Workshop 17

Nouveautés de PostgreSQL 17

17

Contents

0.1 Introduction 1

1/ Utilisation **3**

1.1 Fonction `JSON_TABLE()` 4
1.2 Fonction de conversions entre types JSON 7
1.3 Fonctions `JSON()`, `JSON_SCALAR()` et `JSON_SERIALIZE()` 8
1.4 Fonctions `JSON_EXISTS()`, `JSON_QUERY()` et `JSON_VALUE()` 10
1.5 Nouvelles possibilités de COPY 14
1.6 `INSERT`, colonne `IDENTITY` et partitionnement 17
1.7 Modification de l'expression d'une colonne générée 20
1.8 Génération de nombres aléatoires 22
1.9 Type `interval` et valeur `infinity` 23
1.10 Contraintes d'exclusion sur les tables partitionnées 25

2/ Administration **27**

2.1 Nouveau fournisseur de collation 28
2.2 Nouveau rôle pg_maintain et droit MAINTAIN 30
2.3 Résumés des WAL 33
2.4 Options –index et –jobs sur reindexdb 36
2.5 Option –all avec vacuumdb et reindexdb 37
2.6 Support des *triggers* sur REINDEX 38
2.7 Support des *triggers* sur connexions 40
2.8 Nouveau paramètre allow_alter_system 43
2.9 Support des variables personnalisées par `ALTER SYSTEM` 44
2.10 Nouveau timeout pour les transactions longues 45
2.11 Nouvelle fonctionnalité pour amcheck 47
2.12 Éviction de blocs du cache avec pg_buffercache 51
2.13 Paramètre huge_pages_status 53

3/ Sauvegardes **55**

3.1 Sauvegardes incrémentales 56
3.2 Option `--filter` pour `pg_dump`, `pg_dumpall` et `pg_restore` 58
3.3 Autres nouvelles options `pg_dump`/`pg_restore` 59

4/ Réplication **61**

4.1 Failover des slots de réplication logique 62

4.2 Nouvelles colonnes pour pg_replication_slots . . . 76
4.3 Nouveaux messages des *walsenders* . . . 77
4.4 Outil pg_createsubscriber . . . 78

5/ Supervision **87**
5.1 Nouvelle vue pg_stat_checkpointer . . . 88
5.2 Nouvelles colonnes pour pg_stat_progress_vacuum . . . 90
5.3 Nouvelles traces d'une recovery . . . 92
5.4 Nouvelles traces des connexions `trust` . . . 93
5.5 Nouvelle vue `pg_wait_events` . . . 94

6/ Performance **95**
6.1 Regroupement des I/O . . . 96
6.2 Suppression de la limite mémoire de VACUUM . . . 98
6.3 Nouvelles options pour `EXPLAIN` . . . 99
6.4 MergeAppend pour UNION sans ALL . . . 101
6.5 Optimisations des CTE . . . 103
6.6 Optimisations des clauses IN . . . 107
6.7 Meilleure gestion des contraintes NOT NULL . . . 110

7/ Fonctionnement interne **113**
7.1 Nouvelle fonctions unicode . . . 114
7.2 AT LOCAL . . . 115
7.3 Fonction pg_column_toast_chunk_id() . . . 116

8/ Régressions **119**

9/ Peut-être pour la prochaine fois ? **121**

10/ Questions **123**

Notes **125**

Notes **127**

Notes **129**

Nos autres publications **131**
Formations . . . 132
Livres blancs . . . 133
Téléchargement gratuit . . . 134

11/ DALIBO, L'Expertise PostgreSQL **135**

0.1 INTRODUCTION

- Développement depuis juin 2023
- 3 versions beta, 1 version RC
- Version finale : 26 septembre 2024
- Actuellement en 17.2
- Des centaines de contributeurs

Le développement de la version 17 a suivi l'organisation habituelle : un démarrage en juin 2023, des *Commit Fests* tous les deux mois jusqu'en mars 2024, un *feature freeze*, 3 versions beta, une version RC, et enfin la GA.

La version finale est parue le 26 septembre 2024. Plusieurs versions correctives sont sorties depuis.

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

1/ Utilisation

1.1 FONCTION JSON_TABLE()

- Conversion d'une donnée JSON en vue
- Clause NESTED PATH

Une nouvelle fonction permet de présenter des données JSON sous forme de vue. Cette fonction a été dénommée `JSON_TABLE()`. Le premier paramètre que prend la fonction, appelé `context_item`, est une source JSON sur laquelle sera évaluée l'expression JSON. Il est possible de récupérer des données présentes dans une structure imbriquée, avec la clause NESTED PATH.

Prenons l'exemple suivant d'une table créée à partir d'un fichier JSON.

```
postgres=# create table t1 (log json);
CREATE TABLE
postgres=# copy t1(log) from '/tmp/data.json' ;
COPY 12
```

Voici un exemple d'une ligne contenue dans la table `t1` :

```
{"guid":"c39c493b-6759-47b8-9c3f-
↪  10b1b59ba8ef","isActive":false,"balance":"$3,363.44","age":23,"name":"Claudine
↪  Howell","gender":"female","email":"claudinehowell@aquasseur.com","address":"482
↪  Fiske Place, Ogema, Pennsylvania, 1753","registered":"2023-12-22T05:16:27
↪  -01:00","latitude":-88.430942,"longitude":34.196733}
```

Si l'on souhaite récupérer une vue avec uniquement les informations de l'âge et du nom présents dans les lignes de la table `t1`, la commande suivante peut être utilisée :

```
postgres=# select resultats.* from t1, JSON_TABLE(log, '$' COLUMNS (age int PATH
↪  '$.age', name text PATH '$.name')) as resultats;
 age |        name
-----+--------------------
  23 | Claudine Howell
  34 | Ingram Buchanan
  21 | Jensen Jacobs
  28 | Nadia Merritt
  35 | Ashley Mckenzie
  28 | Louella Berry
  21 | Mooney Thompson
  26 | Marquez Orr
  23 | Sharp Mcdowell
  39 | Letitia Workman
```

```
  25 | Madeleine Mcintyre
  24 | Wheeler Glass
(12 rows)
```

- `JSON_TABLE(log, '$'` indique que toute la source JSON doit être utilisée lors de l'évaluation ;
- `COLUMNS (age int PATH '$.age', name text PATH '$.name'))` est la clause qui permet d'indiquer les colonnes présentes dans la vue résultante et où retrouver les données associées ;
- La clause PATH indique l'emplacement de la donnée dans la donnée JSON.

Le résultat peut être manipulé comme habituellement et donc utilisé, par exemple, pour la création d'une table :

```
create table t2 as select resultats.* from t1, JSON_TABLE(log, '$' COLUMNS (age int
↪  PATH '$.age', name text PATH '$.name', balance text PATH '$.balance' )) as
↪  resultats;

postgres=# \d t2
                Table "public.t2"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 age     | integer |           |          |
 name    | text    |           |          |
 balance | text    |           |          |
```

Voici un autre exemple avec des données imbriquées. Le but est de retrouver tous les noms des amis de chacune des personnes.

```
{"name1":"Roach Crosby","friends":[{"id":0,"name":"Deloris
↪  Wilkerson"},{"id":1,"name":"Pitts Lambert"},{"id":2,"name":"Alicia Myers"}]}

postgres=# create table t3 (log json);
CREATE TABLE
postgres=# copy t3(log) from '/tmp/data2.json';
COPY 5
```

L'expression d'évaluation JSON est légèrement plus complexe :

```
postgres=# select resultats.* from t3, JSON_TABLE(log, '$' COLUMNS ( name_1 text
↪  PATH '$.name1',  NESTED PATH '$.friends[*]' COLUMNS ( name_2 text PATH '$.name'
↪  ))) as resultats;
     name_1     |       name_2
----------------+---------------------
 Roach Crosby   | Deloris Wilkerson
 Roach Crosby   | Pitts Lambert
 Roach Crosby   | Alicia Myers
```

```
 Petty Goodwin  | Mandy Baldwin
 Petty Goodwin  | Jordan Fitzgerald
 Petty Goodwin  | Pickett Glenn
 Alyssa Campos  | Margaret Harrington
 Alyssa Campos  | Ofelia Robinson
 Alyssa Campos  | Mccarthy Conway
 Bernard French | Leon Byers
 Bernard French | Sellers Macias
 Bernard French | Madge Barron
 Willa Bell     | Bauer Burton
 Willa Bell     | Wade Cameron
 Willa Bell     | Queen Cardenas
(15 rows)
```

1.2 FONCTION DE CONVERSIONS ENTRE TYPES JSON

- Ensemble de fonctions SQL/JSON path pour convertir des valeurs JSON vers d'autres types JSON :
  - `.bigint()`, `.boolean()`, `.date()`, `.decimal([precision [, scale]])`
  - `.integer()`, `.number()`, `.string()`, `.time()`, `.time_tz()`,
  - `.timestamp()` et `.timestamp_tz()`

Voici quelques exemples :

- le type de sortie est bien `jsonb` :

```
SELECT jsonb_path_query('{"val": "123.45"}', '$.val.number()') \gdesc
```

```
      Column      | Type
------------------+-------
 jsonb_path_query | jsonb
(1 row)
```

- `val` peut être converti dans tout type compatible grâce aux fonctions :

```
SELECT jsonb_path_query('{"val": "123.45"}', '$.val.number()'),
       jsonb_path_query('{"val": "123.45"}', '$.val.decimal(10,1)');
```

```
 jsonb_path_query | jsonb_path_query
------------------+------------------
 123.45           | 123.5
(1 row)
```

```
SELECT jsonb_path_query('{"val": "123.45"}', '$.val.integer()');
```

```
ERROR:  argument "123.45" of jsonpath item method .integer() is invalid for
↪  type integer
```

1.3 FONCTIONS JSON(), JSON_SCALAR() ET JSON_SERIALIZE()

- JSON() : convertir une expression au format text en json
- JSON_SCALAR() : convertir un type de PostgreSQL en type JSON
- JSON_SERIALIZE() : convertir une expression JSON en text ou bytea

Les fonctions JSON(), JSON_SCALAR() et JSON_SERIALIZE() ont été implémentées pour coller avec le standard et améliorer la compatibilité avec d'autres bases de données.

