promote_triggered        | f
last_replayed_read_lsn   | 0/4F7B5040
last_replayed_end_lsn    | 0/4F7B5078
last_replayed_tli        | 1
replay_end_lsn           | 0/4F7B5078
replay_end_tli           | 1
recovery_last_xact_time  | 2026-07-03 11:11:36.263827+02
current_chunk_start_time | 2026-07-03 11:13:45.871371+02
pause_state              | not paused
```

6.3 AUTRES NOUVELLES COLONNES DES VUES STATISTIQUES



- Nouvelle colonne `stats_reset` dans :
  - `pg_stat_all_tables`
  - `pg_stat_all_indexes`,
  - `pg_statio_all_sequences`
  - `pg_stat_user_functions`,
  - `pg_stat_database_conflicts`
- `pg_stat_replication_slots.mem_exceeded_count`
- `pg_stat_progress_basebackup.backup_type`
- `pg_stat_wal.wal_fpi_bytes`
- `pg_stat_wal_receiver.status`
- `pg_available_extensions.location`
- `pg_available_extension_version.location`

La colonne `stats_reset` existait déjà dans un bon nombre des catalogues statistiques. Mais elle manquait dans certains. Cet oubli est enfin réparé.

La colonne `mem_exceeded_count` apparaît dans la vue `pg_stat_replication_slots`. Elle est incrémentée à chaque fois que le besoin de mémoire de la réplication logique dépasse la valeur du paramètre `logical_decoding_work_mem`. C'est utile pour dimensionner ce dernier paramètre.

La vue `pg_stat_progress_basebackup` dispose d'une nouvelle colonne nommée `backup_type` qui précise le type de sauvegarde : soit complète (`full`) soit incrémentale (`incremental`), telle que demandée par `pg_basebackup`. Rappelons que cette vue ne concerne pas les sauvegardes faites par des outils qui recopient les fichiers de la base par d'autres moyens, notamment pgBackRest.

Il existait déjà la colonne `wal_fpi` dans la vue `pg_stat_wal`. Cela permettait de connaître le nombre d'enregistrements FPI dans les journaux de transactions, mais cela ne permet pas de connaître la volumétrie de ces FPI. La version 19 ajoute une colonne pour la volumétrie, nommée `wal_fpi_bytes`.

La colonne `status` peut maintenant avoir la valeur `connecting` ajoutée à la vue `pg_stat_wal_receiver`. Cela permet de voir que le `wal receiver` est en cours de connexion.

Les vues `pg_available_extensions` et `pg_available_extension_version` indiquent

maintenant l'emplacement des extensions sur le système. Cela fait suite à l'ajout du paramètre `extension_control_path` en version 18. Par défaut, on verra `$system`.

6.4 NOUVELLE VUE PG_STAT_AUTOVACUUM_SCORES



- Informations sur les scores autovacuum des tables
  - se base sur le nouveau système de priorisation
  - score élevé = prioritaire
- 9 colonnes
  - `score`
  - `xid_score`
  - `mxid_score`
  - `vacuum_score`
  - `vacuum_insert_score`
  - `analyze_score`
  - `do_vacuum`
  - `do_analyze`
  - `for_wraparound`

Cette vue montre les scores des différentes tables, ce qui permet de mieux appréhender le nouveau système de priorisation des traitements.

Les colonnes `relid`, `schemaname` et `relname` permettent d'identifier la table.

Les colonnes de score sont au nombre de cinq :

- `xid_score`, score concernant le bouclage des identifiants de transaction ;
- `mxid_score`, score concernant le bouclage de MultiXid ;
- `vacuum_score` et `vacuum_insert_score`, scores concernant le traitement par `VACUUM` ;
- `analyze_score`, score concernant le traitement par `ANALYZE` ;
- `score`, score le plus élevé parmi les autres précédents scores.

Restent trois colonnes permettant de savoir si la table va bientôt être traitée et pourquoi :

- `do_vacuum`, pour un `VACUUM` ;
- `do_analyze`, pour un `ANALYZE` ;
- `for_wraparound`, pour geler les identifiants.

Voici un exemple d'utilisation. Commençons par créer une table `t1` :

```
CREATE TABLE t1(id integer);
```

Regardons le contenu de la vue `pg_stat_user_tables` pour la table `t1` :

```
SELECT n_live_tup, n_dead_tup, n_mod_since_analyze, n_ins_since_vacuum,
       last_autovacuum, last_autoanalyze
FROM  pg_stat_user_tables
WHERE relname='t1' \gx
```

```
-[ RECORD 1 ]-------+--
n_live_tup          | 0
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_autovacuum     |
last_autoanalyze    |
```

Il n'y a ni ligne vivante ni morte, car la table vient d'être créée. Regardons maintenant le contenu de la vue `pg_stat_autovacuum_scores` toujours pour cette table :

```
SELECT score, vacuum_score, vacuum_insert_score, analyze_score,
  format('%s %s %s',
    CASE WHEN do_vacuum      THEN 'VACUUM'           END,
    CASE WHEN do_analyze     THEN 'ANALYZE'          END,
    CASE WHEN for_wraparound THEN '(for wraparound)' END) AS doing
FROM pg_stat_autovacuum_scores
WHERE relname='t1' \gx
```

```
-[ RECORD 1 ]-------+------
score               | 5e-09
vacuum_score        | 0
vacuum_insert_score | 0
analyze_score       | 0
doing               |
```

Score de 0, donc pas de traitement sur la table.

Insérons 1000 lignes :

```
INSERT INTO t1 SELECT generate_series(1, 1000);
```

Et retournons voir les deux vues (avec les mêmes requêtes que ci-dessus) :

```
-[ RECORD 1 ]-------+-----
n_live_tup          | 1000
n_dead_tup          | 0
n_mod_since_analyze | 1000
n_ins_since_vacuum  | 1000
last_autovacuum     |
last_autoanalyze    |

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

```
score               | 20
vacuum_score        | 0
vacuum_insert_score | 1
analyze_score       | 20
doing               |  ANALYZE
```

Nous avons inséré 1000 lignes, elles sont donc bien vivantes. Les compteurs `n_mod_since_analyze` et `n_ins_since_vacuum` sont eux aussi incrémentés d'autant.

De ce fait, la vue `pg_stat_autovacuum_scores` indique un score assez fort pour un ANALYZE et indique que ce traitement sera fait à la prochaine exécution d'un `autovacuum worker`.

Et ce sera le cas une minute plus tard :

```
-[ RECORD 1 ]-------+-----------------------------
n_live_tup          | 1000
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 1000
last_autovacuum     |
last_autoanalyze    | 2026-07-02 17:38:00.709842+02
```

Un ANALYZE automatique a eu lieu à 17h38, la colonne `n_mod_since_analyze` a été réinitialisée à zéro.

```
-[ RECORD 1 ]-------+-------------------
score               | 0.8333333333333334
vacuum_score        | 0
vacuum_insert_score | 0.8333333333333334
analyze_score       | 0
doing               |
```

Quant à la vue `pg_stat_autovacuum_scores`, les scores ont été revus à la baisse et aucun traitement n'est prévu.

Maintenant, supprimons 500 lignes (soit la moitié de la table) :

```
DELETE FROM t1 WHERE id<501;
```

Les vues racontent autre chose :

```
-[ RECORD 1 ]-------+-----------------------------
n_live_tup          | 500
n_dead_tup          | 500
n_mod_since_analyze | 500
n_ins_since_vacuum  | 1000
last_autovacuum     |
last_autoanalyze    | 2026-07-02 17:38:00.709842+02
```

```
-[ RECORD 1 ]-------+-------------------
score               | 3.3333333333333335
vacuum_score        | 2
vacuum_insert_score | 0.8333333333333334
analyze_score       | 3.3333333333333335
doing               | VACUUM ANALYZE
```

50 % des lignes sont mortes, un `VACUUM ANALYZE` est de rigueur, et c'est bien ce que dit la table `pg_stat_autovacuum_scores`. Une minute après, la table est traitée :

```
-[ RECORD 1 ]-------+------------------------------
n_live_tup          | 500
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_autovacuum     | 2026-07-02 17:39:00.710724+02
last_autoanalyze    | 2026-07-02 17:39:00.714655+02
```

Cette fois, c'est la colonne `n_dead_tup` qui est réinitialisée à zéro, et l'heure du dernier `VACUUM` automatique est bien 17h39.

```
-[ RECORD 1 ]-------+------
score               | 2e-08
vacuum_score        | 0
vacuum_insert_score | 0
analyze_score       | 0
doing               |
```

Les scores sont aussi revenus à zéro, la table étant traitée.

Cette nouvelle vue est très intéressante en terme de supervision manuelle comme automatique. Nous pouvons facilement savoir le nombre de tables à traiter. Par exemple :

```
SELECT count(*) FILTER (WHERE do_vacuum) AS nb_tables_pour_vacuum,
       count(*) FILTER (WHERE do_analyze) AS nb_tables_pour_analyze
FROM pg_stat_autovacuum_scores
```

Il est aussi possible de trouver la liste des tables qui seront traitées, par ordre de priorité :

```
SELECT relname, score
FROM pg_stat_autovacuum_scores
WHERE do_vacuum OR do_analyze OR for_wraparound
ORDER BY score DESC
```

6.5 NOUVELLES COLONNES POUR PG_STAT_PROGRESS_VACUUM ET PG_STAT_PROGRESS_ANALYZE



Si un VACUUM ou un ANALYZE est en cours :

- `pg_stat_progress_vacuum.mode`
    - `normal`
    - `aggressive`
    - `failsafe`
- `pg_stat_progress_vacuum/analyze.started_by`
    - `manual`
    - `autovacuum`
    - `autovacuum_wraparound`

Les catalogues statistiques `pg_stat_progress_vacuum` et `pg_stat_progress_analyze` disposent maintenant de nouvelles colonnes.

La colonne `mode` de `pg_stat_progress_vacuum`, précise le mode du traitement :

- `normal`, le cas usuel ;
- `aggressive`, pour recycler les numéros de transaction, principalement quand l'âge de la table atteint `vacuum_freeze_table_age` ;
- ou `failsafe` (exceptionnel), pour recycler les numéros de transaction en urgence, en mode dégradé.

La colonne `started_by`, dans les deux vues, indique qui a lancé ce traitement :

- `manual` correspond à un ordre VACUUM ou ANALYZE explicite, manuel, scripté ou lancé par `vacuumdb` ;
- `autovacuum` correspond à un processus lancé par le démon ;
- `autovacuum_wraparound` (dans `pg_stat_progress_vacuum`) apparaît quand le but est d'abord de recycler des numéros de transaction.

6.6 NOUVEAUTÉS DE PG_STAT_STATEMENTS



- 2 nouvelles colonnes pour les requêtes préparées :
  - `generic_plan_calls`
  - `custom_plan_calls`
- Normalise la taille des FETCH des curseurs

6.6.1 Requêtes FETCH

Pour tester cette nouvelle fonctionnalité, nous avons créé ce script SQL :

```
SELECT pg_stat_statements_reset();

BEGIN;
DECLARE pgss_cursor CURSOR FOR SELECT datname FROM pg_database;
FETCH NEXT pgss_cursor ;
FETCH 1 pgss_cursor ;
FETCH 2 pgss_cursor ;
FETCH -3 pgss_cursor ;
COMMIT;

SELECT queryid, query, calls
FROM pg_stat_statements
WHERE query ilike '%fetch%';
```

Exécuté sur une version 18, nous obtenons ce résultat :

```
       queryid        |         query          | calls
----------------------+------------------------+-------
 -2540119974538564947 | FETCH 2 pgss_cursor    |     1
  8065945752656983846 | FETCH -3 pgss_cursor   |     1
 -8919040765459638991 | FETCH NEXT pgss_cursor |     2
(3 lignes)
```

Noter que `FETCH 1` et `FETCH NEXT` sont identifiés comme la même commande, avec deux appels.

Nous voyons bien qu'il n'y a pas de normalisation dans le nombre de lignes à récupérer dans un FETCH. Ceci pose problème car cela va remplir le tableau `pg_stat_statements` dont le nombre de lignes est limité.

Exécuté sur une version 19, nous obtenons ce résultat :

```
       queryid       |         query          | calls
---------------------+------------------------+-------
 6753931942005217115 | FETCH $1 pgss_cursor   |     3
 4217710918409650319 | FETCH NEXT pgss_cursor |     1
(2 lignes)
```

Nous avons un regroupement du nombre de lignes des FETCH (dans notre exemple, 1, 2, -3).

6.6.2 Décompte des plans génériques et personnalisés

Deux nouveaux compteurs sont ajoutés à `pg_stat_statements` :

- `generic_plan_calls`
- `custom_plan_calls`

Ces compteurs tracent combien de fois une requête préparée utilise un plan générique ou un plan personnalisé.

Effectuons un test sur une requête préparée `p1` en forçant le choix des plans.

```
SELECT pg_stat_statements_reset();

PREPARE p1 AS SELECT $1 AS a;
SET plan_cache_mode TO force_generic_plan;
EXECUTE p1(1);
```

```
 a
---
 1
(1 row)
```

```
SELECT calls, generic_plan_calls, custom_plan_calls, query
FROM pg_stat_statements;
```

```
 calls | generic_plan_calls | custom_plan_calls |                    query
-------+--------------------+-------------------+-----------------------------------------------
     1 |                  1 |                 0 | PREPARE p1 AS SELECT $1 AS a
     1 |                  0 |                 0 | SELECT pg_stat_statements_reset() IS NOT NULL AS t
     1 |                  0 |                 0 | SET plan_cache_mode TO $1
(3 lignes)
```

En forcant le planificateur de requête à choisir le plan générique pour la requête préparée, on voit que le compteur `generic_plan_calls` de la requête préparée **p1** est incrementé de 1.

```
SET plan_cache_mode TO force_custom_plan;
EXECUTE p1(1);
```

```
 a
---
 1
(1 row)
```

```
SELECT calls, generic_plan_calls, custom_plan_calls, query FROM pg_stat_statements;
```

```
 calls | generic_plan_calls | custom_plan_calls |                        query
-------+--------------------+-------------------+-----------------------------------------------------------
     2 |                  1 |                 1 | PREPARE p1 AS SELECT $1 AS a
     1 |                  0 |                 0 | SELECT calls, generic_plan_calls, custom_plan_calls, query FROM pg_stat_statements
     1 |                  0 |                 0 | SELECT pg_stat_statements_reset() IS NOT NULL AS t
     2 |                  0 |                 0 | SET plan_cache_mode TO $1
(4 lignes)
```

Même constat avec le plan personnalisé, la requête préparée voit sa colonne `custom_plan_calls` passer de 0 à 1.