- JSON() : permet de convertir une expression au format text ou bytea (encodée en UTF8) en une valeur avec le type json (pas jsonb !) Comme le montre l'exemple ci-dessous, elle permet aussi de détecter des doublons de clé dans l'entrée.

```
SELECT json('{"a":123, "b":[true,"foo"], "a":"bar"}');

{"a":123, "b":[true,"foo"], "a":"bar"}
-- résultat  de la même requête avec \gdesc
 Column | Type
--------+------
 json   | json
(1 row)

SELECT json('{"a":123, "b":[true,"foo"], "a":"bar"}'  with unique keys);

ERROR:  duplicate JSON object key value
```

- JSON_SCALAR() : permet de transformer une valeur scalaire de PostgreSQL en valeur scalaire JSON.

```
SELECT json_scalar(12.5),
       json_scalar('test'),
       json_scalar(true),
       json_scalar(current_timestamp);

 json_scalar | json_scalar | json_scalar |            json_scalar
-------------+-------------+-------------+-----------------------------------
 12.5        | "test"      | true        | "2024-10-24T14:30:13.889326+02:00"
(1 row)
```

- JSON_SERIALIZE() : permet de convertir une expression JSON en chaîne de caractères ou en chaîne binaire.

```
SELECT json_serialize('{ "a" : 1 } ' RETURNING bytea),
       json_serialize('{ "a" : 1 } ' RETURNING text);

       json_serialize       | json_serialize
----------------------------+----------------
 \x7b20226122203a2031207d20 | { "a" : 1 }
(1 row)
-- même requête avec \gdesc
       Column     | Type
----------------+-------
 json_serialize | bytea
 json_serialize | text
(2 rows)
```

1.4 FONCTIONS JSON_EXISTS(), JSON_QUERY() ET JSON_VALUE()

- JSON_EXISTS() : renvoie vrai si une expression SQL/JSON path renvoie un objet
- JSON_QUERY() : renvoie l'objet JSON sélectionné par l'expression SQL/JSON path
- JSON_VALUE() : renvoie le contenu d'un champ de l'objet JSON sélectionné par l'expression SQL/JSON path

Dans les exemples suivants, nous nous baserons sur cette table :

```
CREATE TABLE json(x jsonb);
INSERT INTO json VALUES(
'{
  "users": [
  {
    "id": 1,
    "name": "Pierre",
    "age": 15,
    "phone": ["00 00 00 00 01", "00 00 00 00 02"]
  },
  {
    "id": 2,
    "name": "Paul",
    "age": 35,
    "address": {
      "address": "2 rue des bois",
      "code": 44000,
      "city": "Nantes"
    },
    "phone": []
  }]
}');
```

JSON_EXISTS(), dont voici le prototype, renvoie vrai si l'expression SQL/JSON path appliquée à l'objet JSON en paramètre trouve un ou des objets dans le document JSON, faux sinon.

```
JSON_EXISTS (
   context_item, path_expression
   [PASSING { value AS varname } [, ...]]
```

```
   [{ TRUE | FALSE | UNKNOWN | ERROR } ON ERROR ]
) → boolean
```

Des paramètres peuvent être passés à l'expression SQL/JSON path avec la clause `PASSING`. Lorsque la fonction échoue, son comportement peut être paramétré avec `TRUE | FALSE | ERROR | UNKNOWN ON ERROR`.

```
SELECT
  json_exists(x, 'strict $.users[$a].phone[*] ? (@ == $b)'
                 PASSING 0 AS a, '00 00 00 00 01' AS b ERROR ON ERROR),
  json_exists(x, 'strict $.users[$a].phone[*] ? (@ == $b)'
                 PASSING 1 AS a, '00 00 00 00 01' AS b ERROR ON ERROR),
  json_exists(x, 'strict $.users[$a].phone[*] ? (@ == $b)'
                 PASSING 9 AS a, '00 00 00 00 01' AS b UNKNOWN ON ERROR)
FROM json;
```

```
 json_exists | json_exists | json_exists
-------------+-------------+-------------
 t           | f           | ¤
(1 row)
```

`JSON_QUERY()`, dont voici le prototype, renvoie le résultat de l'expression SQL/JSON path appliquée à l'objet JSON en paramètre sous forme d'un objet `jsonb`.

```
JSON_QUERY (
  context_item, path_expression
  [ PASSING { value AS varname } [, ...]]
  [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ]
  [ { WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } } [ ARRAY ] WRAPPER ]
  [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
  [ { ERROR | NULL | EMPTY { [ ARRAY ] | OBJECT } | DEFAULT expression } ON EMPTY ]
  [ { ERROR | NULL | EMPTY { [ ARRAY ] | OBJECT } | DEFAULT expression } ON ERROR ]
) → jsonb
```

```
SELECT json_query(x, 'strict $.users[$a].address' PASSING cnt AS a
                     DEFAULT {"city": "unknown"} ON ERROR)
  FROM json, generate_series(0,1) AS cnt;
```

```
                          json_query
---------------------------------------------------------------
 {"city": "unknown"}
 {"city": "Nantes", "code": 44000, "address": "2 rue des bois"}
(2 rows)
```

Lorsque les valeurs retournées par l'expression SQL/JSON path contiennent plusieurs valeurs, le document JSON renvoyé n'est plus valide :

```
SELECT json_query(x, 'strict $.users[*].age' ERROR ON ERROR)
  FROM json;
```

```
ERROR:  JSON path expression in JSON_QUERY should return single item without wrapper
HINT:  Use the WITH WRAPPER clause to wrap SQL/JSON items into an array.
```

Il peut alors être nécessaire d'ajouter la clause `WITH WRAPPER` pour renvoyer les éléments dans un tableau.

```
SELECT json_query(x, 'strict $.users[*].age' WITH WRAPPER ERROR ON ERROR)
  FROM json;
```

```
 json_query
------------
 [15, 35]
(1 row)
```

`JSON_VALUE()`, dont voici le prototype, renvoie la valeur obtenue par l'expression SQL/JSON path appliquée à l'objet JSON en paramètre.

```
JSON_VALUE (
  context_item, path_expression
  [ PASSING { value AS varname } [, ...]]
  [ RETURNING data_type ]
  [ { ERROR | NULL | DEFAULT expression } ON EMPTY ]
  [ { ERROR | NULL | DEFAULT expression } ON ERROR ]
) → text
```

La différence par rapport à `JSON_QUERY()` est que la fonction ne peut renvoyer la valeur que d'un champ JSON.

```
SELECT json_value(x, 'strict $.users[1].address' ERROR ON ERROR)
  FROM json;
```

```
ERROR:  JSON path expression in JSON_VALUE should return single scalar item
```

```
SELECT json_query(x, 'strict $.users[$a].phone' PASSING cnt AS a)
FROM json, generate_series(0,1) AS cnt;
```

```
              json_query
---------------------------------------
 ["00 00 00 00 01", "00 00 00 00 02"]
 []
(2 rows)
```

Comme pour les autres fonctions, il est possible de changer le type de retour :

```
SELECT
  json_query(x, 'strict $.users[$a].age' PASSING cnt AS a) AS "no returning",
  json_query(x, 'strict $.users[$a].age' PASSING cnt AS a RETURNING int) AS
↪  "returning"
FROM json, generate_series(0,1) AS cnt \gdesc
```

```
    Column    |  Type
--------------+---------
 no returning | jsonb
 returning    | integer
(2 rows)
```

1.5 NOUVELLES POSSIBILITÉS DE COPY

- Deux nouvelles options pour COPY
  - `ON_ERROR`
  - `LOG_VERBOSITY`
- Deux options qui évoluent
  - `FORCE_NULL`
  - `FORCE_NOT_NULL`
- Une nouvelle colonne pour `pg_stat_progress_copy`

L'un des gros reproches formulés par les utilisateurs de COPY est l'annulation forcée de l'insertion des lignes valides à cause d'une ligne invalide. Certains aimeraient pouvoir insérer les lignes valides et soit ignorer les lignes invalides, soit les enregistrer dans une table d'erreur ou dans les fichiers de trace. Ce manque a d'ailleurs donné lieu à l'écriture de différents outils, comme pgloader.