Cette évolution permet de savoir rapidement si une requête est replanifiée systématiquement ou bénéficie d'un plan générique, les deux cas pouvant poser des soucis différents.

6.7 CONFIGURATION DE LOG_MIN_MESSAGES PAR PROCESSUS



- Niveaux configurables par type de processus
  - archiver
  - autovacuum
  - checkpointer
  - ...
- Par ex :

```
log_min_messages = 'warning,archiver:debug1,autovacuum:debug3'
```

Avant la version 19, le niveau indiqué dans la configuration du paramètre `log_min_messages` s'appliquait uniformément à tous les processus. Cela pouvait se révéler problématique si nous souhaitions obtenir un niveau de traces plus poussé pour un processus posant problème. Donc, en version 19, il est possible d'avoir un autre comportement : un niveau de traces par type de processus. Par exemple, en se plaçant en DEBUG3 pour l'*autovacuum*, il est possible d'obtenir des traces sur le score des tables, sans avoir du DEBUG3 pour tous les processus, ce qui serait trop volumineux.

Avec ce paramétrage :

```
log_min_messages = warning,autovacuum:debug3
```

on configure un niveau `warning` pour tout le monde, et niveau `debug3` pour `autovacuum`. Nous obtenons ainsi des traces comme celles-ci :

```
DEBUG:  InitPostgres
DEBUG:  autovacuum: processing database "garmin"
DEBUG:  pg_statistic: vac: 0 (thresh 156, score 0.00), ins: 0 (thresh 1099, score 0.00), anl: 699 (thresh 103, score 0.00), xid score: 0.00,…
DEBUG:  pg_type: vac: 0 (thresh 187, score 0.00), ins: 14 (thresh 1009, score 0.01), anl: 0 (thresh 118, score 0.00), xid score: 0.00,…
DEBUG:  pg_foreign_table: vac: 0 (thresh 50, score 0.00), ins: 0 (thresh 1000, score 0.00), anl: 0 (thresh 50, score 0.00), xid score: 0.00,…
DEBUG:  pg_authid: vac: 8 (thresh 55, score 0.15), ins: 7 (thresh 1005, score 0.01), anl: 0 (thresh 52, score 0.00), xid score: 0.00, mxid…
[...]
```

Ce qui nous donne les scores par table à chaque vérification d'une base.

Dans l'exemple plus haut, `archiver:debug1` est pratique pour voir explicitement l'archivage des journaux :

```
DEBUG:  archived write-ahead log file "000000010000001900000003B"
DEBUG:  archived write-ahead log file "000000010000001900000003C"
DEBUG:  archived write-ahead log file "000000010000001900000003D"
DEBUG:  archived write-ahead log file "000000010000001900000003E"
```

```
DEBUG:  archived write-ahead log file "000000010000000190000003F"
DEBUG:  archived write-ahead log file "000000010000001900000040"
```

Avec `archiver:debug3`, on voit les commande exactes de l'archivage :

```
DEBUG:  executing archive command "cp pg_wal/000000010000001900000092
↪  /var/lib/postgresql/walarchive/"
DEBUG:  archived write-ahead log file "000000010000001900000092"
DEBUG:  executing archive command "cp pg_wal/000000010000001900000093
↪  /var/lib/postgresql/walarchive/"
DEBUG:  archived write-ahead log file "000000010000001900000093"
```

La liste des processus est sur https://www.postgresql.org/docs/19/runtime-config-logging.html#GUC-LOG-MIN-MESSAGES.

Attention, tout cela risque de prendre beaucoup de place dans les traces.

L'ancien comportement est cependant toujours disponible.

6.8 NOUVEAU PARAMÈTRE LOG_AUTOANALYZE_MIN_DURATION



- Avant :
  - `log_autovacuum_min_duration` : trace autovacuum et autoanalyze
- À partir de la v19 :
  - `log_autovacuum_min_duration` : trace autovacuum
  - `log_autoanalyze_min_duration` : trace autoanalyze

Avant la version 19, configurer `log_autovacuum_min_duration` permettait de tracer les traitements VACUUM et ANALYZE d'un `autovacuum worker`.

À partir de la version 19, `log_autovacuum_min_duration` ne concerne que les traitements VACUUM. `log_autoanalyze_min_duration` a été ajouté pour tracer les traitements ANALYZE.

Cela permet de mieux gérer la quantité d'informations envoyées dans les fichiers de trace.

6.9 ACTIVATION DE LOG_LOCK_WAITS PAR DÉFAUT



- `log_lock_waits = on` par défaut

Ce paramètre était à `off` par défaut depuis son intégration. Cependant, il a été déclaré suffisamment utile pour être activé par défaut. En effet, si une requête attend un verrou plus d'une seconde, c'est une information importante à connaître.

6.10 VACUUM VERBOSE : AJOUT DU NOMBRE D'OCTETS DES FPI



```
VACUUM (VERBOSE) table1 ;
WAL usage: 54654 records, 6 full page images, 11895327 bytes,
47596 full page image bytes, 0 buffers full
```

- Ajoute aussi le nombre d'octets écrits pour des FPI
  - écriture dans les WAL
  - FPI = *Full Page Image*
- Concerne aussi ANALYZE

Voici toutes les informations remontées par l'option VERBOSE de la commande VACUUM pour une table sans index :

```
INFO:  vacuuming "demo.public.t1"
INFO:  finished vacuuming "demo.public.t1": index scans: 0
pages: 0 removed, 3 remain, 3 scanned (100.00% of total), 0 eagerly scanned
tuples: 0 removed, 500 remain, 0 are dead but not yet removable
removable cutoff: 188125, which was 0 XIDs old when operation ended
new relfrozenxid: 188125, which is 2892 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 0 pages set all-visible, 0 pages set all-frozen (0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
↪  identifiers removed
delay time: 0.000 ms
I/O timings: read: 12.086 ms, write: 0.000 ms
avg read rate: 4.338 MB/s, avg write rate: 0.620 MB/s
buffer usage: 15 hits, 7 reads, 1 dirtied
WAL usage: 1 records, 1 full page images, 4463 bytes, 4344 full page image bytes, 0
↪  buffers full
memory usage: dead item storage 0.02 MB accumulated across 0 resets (limit 64.00 MB
↪  each)
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
VACUUM
```

La partie modifiée correspond à la ligne `WAL usage`. L'information ajoutée est le `full page image bytes`. Il est ainsi possible de savoir combien d'octets font partie des FPI, et donc si on enregistre plus de FPI que d'enregistrements standards.

Ce nouvel affichage montre l'intérêt de l'activation du paramètre `wal_compression`. Il permet donc de calculer un taux de compression :

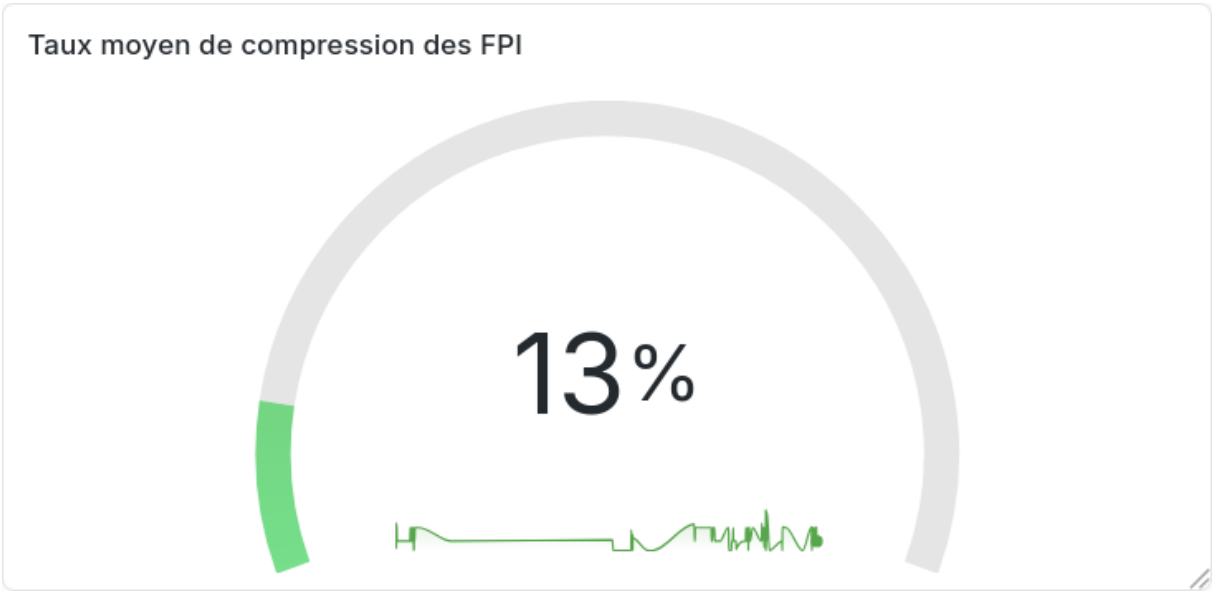


Figure 6/ .1: Taux de compression

6.11 VACUUM VERBOSE : AJOUT DE LA CONSOMMATION MÉMOIRE ET DE LA PARALLÉLISATION



```
memory usage: dead item storage 1.20 MB accumulated across 1 reset
              (limit 64.00 MB each)
parallel workers: index vacuum: 3 planned, 3 launched in total
```

- `memory usage` : utilisation de la mémoire
  - espace nécessaire pour les éléments à nettoyer, le nombre de réinitialisations de cet espace, et la limite de cet espace
- `parallel workers` : utilisation de la parallélisation
  - indique le nombre de workers planifiés et lancés

6.11.1 Utilisation de la mémoire

Le VACUUM utilise de la mémoire, à hauteur de la configuration du paramètre `maintenance_work_mem` (ou `autovacuum_work_mem` en cas d'un lancement par l'*autovacuum*). Cet espace mémoire sert à stocker les éléments à supprimer dans les index de la table traitée. Si cet espace n'est pas suffisant, il traitera les index pour les éléments indiqués en mémoire, puis réinitialisera l'espace, continuera la lecture de la table pour trouver les autres éléments, et traitera de nouveau les index. Avoir un `maintenance_work_mem` d'une taille insuffisante générera plusieurs lectures des mêmes index, ce qui est une perte de temps. Cette ligne supplémentaire permet, non seulement, de s'en apercevoir, mais de connaître aussi la quantité de mémoire nécessaire.

Voici un exemple de trace pour un `maintenance_work_mem` à 64 Mo (sa valeur par défaut).

Commençons par créer un jeu de tests :

```
CREATE TABLE t1(c1 integer);
ALTER TABLE t1 SET (autovacuum_enabled = false);
INSERT INTO t1 SELECT generate_series(1, 10_000_000);
CREATE INDEX ON t1(c1);
```

Exécutons un VACUUM (VERBOSE) sur cette table. Voici les traces générés :

```
INFO:  vacuuming "demo.public.t1"
INFO:  finished vacuuming "demo.public.t1": index scans: 0
pages: 0 removed, 44248 remain, 44248 scanned (100.00% of total), 0 eagerly scanned
tuples: 0 removed, 10000000 remain, 0 are dead but not yet removable
```

```
removable cutoff: 14953, which was 0 XIDs old when operation ended
new relfrozenxid: 14953, which is 6 XIDs ahead of previous value
frozen: 44248 pages from table (100.00% of total) had 10000000 tuples frozen
visibility map: 44248 pages set all-visible, 44248 pages set all-frozen (0 were
 ↪  all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
 ↪  identifiers removed
delay time: 0.000 ms
I/O timings: read: 19.612 ms, write: 111.554 ms
avg read rate: 140.036 MB/s, avg write rate: 187.218 MB/s
buffer usage: 60165 hits, 28371 reads, 37930 dirtied
WAL usage: 44254 records, 44255 full page images, 363981706 bytes, 361105024 full
 ↪  page image bytes, 15053 buffers full
memory usage: dead item storage 0.02 MB accumulated across 0 resets (limit 64.00 MB
 ↪  each)
system usage: CPU: user: 0.81 s, system: 0.36 s, elapsed: 1.58 s
```

En ligne 2, on voit `index scans: 0`. Cela nous indique qu'aucun parcours d'index n'a été réalisé. C'est étonnant vu qu'il y a bien un index sur cette table. La raison est indiquée un peu plus bas :

```
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
 ↪  identifiers removed
```

La phase de parcours des index a été jugée inutile vu qu'aucune ligne n'est morte depuis le dernier VACUUM (dans notre cas, depuis la création de la table).

De ce fait, l'utilisation mémoire est très faible :

```
memory usage: dead item storage 0.02 MB accumulated across 0 resets (limit 64.00 MB
 ↪  each)
```

`0 resets`, donc 0 parcours d'index. La limite est à 64 Mo, tout comme le paramètre `maintenance_work_mem` sur cette instance.