La version 17 ajoute l'option `ON_ERROR` qui accepte deux valeurs. La valeur `stop` correspond au comportement historique : la requête tombe en erreur et aucune ligne, y compris les valides, n'est insérée. Comme il s'agit du comportement historique, c'est la valeur par défaut quand l'option n'est pas utilisée.

L'autre valeur est `ignore` et elle fait exactement cela : elle permet d'ignorer les lignes invalides. Cependant, attention, cela concerne uniquement les erreurs de conversion de la donnée du fichier vers le type de données de la colonne. De ce fait, les erreurs sur les contraintes ne sont pas ignorées.

En voici un exemple complet :

```
postgres=# \! cat materiel.csv
1,tente
2,sac de couchage
1,duvet
4,oreiller

postgres=# CREATE TABLE materiel (id integer, nom varchar(10));
CREATE TABLE
```

```
postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv);
ERROR:  value too long for type character varying(10)
CONTEXT:  COPY materiel, line 2, column nom: "sac de couchage"

postgres=# TABLE materiel;
 id | nom
----+-----
(0 rows)
```

La deuxième colonne de la deuxième ligne du fichier CSV contient un texte de plus de dix caractères. Une colonne `varchar(10)` ne peut pas l'accueillir. La requête est donc en erreur, aucune ligne n'est insérée.

```
postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv,
 ↪  ON_ERROR ignore);
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3

postgres=# TABLE materiel;
 id |   nom
----+----------
  1 | tente
  1 | duvet
  4 | oreiller
(3 rows)
```

En ajoutant l'option `ON_ERROR ignore`, la ligne deux est ignorée. Un message de niveau `NOTICE` apparaît et la requête est considérée comme réussie, toutes les autres lignes sont insérées.

```
postgres=# TRUNCATE materiel;
TRUNCATE TABLE

postgres=# ALTER TABLE materiel ADD PRIMARY KEY (id);
ALTER TABLE

postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv, ON_E
ERROR:  duplicate key value violates unique constraint "materiel_pkey"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  COPY materiel, line 3
```

Dans le cas d'une violation de contrainte, y compris avec l'option `ON_ERROR ignore`, les lignes valides ne seront pas insérées et la requête est en erreur.

Pour que la fonctionnalité soit complète, il faudrait pouvoir enregistrer les lignes invalides dans une table ou un fichier. Les valeurs `log` et `table` ont été discutées, gageons que cela arrivera dans une prochaine version.

Le message de niveau `NOTICE` qui apparaît pour chaque ligne invalide indique uniquement le nombre de lignes invalides. Pour avoir plus de détails, il faut utiliser la nouvelle option `LOG_VERBOSITY`. Cette dernière peut avoir deux valeurs : `default` ou `verbose`. Si nous reprenons l'exemple précédent :

```
postgres=# ALTER TABLE materiel DROP CONSTRAINT materiel_pkey;
ALTER TABLE
postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv,
 ↪  ON_ERROR ignore);
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3
postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv,
 ↪  ON_ERROR ignore, LOG_VERBOSITY default);
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3
postgres=# COPY materiel FROM '/home/guillaume/materiel.csv' WITH (FORMAT csv,
 ↪  ON_ERROR ignore, LOG_VERBOSITY verbose);
NOTICE:  skipping row due to data type incompatibility at line 2 for column nom: "sac
 ↪  de couchage"
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3
```

Attention, ce message est affiché pour chaque ligne invalide, ce qui peut devenir très verbeux si le fichier en entrée est d'une qualité pauvre.

Enfin, toujours dans la lignée de la nouvelle option `ON_ERROR`, il est possible de suivre le nombre de lignes ignorées lors de l'exécution de la requête `COPY FROM` avec la nouvelle colonne `tuples_skipped` de la vue `pg_stat_progress_copy`.

Quant aux deux options `FORCE_NULL` et `FORCE_NOT_NULL`, elles n'acceptaient qu'une liste de colonnes. Il est maintenant possible d'utiliser `*` à la place de la liste de toutes les colonnes.

1.6 INSERT, COLONNE IDENTITY ET PARTITIONNEMENT

- Les colonnes IDENTITY sont supportées pour les tables partitionnées
- Les INSERT sont désormais possibles

Jusqu'à présent, une insertion dans une partition d'une table partitionnée possédant une colonne IDENTITY n'était pas faisable. Les INSERT faits dans la table parente eux passent sans problème, comme on peut le voir dans l'exemple suivant avec la table t1 :

```
-- version 16
postgres=# create table t1 (i int, j int generated always as identity) partition by
 ↪  range (i);
CREATE TABLE
postgres=# create table t1_0 partition of t1 for values from (0) to (1000);
CREATE TABLE
postgres=# insert into t1 values (100);
INSERT 0 1
postgres=# insert into t1 values (200);
INSERT 0 1
postgres=# insert into t1 values (300);
INSERT 0 1
postgres=# select * from t1;
  i  | j
-----+---
 100 | 1
 200 | 2
 300 | 3
(3 rows)
```

Un INSERT exécuté sur la partition t1_0 ne fonctionnait pas :

```
-- version 16 :
postgres=# insert into t1_0 values (400);
ERROR:  null value in column "j" of relation "t1_0" violates not-null constraint
DETAIL:  Failing row contains (400, null).
```

Chose désormais acceptée dans la version 17 de PostgreSQL …

```
-- version 17 :
postgres=# insert into t1_0 values (400);
INSERT 0 1
```

… et la valeur insérée dans la colonne j suit la séquence utilisée jusqu'à présent :

```
postgres=# select * from t1;
  i  | j
-----+---
 100 | 1
 200 | 2
 300 | 3
 400 | 4
(4 rows)
```

Si une autre table doit être attachée à une table partitionnée ayant une colonne `IDENTITY`, elle doit respecter les contraintes de la colonne `IDENTITY`, en l'occurrence `NOT NULL`.

```
postgres=# create table t2 (i int, j int) partition by range (i);
CREATE TABLE
postgres=# create table t2_0 partition of t2 for values from (2000) to (3000);
CREATE TABLE
postgres=# insert into t2 values (2100);
INSERT 0 1
postgres=# alter table t1 attach partition t2 for values from (2000) to (3000);
ERROR:  column "j" in child table must be marked NOT NULL
```

Après avoir adapté les données et la table `t2` pour qu'elle respecte cette contrainte, il est possible d'attacher cette table à `t1`.

```
postgres=# insert into t2 values (2100,1);
INSERT 0 1
postgres=# alter table t1 attach partition t2 for values from (2000) to (3000);
ALTER TABLE
postgres=# select * from t1;
  i   | j
------+---
  100 | 1
  200 | 2
  300 | 3
  400 | 4
  600 | 9
 1100 | 6
 1200 | 7
 2100 | 1
(8 rows)
```

Maintenant, un ajout dans cette partition reprendra la suite de la séquence utilisée par `IDENTITY`.

```
postgres=# insert into t2 values (2200);
INSERT 0 1
postgres=# select * from t1;
  i   | j
------+----
```

```
  100 |  1
  200 |  2
  300 |  3
  400 |  4
  600 |  9
 1100 |  6
 1200 |  7
 2100 |  1
 2200 | 10
(9 rows)
```

Si une partition est détachée de la table parente, alors la clause `IDENTITY` de la colonne (`j` dans notre exemple) est retirée. La clause `NOT NULL` est quant à elle conservée.

```
postgres=# alter table t1 detach partition t1_0;
ALTER TABLE
postgres=# select * from t1_0;
  i  | j
-----+---
 100 | 1
 200 | 2
 300 | 3
 400 | 4
 600 | 9
(5 rows)

postgres=# \d t1_0;
               Table "public.t1_0"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 i      | integer |           |          |
 j      | integer |           | not null |
```

1.7 MODIFICATION DE L'EXPRESSION D'UNE COLONNE GÉNÉRÉE

- `ALTER TABLE ... ALTER COLUMN ... SET EXPRESSION AS`
- Possibilité de modifier l'expression d'une colonne générée
- Les anciennes valeurs sont régénérées

Création d'une table avec une colonne générée appelée `mesure_corrigee`.

```
postgres=# create table t1 (mesure int, mesure_corrigee int generated always as
 ↪  (mesure + 7) stored);
CREATE TABLE
```

Insertion de quelques mesures. Celles-ci apparaissent bien corrigées dans la colonne.

```
postgres=# insert into t1 values (10), (20), (30);
INSERT 0 3
postgres=# select * from t1 ;
 mesure | mesure_corrigee
--------+-----------------
     10 |              17
     20 |              27
     30 |              37
(3 rows)
```

L'expression de la colonne `mesure_corrigee` peut être modifiée dans cette nouvelle version de PostgreSQL.

```
postgres=# alter table t1 alter column mesure_corrigee set expression as (mesure +
 ↪  8);
ALTER TABLE
postgres=# insert into t1 values (40), (50), (60);
INSERT 0 3
```

Les nouvelles lignes insérées ont bien leur colonne `mesure_corrigee` générée par la nouvelle expression … tout comme les anciennes lignes !

```
postgres=# select * from t1 ;
 mesure | mesure_corrigee
--------+-----------------
     10 |              18
     20 |              28
     30 |              38
     40 |              48
```

```
     50 |              58
     60 |              68
(6 rows)
```

1.8 GÉNÉRATION DE NOMBRES ALÉATOIRES

- Nouvelle possibilité d'utilisation de la fonction `random`
  - bornes `min` et `max`

La fonction renvoie une valeur comprise entre les deux bornes indiquées. La précision, après la virgule, dépend du nombre de chiffres après la virgule que vous avez indiqué.

```
postgres=# select random(10, 20);
 random
--------
     17
(1 row)

postgres=# select random(10.0, 20.0);
 random
--------
   16.1
(1 row)

postgres=# select random(10.000, 20.000);
 random
--------
 18.605
(1 row)
```

1.9 TYPE INTERVAL ET VALEUR INFINITY

- Le type `interval` supporte désormais les valeurs `infinity`

Cette version introduit la notion d'infini pour le type interval.

Il est donc désormais possible de réaliser des opérations arithmétiques avec des intervalles infinis :

```
SELECT '2024-10-23 06:30:00+02'::timestamptz + INTERVAL '+infinity' AS "cas 1",
       '2024-10-23 06:30:00+02'::timestamptz + INTERVAL '-infinity' AS "cas 2";

   cas 1  |   cas 2
----------+-----------
 infinity | -infinity
```

Le comportement de certaines fonctions a également changé, voici quelques exemples :

- `isfinite()` peut désormais renvoyer autre chose que *vrai* :

```
SELECT isfinite('2024-10-23 06:30:00+02'::timestamptz) AS finite,
       isfinite('2024-10-23 06:30:00+02'::timestamptz + INTERVAL '-infinity')
 ↪  AS infinite;

-- version 17
  finite | infinite
--------+----------
 t      | f
(1 row)
```

- `age()` :

```
SELECT age('infinity'::timestamp);

-- version 16
                        age
-------------------------------------------------
 -292252 years -2 mons -18 days -04:00:54.775807
-- version 17
    age
-----------
 -infinity
```

PostgreSQL vous protège de vous-même !

```
SELECT generate_series(timestamp'-infinity',
                       timestamp'-infinity',
                       interval '-infinity');

ERROR:  step size cannot be infinite
```

1.10 CONTRAINTES D'EXCLUSION SUR LES TABLES PARTITIONNÉES

- Les contraintes d'exclusion sont autorisées si elles :
  - incluent toutes les colonnes de la clé de partitionnement
  - n'utilisent que l'opérateur =

Voici un exemple :

```
CREATE TABLE rendezvous(heure tstzrange, medecin int, client int) PARTITION BY RANGE
 ↪  (heure);
CREATE TABLE rendezvous_1 PARTITION OF rendezvous
  FOR VALUES FROM ('[2024-01-01 00:00:00+02, 2024-01-02 00:00:00+02)'::tstzrange)
               TO ('[2024-01-31 00:00:00+02, 2024-02-01 00:00:00+02)'::tstzrange);
CREATE TABLE rendezvous_2 PARTITION OF rendezvous
  FOR VALUES FROM ('[2024-02-01 00:00:00+02, 2024-02-02 00:00:00+02)'::tstzrange)
               TO ('[2024-02-29 00:00:00+02, 2024-03-01 00:00:00+02)'::tstzrange);
CREATE EXTENSION btree_gist ;
```

Ajoutons une contrainte d'exclusion avec l'opérateur = :

```
ALTER TABLE rendezvous ADD CONSTRAINT overlap EXCLUDE USING gist (heure WITH =);
INSERT INTO rendezvous VALUES('[2024-01-01 10:00:00+02, 2024-01-01 11:30:00+02)', 1,
 ↪  1);
INSERT INTO rendezvous VALUES('[2024-01-01 10:00:00+02, 2024-01-01 11:30:00+02)', 1,
 ↪  1);
```

```
ALTER TABLE
INSERT 0 1
ERROR:  conflicting key value violates exclusion constraint "rendezvous_1_heure_excl"
DETAIL:  Key (heure)=(["2024-01-01 09:00:00+01","2024-01-01 10:30:00+01")) conflicts
 ↪  with existing key (heure)=(["2024-01-01 09:00:00+01","2024-01-01 10:30:00+01")).
```

Nous pouvons voir que la contrainte se comporte comme une contrainte d'unicité.

Essayons avec l'opérateur && :

```
ALTER TABLE rendezvous ADD CONSTRAINT overlap EXCLUDE USING gist (heure WITH &&);
```

```
ERROR:  cannot match partition key to index on column "heure" using non-equal
 ↪  operator "&&"
```

Nous pouvons voir que tout opérateur différent de = est refusé. C'est compréhensible, un autre opérateur pourrait contraindre PostgreSQL à vérifier si la ligne insérée entre en conflit avec des lignes

d'autres partitions.

Cela pourrait changer avec les clefs temporelles en version 18.

2/ Administration

2.1 NOUVEAU FOURNISSEUR DE COLLATION

- Fournisseur interne de collation
- Avantages:
  - portabilité garantie
  - meilleures performances
- Inconvénients
  - pas de tri linguistique
  - pas de comparaisons avancées

Les collations sont un souci majeur pour toute mise à jour du système d'exploitation et pour toute mise en place d'une réplication physique. En effet, pour ne pas avoir de corruption des index, il faut que le tri des données soit garanti identique entre les différents nœuds d'un cluster de réplication. Cela sous-entend la même version du fournisseur de collation, que ce soit la librairie C ou la librairie ICU. Dans le cas de la librairie C, cela sous-entend aussi la même version de la distribution Linux. Donc pas de mélange de système d'exploitation, pas de mise à jour du système d'exploitation, etc.

C'est une limitation très gênante, avec des conséquences potentiellement graves.

La version 17 améliore la donne en proposant un fournisseur interne de collation pour les encodage C (ASCII) et UTF-8. Comme le fournisseur est interne, les soucis de mises à jour systèmes disparaissent complètement.

Cerise sur le gateau, les performances sont aussi au rendez-vous. Le fournisseur interne est bien plus performant que les autres fournisseurs.

Il y a néanmoins des inconvénients à utiliser ce fournisseur. Les tris linguistiques ne sont pas gérés. De même, il n'est pas possible d'utiliser les comparaisons avancées.

La création d'une base avec ce fournisseur doit passer par la clause `builtin_locale`, ainsi :

```
CREATE DATABASE nouvelle_base
  LOCALE_PROVIDER builtin
  BUILTIN_LOCALE "C.UTF-8"
  TEMPLATE template0;
```

L'article de Daniel Vérité (en anglais) apporte de nombreuses informations et de nombreux tests sur ce nouveau fournisseur. Vous le trouverez sur https://postgresql.verite.pro/blog/2024/07/01/pg17-utf8-collation.html.

Il existe aussi l'excellente conférence de Jeremy Schneider and Jeff Davis (ce dernier étant l'auteur de cette fonctionnalité sur ce sujet), conférence réalisée pendant l'événement « PGConf.dev 2024 ». Elle est disponible sur YouTube[1].

[1] https://www.youtube.com/watch?v=KTA6oau7tl8

2.2 NOUVEAU RÔLE PG_MAINTAIN ET DROIT MAINTAIN

- Permettre de gérer les opérations de maintenance sans être propriétaire des tables
- Rôle `pg_maintain`
  - droit global pour tous les objets de l'instance
- Droit `MAINTAIN`
  - droit par table
- Opérations de maintenance couvertes
  - VACUUM, ANALYZE, REINDEX
  - REFRESH MATERIALIZED VIEW
  - CLUSTER, LOCK TABLE

Avant la version 17, les opérations de maintenance ne pouvaient être réalisées que par le propriétaire de l'objet ou par un superutilisateur. Certes, il est toujours possible d'avoir un groupe propriétaire de l'objet et d'ajouter un rôle de maintenance comme membre de ce groupe. Mais si jamais un objet était créé par quelqu'un et que la propriété de cet objet n'était pas donnée au groupe, le rôle de maintenance ne pouvait pas faire d'opération de maintenance sur cet objet.

Il a donc été décidé d'ajouter un droit qui permet de réaliser des opérations de maintenance sur un objet, sans avoir plus de droits, notamment lecture et écriture. Ce droit s'appelle `MAINTAIN`. Cependant, pour éviter d'avoir à ajouter ce droit à tous les objets créés dans l'instance, un rôle nommé `pg_maintain` a aussi été ajouté qui donne ce droit à tous ces membres pour tous les objets de l'instance.