Créons quelques enregistrements morts :

```
DELETE FROM t1 WHERE c1<2_000_000;
```

Et relançons un `VACUUM (VERBOSE)` :

```
INFO:  vacuuming "demo.public.t1"
INFO:  finished vacuuming "demo.public.t1": index scans: 1
pages: 0 removed, 44248 remain, 8851 scanned (20.00% of total), 0 eagerly scanned
tuples: 1999999 removed, 7999963 remain, 0 are dead but not yet removable
removable cutoff: 14955, which was 0 XIDs old when operation ended
new relfrozenxid: 14955, which is 2 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 8850 pages set all-visible, 8850 pages set all-frozen (0 were
 ↪  all-visible)
```

```
index scan needed: 8850 pages from table (20.00% of total) had 1999999 dead item
 ↪  identifiers removed
index "t1_c1_idx": pages: 27422 in total, 5483 newly deleted, 5483 currently
 ↪  deleted, 0 reusable
delay time: 0.000 ms
I/O timings: read: 4.076 ms, write: 0.000 ms
avg read rate: 433.030 MB/s, avg write rate: 137.893 MB/s
buffer usage: 64896 hits, 21954 reads, 6991 dirtied
WAL usage: 35188 records, 12163 full page images, 66136298 bytes, 54409416 full page
 ↪  image bytes, 6945 buffers full
memory usage: dead item storage 1.20 MB accumulated across 1 reset (limit 64.00 MB
 ↪  each)
system usage: CPU: user: 0.30 s, system: 0.06 s, elapsed: 0.39 s
```

Cette fois, il y a bien eu un parcours d'index (`index scans: 1` en ligne 2). L'opération a bien vu qu'il y avait 199 999 lignes mortes dans la table. L'utilisation mémoire est donc plus importante que la dernière fois :

```
memory usage: dead item storage 1.20 MB accumulated across 1 reset (limit 64.00 MB
 ↪  each)
```

1,2 Mo ont été nécessaires. Notre configuration de `maintenance_work_mem` est suffisante.

Maintenant, diminuons la quantité de mémoire et augmentons le nombre de lignes mortes :

```
SET maintenance_work_mem TO '1MB';
TRUNCATE t1;
INSERT INTO t1 SELECT generate_series(1, 10_000_000);
DELETE FROM t1 WHERE c1<4_000_000;
```

Voici ce que nous donnent les traces d'un `VACUUM (VERBOSE)` :

```
INFO:  vacuuming "demo.public.t1"
INFO:  finished vacuuming "demo.public.t1": index scans: 2
pages: 0 removed, 44248 remain, 44248 scanned (100.00% of total), 0 eagerly scanned
tuples: 3999999 removed, 6000001 remain, 0 are dead but not yet removable
removable cutoff: 14968, which was 0 XIDs old when operation ended
new relfrozenxid: 14965, which is 1 XIDs ahead of previous value
frozen: 26548 pages from table (60.00% of total) had 5999800 tuples frozen
visibility map: 44248 pages set all-visible, 44247 pages set all-frozen (0 were
 ↪  all-visible)
index scan needed: 17700 pages from table (40.00% of total) had 3999999 dead item
 ↪  identifiers removed
index "t1_c1_idx": pages: 27421 in total, 10966 newly deleted, 10966 currently
 ↪  deleted, 0 reusable
delay time: 0.000 ms
I/O timings: read: 30.290 ms, write: 192.317 ms
avg read rate: 261.319 MB/s, avg write rate: 153.115 MB/s
buffer usage: 138410 hits, 88348 reads, 51766 dirtied
```

```
WAL usage: 124776 records, 45384 full page images, 367750875 bytes, 325862404 full
↪  page image bytes, 3632 buffers full
memory usage: dead item storage 1.58 MB accumulated across 2 resets (limit 1.00 MB
↪  each)
system usage: CPU: user: 1.46 s, system: 0.48 s, elapsed: 2.64 s
```

Cette fois, il y a eu deux parcours d'index (`index scans: 2` en ligne 2). Ceci indique que la mémoire n'a pas été suffisante, ce que confirme l'avant-dernière ligne :

```
memory usage: dead item storage 1.58 MB accumulated across 2 resets (limit 1.00 MB
↪  each)
```

1,58 Mo auraient été nécessaires. Avec un `maintenance_work_mem` à 1 Mo, il a donc fallu deux parcours dans l'index.

Cette nouvelle information va fortement aider à une meilleure configuration du paramètre `maintenance_work_mem`.

6.11.2 Utilisation de la parallélisation

L'option VERBOSE ajoute une ligne sur la parallélisation de l'opération. Pour rappel, un VACUUM peut utiliser la parallélisation pour le traitement des index. Cela signifie donc que la table doit avoir plus d'un index pour que cette parallélisation soit effective. Prenons un exemple :

```
SET maintenance_work_mem TO '1MB';
TRUNCATE t1;
INSERT INTO t1 SELECT generate_series(1, 10_000_000);
CREATE INDEX ON t1(c1);
DELETE FROM t1 WHERE c1<4_000_000;
```

Voici les traces d'un `VACUUM (VERBOSE, PARALLEL 2)` :

```
INFO:  vacuuming "demo.public.t1"
INFO:  launched 1 parallel vacuum worker for index vacuuming (planned: 1)
INFO:  launched 1 parallel vacuum worker for index vacuuming (planned: 1)
INFO:  launched 1 parallel vacuum worker for index vacuuming (planned: 1)
INFO:  finished vacuuming "demo.public.t1": index scans: 3
pages: 0 removed, 44248 remain, 44248 scanned (100.00% of total), 0 eagerly scanned
tuples: 4999773 removed, 5000227 remain, 0 are dead but not yet removable
tuples missed: 226 dead from 1 pages not removed due to cleanup lock contention
removable cutoff: 15099, which was 0 XIDs old when operation ended
new relfrozenxid: 15094, which is 3 XIDs ahead of previous value
frozen: 22124 pages from table (50.00% of total) had 4999976 tuples frozen
visibility map: 44247 pages set all-visible, 44246 pages set all-frozen (0 were
↪  all-visible)
index scan needed: 22123 pages from table (50.00% of total) had 4999773 dead item
↪  identifiers removed
```

```
parallel workers: index vacuum: 3 planned, 3 launched in total
index "t1_c1_idx": pages: 27421 in total, 13705 newly deleted, 13705 currently
 ↪  deleted, 0 reusable
index "t1_c1_idx1": pages: 27422 in total, 13706 newly deleted, 13706 currently
 ↪  deleted, 0 reusable
delay time: 0.000 ms
I/O timings: read: 36.334 ms, write: 200.897 ms
avg read rate: 549.636 MB/s, avg write rate: 198.512 MB/s
buffer usage: 244969 hits, 199709 reads, 72129 dirtied
WAL usage: 157077 records, 70636 full page images, 507361893 bytes, 462598116 full
 ↪  page image bytes, 2654 buffers full
memory usage: dead item storage 3.50 MB accumulated across 3 resets (limit 1.00 MB
 ↪  each)
system usage: CPU: user: 1.51 s, system: 0.50 s, elapsed: 2.83 s
```

La nouvelle ligne est la suivante :

```
parallel workers: index vacuum: 3 planned, 3 launched in total
```

7/ Réplication

7.1 RÉPLICATION LOGIQUE DES SÉQUENCES



CREATE/ALTER PUBLICATION … FOR ALL SEQUENCES

- Enfin la synchronisation des séquences
  - si source en v19+
- Pas de granularité : toutes les séquences ou aucune
- Synchronisation à trois moments :

```
CREATE SUBSCRIPTION …
ALTER SUBSCRIPTION … REFRESH PUBLICATION -- méta-données
ALTER SUBSCRIPTION … REFRESH SEQUENCES   -- données
```

- Nouveau worker : `sequencesync worker`

La réplication logique réplique toutes les modifications des données des tables. Cependant, les séquences n'étaient pas prises en compte. La version 19 corrige ce manque.