Prenons un exemple. Nous voulons avoir un rôle qui va être chargé d'exécuter les opérations de maintenance comme le `VACUUM`. Nous allons créer une base et y ajouter trois tables :

```
$ export PGDATABASE=b3
$ createdb b3
$ createuser maintenance
$ psql --username postgres --command 'CREATE TABLE t1 (id integer);' --
command 'CREATE TABLE t2 (id integer);' --command 'CREATE TABLE t3 (id integer);'
```

```
CREATE TABLE
CREATE TABLE
CREATE TABLE
```

Maintenant, lançons un VACUUM sur la table t1, d'abord en tant que le propriétaire de la table, puis en tant que l'utilisateur `maintenance` :

```
$ vacuumdb --username postgres --table t1
vacuumdb: vacuuming database "b3"
$ vacuumdb --username maintenance --table t1
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "t1", skipping it
```

Le propriétaire réussit à faire l'opération, l'utilisateur `maintenance` récupère une erreur indiquant qu'il n'a pas le droit de faire cette opération sur `t1`. Donnons-lui le droit pour cette table uniquement :

```
$ psql --username postgres --command 'GRANT MAINTAIN ON TABLE t1 TO maintenance'
GRANT
```

Et lançons de nouveau un VACUUM sur la table :

```
$ vacuumdb --username maintenance --table t1
vacuumdb: vacuuming database "b3"
```

Cette fois, c'est passé sans erreur. Maintenant, lançons un VACUUM sur toute la base :

```
$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "pg_proc", skipping it
WARNING:  permission denied to vacuum "pg_attribute", skipping it
[...]
WARNING:  permission denied to vacuum "pg_publication", skipping it
WARNING:  permission denied to vacuum "t3", skipping it
WARNING:  permission denied to vacuum "pg_foreign_table", skipping it
[...]
WARNING:  permission denied to vacuum "pg_sequence", skipping it
WARNING:  permission denied to vacuum "t2", skipping it
WARNING:  permission denied to vacuum "pg_publication_namespace", skipping it
[...]
```

Nous avons des erreurs pour toutes les tables utilisateurs et systèmes de la base b3, sauf évidemment t1. Rendons donc le membre du rôle pg_maintain :

```
$ psql --username postgres --command 'GRANT pg_maintain TO maintenance'
GRANT ROLE
```

Tentons déjà l'opération de maintenance sur la table t2 :

```
$ vacuumdb --username maintenance --table t2
vacuumdb: vacuuming database "b3"
```

C'est passé. Maintenant, tentons la base complète :

```
$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
```

Ça passe aussi. Pour autant, cet utilisateur ne peut pas lire les données de la table :

```
$ psql --username maintenance --command 'SELECT * FROM t1'
ERROR:  permission denied for table t1
```

Il est aussi à noter que cette fonctionnalité devait arriver en version 16 mais qu'elle a été retirée un peu avant la sortie de la version 16 car elle posait apparemment quelques soucis.

2.3 RÉSUMÉS DES WAL

- Nouveau processus en arrière-plan : `walsummarizer`
- Fichiers sommaires stockés dans `$PGDATA/pg_wal/summaries`
- Nouveaux paramètres
  - activation avec `summarize_wal`
  - rétention des fichiers avec `wal_summary_keep_time`
- Nouvel utilitaire `pg_walsummary` permettant de consulter le contenu des sommaires

Activation

Par défaut, le processus `walsummarizer` n'est pas lancé. Pour qu'il le soit au démarrage de PostgreSQL, il faut configurer le paramètre `summarize_wal` à `on` et le paramètre `wal_level` à une valeur autre que `minimal`.

```
summarize_wal = on
```

Le paramètre `wal_summary_keep_time` détermine combien de temps les fichiers `summaries` sont conservés avant d'être automatiquement supprimés. Par défaut, ils sont gardés pendant 10 jours.

Rétention définie à 15 jours pour l'exemple :

```
wal_summary_keep_time = '15d'
```

Si le processus `walsummarizer` est désactivé, les fichiers existants ne seront pas supprimés, car le processus ne sera pas en cours d'exécution.

Fichiers sommaires WAL

Les fichiers sommaires de WALs (ou `summaries`) sont générés dans le répertoire `pg_wal/summaries/` par le processus `walsummarizer`. Chaque sommaire couvre une timeline spécifique. Il enregistre des informations sur les modifications apportées aux données sur cette timeline. Ce processus est nécessaire pour les sauvegardes incrémentales.

Voici un exemple du contenu de ce répertoire :

```
postgres@17:~$ ls -altrh data_v17/pg_wal/summaries/
```

```
total 76K
-rw------- 1 postgres postgres 4,7K sept.  14 14:42
 ↪  000000010000000001000028000000000010B1D38.summary
-rw------- 1 postgres postgres 4,9K sept.  14 14:42
 ↪  00000001000000000010B1D3800000000014DF280.summary
-rw------- 1 postgres postgres   56 sept.  14 14:42
 ↪  00000001000000000014DF28000000000014DF380.summary
-rw------- 1 postgres postgres   56 sept.  14 14:42
 ↪  00000001000000000014E5D6800000000014E5E68.summary
...
-rw------- 1 postgres postgres 1,9K sept.  14 14:54
 ↪  000000010000000001F87F400000000004A9A748.summary
drwx------ 2 postgres postgres 4,0K sept.  14 14:54 .
drwx------ 4 postgres postgres 4,0K sept.  14 14:55 ..
```

Consulter les sommaires WAL disponibles

Pour vérifier les sommaires de WALs existants, la fonction `pg_available_wal_summaries()` permet d'énumérer les plages de LSN disponibles :

```
postgres=# SELECT * FROM pg_available_wal_summaries();

-[ RECORD 1 ]--------
tli       | 1
start_lsn | 0/152AA98
end_lsn   | 0/19519D8
```

Cette commande montre qu'un WAL summary est disponible pour la plage de LSN allant de 0/152AA98 à 0/19519D8.

Informations supplémentaires sur les sommaires de WAL

Pour obtenir les informations détaillées sur les blocs modifiés dans un sommaire de WAL, la fonction `pg_wal_summary_contents()` peut être utilisée comme suit :

```
postgres=# SELECT * FROM pg_wal_summary_contents(1, '0/152AA98', '0/19519D8');

-[ RECORD 1 ]--+------
relfilenode    | 0
reltablespace  | 1663
reldatabase    | 24576
relforknumber  | 0
relblocknumber | 0
is_limit_block | t
```

Cette sortie montre les détails des blocs modifiés, notamment l'identifiant de la base de données, le tablespace, et les blocs affectés.

Vérifier le contenu des sommaires avec `pg_walsummary`

Le nouvel utilitaire `pg_walsummary` permet de consulter le contenu des sommaires, à des fins de débogage par exemple :

```
postgres@17:~$ /usr/lib/postgresql/17/bin/pg_walsummary \
> data_v17/pg_wal/summaries//000000010000000049A324288000000049A74A620.summary
TS 1663, DB 24990, REL 0, FORK main: limit 0
TS 1663, DB 24990, REL 827, FORK main: block 0
TS 1663, DB 24990, REL 828, FORK main: limit 0
TS 1663, DB 24990, REL 828, FORK main: block 0
TS 1663, DB 24990, REL 1247, FORK main: limit 0
TS 1663, DB 24990, REL 1247, FORK main: blocks 0..14
```

État du processus `WAL summary`

La commande `pg_get_wal_summarizer_state()` permet de surveiller l'état du processus qui génère les sommaires de WALs :

```
postgres=# SELECT * FROM pg_get_wal_summarizer_state();

-[ RECORD 1 ]--+----------
summarized_tli | 1
summarized_lsn | 0/4A9A748
pending_lsn    | 0/4B51908
summarizer_pid | 365327
```

Cette commande montre le LSN de fin du dernier fichier de résumé et le dernier LSN traité.

Taille des fichiers sommaires

Les fichiers générés par le processus `walsummarizer` sont de taille raisonnable :

```
postgres@17:~$ dropdb bench
createdb bench
psql -c "SELECT pg_create_restore_point('Début test')" -c "CHECKPOINT;"
pgbench -i -s410 --partitions=50 bench

postgres@17:~$ ls -altrh data_v17/pg_wal/summaries/
total 1.6M
```

2.4 OPTIONS –INDEX ET –JOBS SUR REINDEXDB

- Plusieurs index sur des tables différentes peuvent être traités en parallèle
- Options `--jobs` et `--index`

Lorsqu'on effectue un `REINDEX` au niveau des index (et non au niveau de la table entière), il était jusqu'à présent difficile de gérer plusieurs tâches en parallèle car plusieurs index peuvent dépendre de la même table, ce qui pouvait entraîner des conflits de traitement concurrent.

Au lieu de tenter de gérer chaque index individuellement en parallèle, les index sont regroupés par table. Ainsi, chaque tâche de réindexation est dédiée à une table entière, permettant de traiter plusieurs tables en parallèle sans conflit.

Exemple d'utilisation :

```
reindexdb --jobs=2 --index=idx_data1 --index=idx_data2
```

2.5 OPTION –ALL AVEC VACUUMDB ET REINDEXDB

- Quelques options utilisable avec `--all`
- Pour `vacuumdb` : `--table`, `--schema`, `--exclude-schema`
- Pour `reindexdb` : `--table`, `--schema`, `--index` et `--system`

Lorsque l'option `--all` est utilisée, il est désormais possible d'utiliser les options `--table`, `--schema`, `--exclude-schema` pour `vacuumdb`, et `--table`, `--schema`, `--index` pour `reindexdb`, permettant ainsi de spécifier des objets à traiter dans toutes les bases de données.

De plus, la restriction selon laquelle l'option `--system` ne peut pas être utilisée avec les options `--table`, `--schema` et `--index` a été supprimée pour `reindexbd`.

Ces restrictions ont été retirées car aucune raison technique ne justifiait leur maintien.

Voici un exemple pour `vacuumdb` :

```
$ vacuumdb -U postgres --all --schema pg_catalog
vacuumdb: vacuuming database "postgres"
vacuumdb: vacuuming database "template1"
```

Sur la version 16, on a l'erreur suivante :

```
$ vacuumdb -U postgres --all --schema pg_catalog
vacuumdb: error: cannot vacuum specific schema(s) in all databases
```

Et voici un exemple pour `reindexdb` :

```
$ reindexdb -U postgres --all --schema public
reindexdb: reindexing database "postgres"
reindexdb: reindexing database "template1"
```

Sur la version 16, l'erreur suivante apparaît :

```
$ reindexdb -U postgres --all --schema public
reindexdb: error: cannot reindex specific schema(s) in all databases
```

2.6 SUPPORT DES *TRIGGERS* SUR REINDEX

- Support du mot clé `REINDEX` sur les *event triggers*
- `CREATE EVENT TRIGGER ... ON ddl_command_start WHEN TAG IN ('REINDEX')`
  - Grâce à cela, il est possible de connaître la liste complète des index sur lesquels un `REINDEX` a travaillé.

Utilisation de triggers sur REINDEX

Les *event triggers* dans PostgreSQL peuvent désormais capturer les événements de `REINDEX`, permettant de suivre les opérations et d'agir en conséquence. Voici un exemple pratique de mise en place d'un trigger d'événement lors du démarrage d'un `REINDEX`.

Création d'une fonction qui sera exécutée au début de l'opération `REINDEX` pour afficher un message d'information :

```
postgres=# CREATE OR REPLACE FUNCTION reindex_start_func()
RETURNS event_trigger AS $$
BEGIN
  RAISE NOTICE 'START REINDEX: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
```

Création du trigger d'événement `trg_reindex_start` sur l'événement `ddl_command_start`, associé à la commande `REINDEX` :

```
postgres=# CREATE EVENT TRIGGER trg_reindex_start ON ddl_command_start
WHEN TAG IN ('REINDEX')
EXECUTE PROCEDURE reindex_start_func();

CREATE EVENT TRIGGER
```

Dès que la commande `REINDEX` est exécutée, PostgreSQL déclenche le trigger, et le message de notification s'affiche :

```
postgres=# REINDEX INDEX idx1_data1;
```

```
NOTICE: START REINDEX: ddl_command_start REINDEX
REINDEX
```

Gestion des triggers d'événement sur REINDEX

Les triggers sur l'événement `REINDEX` sont enregistrés dans le catalogue de PostgreSQL, et peuvent être désactivés ou modifiés selon les besoins.

Ces triggers permettent de suivre en temps réel l'activité des index dans la base de données, offrant des outils supplémentaires pour la supervision et la gestion des performances.

2.7 SUPPORT DES *TRIGGERS* SUR CONNEXIONS

- Syntaxe
  - `CREATE EVENT TRIGGER ... ON LOGIN ...`
- Permet le déclenchement d'une *fonction* lorsqu'une connexion est réussie
- Option `event_triggers` pour activer/désactiver leurs déclenchement
  - intéressant quand la fonction trigger ne fonctionne pas

Il est désormais possible de positionner des *triggers* sur une connexion. La fonction associée sera exécutée juste après que l'authentification de l'utilisateur soit réussie. L'utilisation est la même que pour les autres *triggers*, dont voici un exemple.

Création d'une table d'historisation des connexions :

```
postgres=# create table t1 (utilisateur text, connexions timestamp);
CREATE TABLE
```

Création de la fonction `log_connexions` qui insère dans la table `t1` l'utilisateur utilisé dans la session ainsi que l'horodatage. La clause `SECURITY DEFINER` permet d'indiquer que cette fonction sera exécutée avec les droits associés à son propriétaire.

```
postgres=# create function log_connexions() returns event_trigger as $$
begin
insert into t1 values (session_user, current_timestamp);
end
$$ language plpgsql security definer;
CREATE FUNCTION
```

Petit rappel : une fonction `security definer` s'exécute avec les droits de son propriétaire. Dans le cas présent, cela permet d'interdire les écritures dans la table de log pour tous les utilisateurs, sauf celui qui a créé la fonction, empêchant ainsi aux autres utilisateurs de supprimer les traces de leurs connexions.

Création du *trigger* `trigger_connexions` qui sera exécuté sur un évènement de type `login`.

```
postgres=# CREATE EVENT TRIGGER trigger_connexions ON login EXECUTE PROCEDURE
 ↪  log_connexions();
CREATE EVENT TRIGGER
```

Désormais, en nous connectant, la table `t1` est bien complétée.

```
postgres=# \c postgres postgres
You are now connected to database "postgres" as user "postgres".
postgres=# select * from t1 ;
 utilisateur |         connexions
-------------+----------------------------
 postgres    | 2024-07-09 11:25:35.357863
(1 row)

postgres=# \c postgres dalibo
You are now connected to database "postgres" as user "dalibo".

postgres=> \c postgres postgres
You are now connected to database "postgres" as user "postgres".

postgres=# select * from t1 ;
 utilisateur |         connexions
-------------+----------------------------
 postgres    | 2024-07-09 11:25:35.357863
 dalibo      | 2024-07-09 11:25:40.914802
 postgres    | 2024-07-09 11:25:45.52029
(3 rows)
```

Une nouvelle colonne apparaît dans le catalogue `pg_database`. Il s'agit de `dathasloginevt` qui indique si la base de données en question possède ou non un *trigger* sur un évènement de type `login`.

```
postgres=# select datname, dathasloginevt from pg_database;
  datname  | dathasloginevt
-----------+----------------
 template1 | f
 template0 | f
 postgres  | t
(3 rows)
```

Si la fonction utilisée renvoie une erreur, il vous sera **impossible** de vous connecter à votre base de données.
Il existe deux solutions pour contourner ce problème. La première solution serait d'arrêter votre instance, de la relancer en mode `single-server`, de corriger la fonction ou de supprimer le *trigger*, puis d'arrêter l'instance et la redémarrer normalement.
Les développeurs de PostgreSQL ont pensé à une solution un peu moins radicale. Un nouveau paramètre existe désormais : `event_triggers`. Passée à `off`, il permet de désactiver l'exécution des *triggers sur événement*. La deuxième solution de contournement consiste donc à modifier ce paramètre et à recharger la configuration de PostgreSQL.
En bref, soyez attentifs !

2.8 NOUVEAU PARAMÈTRE ALLOW_ALTER_SYSTEM

- Nouveau paramètre `allow_alter_system`
- on par défaut
- Si `off` : une erreur est retournée si `ALTER SYSTEM` est utilisé
- Configurable seulement dans le fichier `postgresql.conf`

Lorsque ce paramètre est configuré à `off`, la commande `ALTER SYSTEM` retourne une erreur. C'est, par exemple, utile pour des systèmes utilisant Patroni, pour qui la configuration doit être faite via le fichier de configuration YAML ou la commande `patronictl edit-config`.

La documentation insiste fortement sur le fait que ce n'est **pas** une fonctionnalité destinée à la sécurité. Elle désactive juste la commande `ALTER SYSTEM`. Un superutilisateur malveillant peut toujours trouver un moyen de réactiver l'option ou de changer le paramétrage sans l'utiliser.

2.9 SUPPORT DES VARIABLES PERSONNALISÉES PAR ALTER SYSTEM

- ALTER SYSTEM supporte la modification de GUC inconnu du moteur
- Très pratique pour la configuration des extensions

Jusqu'à présent, la commande ALTER SYSTEM n'acceptait que des paramètres connus du moteur PostgreSQL. Par exemple, si on souhaite configurer l'extension `pg_stat_statements`, cela n'était pas possible avec ALTER SYSTEM :

```
$ psql -U postgres
psql (16.3)
Type "help" for help.

postgres=# create extension pg_stat_statements ;
CREATE EXTENSION
postgres=# alter system set pg_stat_statements.max = '500';
ERROR:  unrecognized configuration parameter "pg_stat_statements.max"
```

Avec la version 17, il est désormais possible de modifier des paramètres de configuration qui sont initialement inconnus du moteur.