Pour l'instant, il est seulement possible de synchroniser toutes les séquences ou aucune. Il n'y a donc pas de granularité comme pour les tables. Les commandes `CREATE PUBLICATION` et `ALTER PUBLICATION` se voient donc complétées d'une seule clause : `FOR ALL SEQUENCES`.

La synchronisation des séquences peut se faire lors de ces trois moments :

- Lors de l'exécution d'un `CREATE SUBSCRIPTION`. Dans ce cas, l'abonné récupère la liste des séquences associées avec les publications. Les séquences publiées sont ajoutées au catalogue système `pg_subscription_rel` dans l'état `INIT`. Enfin, l'abonné lance le worker `sequencesync` pour synchroniser toutes les séquences publiées.
- Lors de l'exécution d'un `ALTER SUBSCRIPTION … REFRESH PUBLICATION`. Les séquences supprimées sur le publieur sont supprimées de la table `pg_subscription_rel` alors que les nouvelles séquences sont ajoutées dans l'état `INIT`. Enfin, l'abonné lance le worker `sequencesync` pour synchroniser uniquement les séquences nouvellement ajoutées.
- Lors de l'exécution d'un `ALTER SUBSCRIPTION … REFRESH SEQUENCES`. Cette nouvelle clause va placer toutes les séquences listées dans la table `pg_subscription_rel` dans l'état `INIT`. L'abonné lance le worker `sequencesync` pour synchroniser toutes les séquences. Il n'y a donc pas de vérification de séquences ajoutées ou supprimées de la publication.

Voici un exemple de son utilisation. Le serveur publieur est sur le port 5519, le serveur abonné est sur le port 5432. Pour simplifier les choses, je vais utiliser deux services, configurés dans mon fichier `pg_service.conf` ainsi :

```
[publieur]
port=5519
user=postgres
dbname=demo

[abonne]
host=/tmp
port=5432
user=guillaume
dbname=demo
```

J'exécute le script ci-dessous sur le publieur :

```
CREATE TABLE employes (
  id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  matricule text UNIQUE NOT NULL,
  nom text);
```

Il faut récupérer la définition des objets de la base démo du futur publieur sur la base démo du futur abonné :

```
$ pg_dump --no-owner "service=publieur" | psql "service=abonne"

SET
SET
SET
SET
SET
SET
┌────────────┐
│ set_config │
├────────────┤
│            │
└────────────┘
(1 row)

SET
SET
SET
SET
SET
SET
CREATE TABLE
```

```
ALTER TABLE
COPY 0
```

```
┌────────┐
│ setval │
├────────┤
│      1 │
└────────┘
```

```
(1 row)

ALTER TABLE
ALTER TABLE
```

Maintenant, insérons quelques données sur le futur publieur :

```
$ psql -X "service=publieur"
```

```
psql (19.0)
Type "help" for help.
```

```
demo=# SELECT * FROM employes;
```

```
 id | matricule | nom
----+-----------+-----
(0 rows)
```

```
demo=# INSERT INTO employes (matricule, nom) VALUES ('P001', 'Guillaume'), ('P002',
↪  'Christophe') RETURNING *;
```

```
 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
(2 rows)

INSERT 0 2
```

Créons la publication pour toutes les tables et toutes les séquences :

```
CREATE PUBLICATION pub1 FOR ALL TABLES, ALL SEQUENCES;
```

Sur l'abonné, souscrivons à cet abonnement :

```
$ psql -X "service=abonne"
```

```
psql (19.0)
Type "help" for help.
```

```
demo=# SELECT * FROM employes;
```

```
 id | matricule | nom
----+-----------+-----
(0 rows)
```

```
demo=# CREATE SUBSCRIPTION sub1 CONNECTION 'port=5519 dbname=demo' PUBLICATION pub1;

NOTICE:  created replication slot "sub1" on publisher
CREATE SUBSCRIPTION

demo=# SELECT * FROM employes;

 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
(2 rows)

demo=# SELECT * FROM employes_id_seq ;

 last_value | log_cnt | is_called
------------+---------+-----------
          2 |       0 | t
(1 row)
```

Nous pouvons voir ici que les données de la table et de la séquence ont été synchronisées un peu après l'abonnement.

Maintenant, ajoutons une nouvelle ligne dans cette table :

```
$ psql -X "service=publieur"

psql (19.0)
Type "help" for help.

demo=# INSERT INTO employes (nom, matricule) VALUES ('Benoît', 'P004') RETURNING *;

 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
  3 | P004      | Benoît
(3 rows)
INSERT 0 1

demo=# SELECT * FROM employes_id_seq;

 last_value | log_cnt | is_called
------------+---------+-----------
          3 |      32 | t
(1 row)
```

Sur le publieur, nous voyons la ligne ajoutée et la séquence incrémentée.

Que voyons-nous sur l'abonné ?

```
$ psql -X "service=abonne"
```

```
psql (19.0)
Type "help" for help.

demo=# SELECT * FROM employes;

 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
  3 | P004      | Benoît
(3 rows)

demo=# SELECT * FROM employes_id_seq;

 last_value | log_cnt | is_called
------------+---------+-----------
          2 |       0 | t
(1 row)
```

La ligne a bien été ajoutée sur l'abonné mais la séquence est restée à son ancienne valeur. En fait, il faut demander son rafraîchissement :

```
demo=# ALTER SUBSCRIPTION sub1 REFRESH SEQUENCES;

ALTER SUBSCRIPTION

demo=# SELECT * FROM employes_id_seq;

 last_value | log_cnt | is_called
------------+---------+-----------
          3 |       0 | t
(1 row)
```

Cette nouveauté sur les séquences en réplication logique facilitera fortement les copies de bases entières et les montées de version par réplication logique.

Évidemment, cela ne fonctionne que si la source de la réplication est en version 19. Les migrations par réplication logique depuis des versions antérieures continueront à devoir synchroniser manuellement les séquences.

7.2 AUTRES NOUVEAUTÉS EN RÉPLICATION LOGIQUE



- `CREATE/ALTER PUBLICATION … EXCEPT table1`
  - exclure certaines tables
- `CREATE SUBSCRIPTION … SERVER …`
  - utilise un foreign server (`postgres_fdw`)
  - mutualise la définition des abonnements
- `wal_level` : passage automatique à `logical`
  - quand apparaît un slot de réplication logique
  - niveau actif : `effective_wal_level`

7.2.1 Exclusion de table

Les instructions `CREATE/ALTER PUBLICATION` permettent de sélectionner toutes les tables d'un coup ou de sélectionner uniquement les tables nécessaires. S'il y a 390 tables qui nous intéressent sur les 400 de la base, il fallait citer les 390 tables individuellement.

À partir de la version 19, il est possible d'utiliser la clause `EXCEPT` pour noter toutes celles qui ne nous intéressent pas. Dans l'exemple précédent, cela revient à nommer uniquement 10 tables au lieu de 390.

7.2.2 Utilisation d'un SERVER comme source de publication

La création d'une souscription nécessite d'indiquer la chaîne de connexion au serveur publieur. La version 19 permet de remplacer la chaîne de connexion par le nom d'un *Foreign Server* du *Foreign Data Wrapper* 'postgres_fdw. C'est d'autant plus utile si nous avons plusieurs abonnements à réaliser vers le même serveur publieur.