Même exemple :

```
$ psql -U postgres
psql (17.2)
Type "help" for help.

postgres=# create extension pg_stat_statements ;
CREATE EXTENSION
postgres=# alter system set pg_stat_statements.max = '500';
ALTER SYSTEM
```

2.10 NOUVEAU TIMEOUT POUR LES TRANSACTIONS LONGUES

- Paramètre `transaction_timeout`
- Durée maximale autorisée pour la durée d'une transaction
- Ne concerne pas les transactions préparées

Le paramètre `transaction_timeout` vient s'ajouter à la liste des timeouts disponibles.

```
\dconfig+ (transaction|statement|idle)*timeout*

                          List of configuration parameters
             Parameter              | Value |  Type   |  Context  | Access privileges
------------------------------------+-------+---------+-----------+-------------------
 ↪  ---
 idle_in_transaction_session_timeout | 0     | integer | user      | ¤
 idle_session_timeout                | 0     | integer | user      | ¤
 statement_timeout                   | 0     | integer | user      | ¤
 transaction_timeout                 | 0     | integer | user      | ¤
(3 rows)
```

Il est donc possible de configurer des timeouts :

- pour les ordres (*statements*) trop longs avec `statement_timeout` ;
- pour les sessions inactives (*idle*) avec `idle_session_timeout` ;
- pour les sessions inactives au sein d'une transaction toujours en cours (*idle in transaction*) avec `idle_in_transaction_session_timeout` ;
- pour les transactions trop longues (`transaction_timeout`).

Il est important que l'application sache gérer l'arrêt de connexion ou l'annulation de la transaction et réagir en conséquence. Dans le cas contraire, l'application pourrait rester déconnectée et, suivant les cas, des données pourraient être perdues.

Démonstration :

```
SET transaction_timeout TO '1s';
BEGIN;
SELECT pg_sleep_for(INTERVAL '2s');
```

```
FATAL:  terminating connection due to transaction timeout
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

Les transactions préparées ne sont pas concernées par ce timeout.

2.11 NOUVELLE FONCTIONNALITÉ POUR AMCHECK

- Détection dans les doublons des index de contraintes d'unicité
- Extension `amcheck`
  - nouveau paramètre `checkunique` pour `bt_index_check` et `bt_index_parent_check()`
- Outil `pg_amcheck`
  - nouvelle option `--checkunique`

L'extension `amcheck` (et par conséquence l'outil `pg_amcheck`) peut désormais détecter les violations de contraintes d'unicité.

Afin de tester cette évolution, il faut installer l'extension `amcheck` dans la base de données.

```
CREATE EXTENSION amcheck;
```

Il faut également mettre en place un jeu d'essai (copié depuis la mailing list hackers). **Attention**, la modification du catalogue de PostgreSQL n'est pas recommandée en temps normal.

```
CREATE TABLE junk (t text);
CREATE UNIQUE INDEX junk_idx ON junk USING btree (t);
INSERT INTO junk (t) VALUES ('fee'), ('fi'), ('fo'), ('fum');
UPDATE pg_catalog.pg_index
   SET indisunique = false
 WHERE indrelid = 'junk'::regclass;
INSERT INTO junk (t) VALUES ('fee'), ('fi'), ('fo'), ('fum');
UPDATE pg_catalog.pg_index
   SET indisunique = true
 WHERE indrelid = 'junk'::regclass;
SELECT * FROM junk ORDER BY 1;
```

```
  t
-----
 fee
 fee
 fi
 fi
 fo
 fo
```

```
 fum
 fum
(8 rows)
```

On peut ensuite analyser un index ou une table et rechercher les violations de clé unique avec l'outil `pg_amcheck`.

```
pg_amcheck --database=postgres --relation=junk --checkunique
```

```
btree index "postgres.public.junk_idx":
    ERROR:  index uniqueness is violated for index "junk_idx"
    DETAIL:  Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5))
 ↪  page lsn=0/19E9498.
```

Le même résultat peut être obtenu en utilisant directement l'extension `amcheck`, où les fonctions `bt_index_check` et `bt_index_parent_check` se sont vues ajouter un nouveau paramètre `checkunique` :

```
\df bt_index_(parent_|)check
```

```
                                List of functions
        Name          |                                Argument data types
----------------------+-------------------------------------------------------------
 ↪  ---------------------
...
bt_index_check        | index regclass, heapallindexed boolean, checkunique boolean
...
bt_index_parent_check | index regclass, heapallindexed boolean, rootdescend boolean,
                        checkunique boolean
(7 rows)
```

Voici un exemple :

```
psql -c "SELECT bt_index_check('junk_idx', false, true);"
```

```
ERROR:  index uniqueness is violated for index "junk_idx"
DETAIL:  Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5)) page
 ↪  lsn=0/19E9498.
```

Afin de mieux comprendre ce que renvoie `amcheck`, nous allons utiliser l'extension `pageinspect` :

```
CREATE EXTENSION pageinspect;
```

La requête suivante permet d'afficher la page numéro 1 de l'index `junk_idx`, qui nous est pointée dans le message d'erreur. C'est le premier nombre des `ctid` `(1,1)` et `(1,2)`. Le deuxième nombre pointe l'`itemoffset` où est stockée l'entrée d'index.

```
SELECT itemoffset, ctid, data FROM bt_page_items(get_raw_page('junk_idx',1));
```

```
 itemoffset | ctid  |          data
------------+-------+------------------------
          1 | (0,1) | 09 66 65 65 00 00 00 00
          2 | (0,5) | 09 66 65 65 00 00 00 00
          3 | (0,2) | 07 66 69 00 00 00 00 00
          4 | (0,6) | 07 66 69 00 00 00 00 00
          5 | (0,3) | 07 66 6f 00 00 00 00 00
          6 | (0,7) | 07 66 6f 00 00 00 00 00
          7 | (0,4) | 09 66 75 6d 00 00 00 00
          8 | (0,8) | 09 66 75 6d 00 00 00 00
(8 rows)
```

Comme le précise le message d'erreur, les entrées 1 et 2 qui pointent respectivement vers les entrées de la table (ou *heap*) `ctid=(0,1)` et `ctid=(0,5)` contiennent les mêmes données, ce qui est en contradiction avec l'existence d'un index unique.

La colonne `ctid` nous donne la position de la ligne dans la table (*heap*). On peut observer que les tuples dans les slots 1 et 5 de la table contiennent les mêmes données.

```
SELECT lp, t_ctid, t_data FROM heap_page_items(get_raw_page('junk',0));
```

```
 lp | t_ctid |   t_data
----+--------+------------
  1 | (0,1)  | \x09666565
  2 | (0,2)  | \x076669
  3 | (0,3)  | \x07666f
  4 | (0,4)  | \x0966756d
  5 | (0,5)  | \x09666565
  6 | (0,6)  | \x076669
  7 | (0,7)  | \x07666f
  8 | (0,8)  | \x0966756d
(8 rows)
```

Le LSN mentionné dans le message d'erreur est extraite de l'entête de la page d'index.

```
SELECT f.d, ph.lsn
  FROM (VALUES
        ('junk page 0', get_raw_page('junk',0)),
        ('junk_idx page 1', get_raw_page('junk_idx',1))) AS F(d,x)
       CROSS JOIN LATERAL page_header(x) AS ph
;
```

```
        d        |    lsn
-----------------+-----------
 junk page 0     | 0/19E9458
 junk_idx page 1 | 0/19E9498
(2 rows)
```

2.12 ÉVICTION DE BLOCS DU CACHE AVEC PG_BUFFERCACHE

- Nouvelle fonction `pg_buffercache_evict(bufferid)`
- Nécessite l'attribut `superuser`
- Destinée à la réalisation de tests

Il est désormais possible d'exclure des blocs du cache avec l'extension `pg_buffercache`. Cette fonctionnalité est réservée aux utilisateurs avec l'attribut `superuser` et ne doit être utilisé que pour des tests. En effet, les garanties prises par l'extension ne permettent pas de garantir que la page exclue est bien toujours celle que l'on avait identifiée, ni qu'elle soit toujours dans l'état que l'on avait observé.

Afin de pouvoir tester cette nouvelle fonctionnalité, nous allons installer l'extension :

```
CREATE EXTENSION pg_buffercache
\df pg_buffercache_evict
```

```
                                List of functions
 Schema |          Name          | Result data type | Argument data types | Type
--------+------------------------+------------------+---------------------+------
 public | pg_buffercache_evict | boolean          | integer             | func
(1 row)
```

On voit que la nouvelle fonction `pg_buffercache_evict` prend en paramètre un entier. Il correspond à l'id du bloc à exclure du cache.

La requête suivante nous permet d'identifier les tables utilisateurs de la base courante présentes dans le cache.