Si nous reprenons l'exemple précédent, en utilisant un *Foreign Server*, cela donnerait :

```
-- un peu de nettoyage
demo=# DROP SUBSCRIPTION sub1;

NOTICE:  dropped replication slot "sub1" on publisher
DROP SUBSCRIPTION

demo=# TRUNCATE employes;
```

```
TRUNCATE TABLE

-- ajout de l'extension postgres_fdw
demo=# CREATE EXTENSION postgres_fdw;

CREATE EXTENSION

-- création du serveur "publieur"
-- même paramètres de connexion que précédemment, mais sous un autre format
demo=# CREATE SERVER publieur FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (port '5519', dbname 'demo');

CREATE SERVER

-- création d'un user mapping
-- l'utilisateur guillaume va se connecter en tant que postgres sur le serveur
 ↪  publieur
demo=# CREATE USER MAPPING FOR guillaume SERVER publieur OPTIONS (user 'postgres',
 ↪  password 'oulala');

CREATE USER MAPPING

-- création de la souscription
-- en remplaçant CONNECTION par SERVER
demo=# CREATE SUBSCRIPTION sub1 SERVER publieur PUBLICATION pub1;

NOTICE:  created replication slot "sub1" on publisher
CREATE SUBSCRIPTION

-- l'abonnement est bien créé, les données synchronisées
demo=# SELECT * FROM employes;

 id | matricule |    nom
----+-----------+------------
  1 | P001      | Guillaume
  2 | P002      | Christophe
  3 | P004      | Benoît
(3 rows)
```

7.2.3 Changement automatique de wal_level

Pour utiliser la réplication logique, il est nécessaire de configurer le paramètre `wal_level` à la valeur `logical`.

À partir de la version 19, ce passage se fait automatiquement si la valeur précédente est `replica`. La bascule se fait quand apparaît un slot de réplication logique, ou quand le dernier disparaît.

Le paramètre `wal_level` indique toujours la configuration du fichier `postgresql.conf`. Pour connaître la valeur réelle, il faut interroger le nouveau paramètre `effective_wal_level` (en lecture seule).

Ce changement est intéressant car il permet de passer d'un `wal_level` à l'autre sans redémarrer PostgreSQL, donc sans arrêt de service. Son caractère automatique permet de limiter l'augmentation des écritures dans les WAL inhérente à la réplication logique aux périodes où elle est réellement utilisée, et ce sans intervention humaine.

`REPACK (CONCURRENTLY)` utilise d'ailleurs transitoirement la réplication logique. `effective_wal_level` passe à `logical` le temps de l'opération.

7.3 NOUVELLE COMMANDE : WAIT FOR



```
WAIT FOR LSN '0/042AB070'
     WITH (MODE 'standby_replay', TIMEOUT '2s', NO_THROW);
```

- Sur un secondaire, permet d'attendre un LSN désigné
- 4 modes :
  - `primary_flush` (primaire)
  - `standby_write`
  - `standby_flush`
  - `standby_replay`
- Timeout optionnel

La nouvelle commande `WAIT FOR` (extension PostgreSQL au standard SQL) permet d'attendre que l'emplacement dans le flux de journaux (LSN) désigné soit atteint dans le mode désiré (écrit, rejoué…).

```
WAIT FOR LSN 'lsn'
    [ WITH ( option [, ...] ) ]

option:

    MODE 'mode'
    TIMEOUT 'timeout'
    NO_THROW

mode:

    standby_replay | standby_write | standby_flush | primary_flush
```

Le mode `primary_flush` ne fonctionne que sur une instance primaire et permet d'attendre que les données soient bien écrites sur disques avant de continuer. Cela peut être utile quand `synchronous_commit` est configuré à `off`.

Les modes `standby_write`, `standby_flush` et `standby_replay` doivent être exécutés sur une instance secondaire. Ils permettent d'attendre respectivement que les données atteignent la mémoire de la standby, qu'elles soient enregistrées sur disque, ou qu'elles soient rejouées et visibles.

Imaginons un traitement qui fait des insertions sur une instance primaire avant d'exécuter des requêtes longues sur l'instance secondaire et doit s'assurer que les données insérées sont prises en

compte avant de les requêter.

Après l'insertion, le LSN peut être récupéré sur la primaire avec la requête :

```
SELECT pg_current_wal_lsn();
```

```
 pg_current_wal_lsn
--------------------
 0/042AB070
(1 row)
```

La commande suivante peut ensuite être exécutée sur l'instance secondaire pour attendre que le LSN `0/042AB070` soit rejoué. Elle attend au maximum 2 secondes avant de rendre la main et n'émet pas d'erreur en cas de timeout (`NO_THROW`). Le résultat est communiqué dans la colonne `status` :

```
WAIT FOR LSN '0/042AB070' WITH (MODE 'standby_replay', TIMEOUT '2s', NO_THROW);
```

```
 status
---------
 success
(1 row)
```

En cas de timeout, la colonne `status` contient `timeout`. Si `NO_THROW` est omis, l'erreur suivante est émise.

```
ERROR:  timed out while waiting for target LSN 0/042AB070 to be replayed; current
↪  standby_replay LSN 0/022AB070
```

Précédemment, pour garantir que les données étaient bien enregistrées par un serveur secondaire, il fallait utiliser la réplication synchrone avec l'option `remote_apply`, ce qui impactait les performances globalement. On pouvait aussi implémenter l'équivalent de la commande `WAIT_FOR` programmatiquement.

8/ Divers

8.1 NOUVEAUTÉS POUR LES MODULES CONTRIB



- Extension pg_buffercache
  - nouvelle vue pg_buffercache_os_pages
  - nouvelle fonction pg_buffercache_os_pages()
  - nouvelles fonctions pg_buffercache_mark_dirty*
- Extension file_fdw
  - gestion des entrées avec plusieurs lignes d'en-tête
- Librairie auto_explain
  - nouveau paramètre auto_explain.log_io
  - nouveau paramètre auto_explain.log_extension_options
- Outil oid2name
  - nouvelle option --extended

8.1.1 pg_buffercache

C'est principalement l'extension pg_buffercache qui a bénéficié de quelques nouveautés.

La vue pg_buffercache contient une ligne pour chaque bloc du cache de PostgreSQL (*buffer*). La vue pg_buffercache_os_pages permet d'associer une page mémoire de l'OS à des blocs précis.

Par exemple, avec les *huge pages* activées, et shared_buffers à seulement 128 Mo, le cache de PostgreSQL utilise 65 pages mémoire de 2 Mo, chacune avec 256 blocs de 8 ko :

```
SELECT os_page_num, count(*) FROM pg_buffercache_os_pages GROUP BY 1 ORDER BY 1;
```

```
 os_page_num | count
-------------+-------
           0 |   194
           1 |   256
           2 |   256
           3 |   256
           4 |   256
           5 |   256
           6 |   256
           7 |   256
```

```
           8 |   256
…
          62 |   256
          63 |   256
          64 |    62
```

Sans *huge pages*, on aurait au contraire deux pages OS de 4 ko par bloc de PostgreSQL :

```
SELECT bufferid, count(*) FROM pg_buffercache_os_pages GROUP BY 1 ORDER BY 1 LIMIT 5;
```

La fonction `pg_buffercache_os_pages()` permet de consulter aussi le nœud NUMA.

Il existe à présent plusieurs fonctions pour marquer des buffers comme modifiés (*dirty*) :

- `pg_buffercache_mark_dirty()` pour marquer le buffer indiqué ;
- `pg_buffercache_mark_dirty_relation()` pour marquer tous les buffers de la relation indiquée ;
- `pg_buffercache_mark_dirty_all()` pour marquer tous les buffers du cache.

8.1.2 file_fdw

L'extension `file_fdw` a été modifiée pour profiter des améliorations de la commande `COPY`. De même pour la librairie `auto_explain` qui tire profit des améliorations de la commande `EXPLAIN`.

8.1.3 auto_explain

Le nouveau paramètre `auto_explain.log_io` est le pendant du nouveau paramètre `IO` de `EXPLAIN`.

Le nouveau paramètre `auto_explain.log_extension_options` est destiné à afficher la sortie des modules qui peuvent étendre `EXPLAIN` comme `pg_plan_advice` ou `pg_overexplain`.

8.1.4 oid2name

Le petit outil `oid2name` sert à faire le lien entre un objet et son identifiant interne. L'option `--extended` rajoute le chemin de l'objet sous le PGDATA.

```
$ /usr/lib/postgresql/19/bin/oid2name -d pgbench -t pgbench_accounts
From database "pgbench":
  Filenode         Table Name
----------------------------
   2647046  pgbench_accounts
```

```
$ /usr/lib/postgresql/19/bin/oid2name -d pgbench -t pgbench_accounts --extended
From database "pgbench":
  Filenode         Table Name      Oid  Schema  Tablespace                  Path
--------------------------------------------------------------------------------
   2647046  pgbench_accounts  2646943  public  pg_default  base/2638366/2647046
```

9/ Ce qui ne sera pas là

9.1 LA PROCHAINE FOIS PEUT-ÊTRE ?



Ce qui vous ne verrez *pas* cette année :

- `MERGE/SPLIT` de partitions
- `GROUP BY ALL`
- SQL Property Graph Queries (SQL/PGQ)
- `UPDATE/DELETE FOR PORTION OF`
- Activation à chaud des checksums
- `pg_dumpall` binaire

Sans doute dans PostgreSQL 20 ?

Il n'est pas rare qu'une fonctionnalité présente dans une bêta de PostgreSQL soit finalement retirée au dernier moment. PostgreSQL étant extrêmement complexe, les interactions entre les différents composants (par exemple la sauvegarde, les *row level securities*, la réplication logique…) ne sont pas toutes détectables rapidement.

Si un bug est détecté pendant la phase bêta, il suffit parfois d'ajouter une limitation ou une restriction, qui sera levée dans une version future. Mais si des bugs sérieux ont été trouvés trop tard, et ne peuvent être corrigés sans modifications d'envergure, la fonctionnalité peut être annulée (*revert*). Les « rustines » dans l'urgence, qui ne résolvent pas vraiment le souci, ne sont pas acceptables, surtout quand elles impliquent une charge future de maintenance.

En général, la fonctionnalité refusée est revue et livrée l'année suivante.

La *Release Management Team*[1] a le dernier mot sur ce que contiendra la version à venir. Mais les développeurs sont assez responsables pour retirer d'eux-même leur patch quand la situation l'exige.

PostgreSQL 19 avait été très ambitieuse sur les nouvelles fonctionnalités. En parallèle, les analyses automatiques par LLM sont récemment devenues plus faciles et plus justes, et repèrent plus de bugs. Robert Haas a lancé le sujet dans le fil *Scary patch contest*[2].

Ainsi, lors de la phase bêta, les développeurs ont déclaré certaines nouvelles fonctionnalités comme insuffisamment stables, malgré un intérêt certain.

- C'est le cas pour la syntaxe de graphes SQL/PGQ, qui aurait été très utile dans certains cas. Il y

[1] https://wiki.postgresql.org/wiki/Release_Management_Team
[2] https://postgresweekly.com/link/190039/92230c196b

avait malheureusement de nombreux soucis[3], notamment dans la manière de les sauvegarder, et le stockage des métadonnées. C'était sa première tentative d'intégration dans PostgreSQL. Les développements vont là aussi continuer pour une intégration finale, probablement en version 20.

- Les fusions et scissions de partitions auraient facilité la vie des administrateurs, mais il y avait encore des problèmes[4], à commencer par l'interaction avec la réplication logique ou les colonnes générées, ou encore le schéma dont hérite les nouvelles partitions. C'est la deuxième fois que la fonctionnalité est acceptée puis retirée, mais les développeurs sont suffisamment intéressés pour continuer le travail.
- La syntaxe `GROUP BY ALL`, pourtant pratique, a également été annulée[5], notamment parce qu'un bug est apparu, dont la correction impliquerait un *refactoring* d'un code assez sensible, ce qui devra attendre la prochaine version de PostgreSQL.
- L'activation à chaud des sommes de contrôle a aussi été renvoyée à une autre version[6]. Elle aurait permis de les utiliser plus largement sur les bases migrées physiquement. Il y avait eu beaucoup de correctifs pour une fonctionnalité qui touche directement aux fichiers de données, et il restait notamment encore un souci lié aux fichiers orphelins éventuels[7].
- A aussi été reportée la possibilité pour `pg_dumpall` de sauvegarder dans les formats binaires (custom, tar, directory), comme `pg_dump`. L'annulation s'est faite sur des critères techniques dans le code[8]. Certaines des améliorations techniques faites en cours de routes sont cependant préservées[9].
- Il ne sera toujours pas possible d'obtenir le DDL d'un objet avec une fonction : le code de `pg_get_role_ddl()` et d'autres fonctions équivalentes faisait doublon avec celui de `pg_dump`/`pg_dumpall`, ce qui n'était pas acceptable[10]. Obtenir un code commun pour les deux fonctionnalités semble aussi très compliqué.

Il y a bon espoir de retrouver ces nouveautés pleinement fonctionnelles dans PostgreSQL 20.

---

[3] https://www.postgresql.org/message-id/flat/dqa5mstx5mna3i7s23pdwl4m6bek7gqsgfccef44wjpswizufi%403aa6vzri3cat
[4] https://www.postgresql.org/message-id/CAN4CZFNCU%3Dt09M%3D%2Br2t9hHLJuujdM4oQ8hCK_Sx-GpfiwMAicw%40mail.gmail.com
[5] https://postgr.es/m/5243308F-8E5C-45AA-828C-FAD96C4F34DA@gmail.com
[6] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=c05d5ce12366f10
[7] https://www.postgresql.org/message-id/flat/E15AC050-C4B5-488D-BB2D-3C7AC9F89EA8%40yesql.se
[8] https://www.postgresql.org/message-id/20260607000218.96.noahmisch@microsoft.com
[9] https://postgr.es/c/7ca548f23a60887541493b2da47f2b2720c35ee3
[10] https://www.postgresql.org/message-id/20260827015242.54.noahmisch@microsoft.com

10/ Questions



Merci de votre écoute !
Nouveautés de la version 19 :
https://dali.bo/workshop19_html
https://dali.bo/workshop19_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.

11/ 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.