```
SELECT n.nspname, c.relname, count(*) AS buffers
  FROM pg_buffercache b
       JOIN pg_class c
            ON b.relfilenode = c.relfilenode
  JOIN pg_namespace n ON n.oid = c.relnamespace
  WHERE n.nspname NOT LIKE 'pg_%'
  GROUP BY n.nspname, c.relname
  ORDER BY 3 DESC;
```

```
 nspname |          relname          | buffers
---------+---------------------------+---------
 public  | junk                      |       5
 public  | junk_idx                  |       2
```

```
 public  | rendezvous_2_heure_excl |       1
 public  | rendezvous_1_heure_excl |       1
 public  | rendezvous_1            |       1
(5 rows)
```

Vérifions que tous les blocs de la relation `rendezvous_1` sont bien dans le cache :

```
EXPLAIN (ANALYZE, BUFFERS, TIMING OFF, COSTS OFF, SUMMARY OFF)
  SELECT * FROM rendezvous_1;
```

```
                    QUERY PLAN
--------------------------------------------------
 Seq Scan on rendezvous_1 (actual rows=1 loops=1)
   Buffers: shared hit=1
(2 rows)
```

Pour rappel, `shared hit` indique le nombre de blocs lus en cache, contrairement à `shared read` qui nécessite de faire appel au noyau pour récupérer un bloc.

Nous allons supprimer le bloc de la relation `rendezvous_1` du cache de PostgreSQL :

```
SELECT n.nspname, c.relname, b.bufferid, pg_buffercache_evict(b.bufferid)
  FROM pg_buffercache b
      JOIN pg_class c
           ON b.relfilenode = relfilenode
  JOIN pg_namespace n ON n.oid = c.relnamespace
  WHERE c.relname = 'rendezvous_1';
```

```
([local]:5443 postgres@postgres=# \e
 nspname |   relname    | bufferid | pg_buffercache_evict
---------+--------------+----------|----------------------
 public  | rendezvous_1 |      688 | t
(1 row)
```

Le code retour de la fonction est `true` sauf si le buffer pointé n'existe pas, était utilisé (`pinned`) ou s'il a été sali (`dirtied`).

Un nouvel `EXPLAIN (ANALYZE)` nous confirme que le bloc n'est plus dans le cache de PostgreSQL :

```
                    QUERY PLAN
--------------------------------------------------
 Seq Scan on rendezvous_1 (actual rows=1 loops=1)
   Buffers: shared read=1
(2 rows)
```

2.13 PARAMÈTRE HUGE_PAGES_STATUS

- Nouveau paramètre `huge_pages_status`
  - en lecture seule !
- Vérifie l'allocation des `huges_pages` (intéressant si `huge_pages = try`)
- Valeurs possibles
  - `on` (*huge pages* allouées)
  - `off` (aucune *huge page* allouée)
  - `unknown` (information non récupérable)

Ce paramètre en lecture seule montre l'état de l'allocation des *huge pages* d'une instance pour laquelle `huge_pages = try`. L'échec de l'allocation des *huge pages* n'empêche pas la séquence de démarrage de se terminer. Le statut des *huge pages* n'est pas facilement accessible sans des outils de supervision du système d'exploitation, et cela n'est pas toujours autorisé par l'environnement d'exécution de PostgreSQL. Comme les autres paramètres liés aux *huge pages*, il est fonctionnel sous Linux et Windows.

Ce paramètre peut retourner les valeurs suivantes :

- `on` lorsque l'allocation des *huge pages* est effectuée ;
- `off` lorsque l'allocation des *huge pages* n'est pas effectuée ;
- `unknown`, état particulier qui ne peut être aperçu qu'en utilisant, par exemple, `postgres -C`, instance arrêtée, car cela n'a pas de sens de vérifier que les allocations utilisent les *huges pages* si PostgreSQL n'est pas démarré.

Exemple d'utilisation, les *huge pages* sont désactivées :

```
postgres=# SHOW huge_pages_status;
 huge_pages_status
-------------------
 off
(1 row)
```

Activons les `huge_pages` sur le système.

La formule à utiliser correspond à environ 10% de plus que la valeur du `shared_buffers` divisée par la taille des *huge pages* utilisées, ici 2 Mo. Pour l'exemple avec 1 Go de `shared_buffers`, cela donne : 1024/2 × 1,10 = 563,2. Le paramètre `vm.nr_overcommit_hugepages = 564` est ajouté dans le fichier `/etc/sysctl.conf` de la machine.

Nous pouvons maintenant recharger la configuration du noyau pour la tester :

```
root@test17:~# sysctl --system
```

Après redémarrage de PostgreSQL, nous pouvons vérifier s'il a utilisé les `huge pages` :

```
postgres=# show huge_pages_status;
 huge_pages_status
-------------------
 on
(1 row)
```

Pour terminer, voici un exemple de la récupération de la valeur du paramètre `huge_pages_status` alors que l'instance est arrêtée :

```
postgres@test17:~$ postgres -C huge_pages_status
unknown
```

3/ Sauvegardes

3.1 SAUVEGARDES INCRÉMENTALES

- Nouvelle fonctionnalité de sauvegarde incrémentale dans PostgreSQL
  - Commande `UPLOAD_MANIFEST` pour télécharger le manifeste
  - Commande `BASE_BACKUP` pour effectuer la sauvegarde

Pour faire une sauvegarde incrémentale, il est désormais possible d'utiliser la nouvelle commande de réplication `UPLOAD_MANIFEST` pour télécharger le manifeste de la sauvegarde précédente. Cette dernière peut être une sauvegarde complète ou une autre sauvegarde incrémentale. Ensuite, la commande `BASE_BACKUP` avec l'option `INCREMENTAL` permet d'effectuer la sauvegarde. L'outil *pg_basebackup* dispose désormais d'une option `--incremental=PATH_TO_MANIFEST` pour déclencher ce comportement.

Sauvegarde complète

La première étape consiste à effectuer une sauvegarde complète de la base de données. Cela sert de point de référence pour les sauvegardes incrémentales suivantes :

```
postgres@17:~$ pg_basebackup -D backup_full_17b2_1
```

Insertion de données dans la base de données

Nous ajoutons ensuite des données dans une table de la base de données pour générer des modifications qui seront prises en compte dans la sauvegarde incrémentale.

```
postgres=# INSERT INTO t1 SELECT * FROM generate_series(0, 10000);

INSERT 0 10001
```

Sauvegarde incrémentale

Après l'ajout de données, nous pouvons effectuer une sauvegarde incrémentale en utilisant l'option `--incremental` qui se base sur le manifeste de la sauvegarde précédente.

```
postgres@17:~$ pg_basebackup -D backup_inc_17b2_1
 ↪  --incremental=backup_full_17b2_1/backup_manifest
```

Combinaison des sauvegardes

Une fois la sauvegarde incrémentale effectuée, nous pouvons combiner la sauvegarde complète et incrémentale en une seule archive utilisable à l'aide de l'outil `pg_combinebackup` :

```
postgres@17:~$ pg_combinebackup backup_full_17b2_1 backup_inc_17b2_1
 ↪  --output=backup_combined_17b2_1
```

Vérification de la sauvegarde combinée

Le répertoire de la sauvegarde combinée contiendra une sauvegarde complète, prête à être utilisée pour une restauration :

```
postgres@17:~$ ls backup_combined_17b2_1/

backup_label     pg_dynshmem    pg_notify     pg_stat_tmp  pg_wal
backup_manifest  pg_hba.conf    pg_replslot   pg_subtrans  pg_xact
base             pg_ident.conf  pg_serial     pg_tblspc    postgresql.auto.conf
global           pg_logical     pg_snapshots  pg_twophase  postgresql.conf
pg_commit_ts     pg_multixact   pg_stat       PG_VERSION
```

Conclusion

L'introduction des sauvegardes incrémentales dans PostgreSQL simplifie la gestion des bases de données volumineuses en réduisant la taille des sauvegardes et le temps nécessaire pour les réaliser. Grâce à l'utilisation des sommaires WALs, seules les parties modifiées des bases de données sont sauvegardées, ce qui optimise les opérations de sauvegarde. Il faudra cependant faire attention, son utilisation étant destinée à des outils externes, il n'y a pas de vérification de continuité ni d'intégrité des sauvegardes, ce qui peut s'avérer fastidieux à faire manuellement.

3.2 OPTION `--FILTER` POUR PG_DUMP, PG_DUMPALL ET PG_RESTORE

- Nouvelle option `--filter` suivie d'un nom de fichier
- Format des lignes de ce fichier
  - commande objet motifobjet
- commande vaut
  - soit `include` pour inclure
  - soit `exclude` pour exclure
- objet vaut au choix
  - table_data, index, table_data_and_children, database,
  - extension, foreign_data, function, table, schema,
  - table_and_children ou trigger.
- motifobjet est une expression rationnelle pour le nom des objets à sélectionner

Cette option se comporte comme les options équivalentes en ligne de commande pour les outils `pg_dump`, `pg_dumpall` et `pg_restore`.

Cela permet d'éviter d'avoir à utiliser plusieurs fois les options `-t` / `-T` / `-n` / `-N`. Voici un exemple de fichier :

```
include table t1?
exclude table t12
```

Si ce contenu se trouve dans le fichier `test.txt`, une sauvegarde effectuée ainsi :

```
pg_dump --filter test.txt b1
```

prendra en compte toutes les tables dont le nom correspond à l'expression `t1?` à l'exception de la table `t12`.

3.3 AUTRES NOUVELLES OPTIONS PG_DUMP/PG_RESTORE

- Nouvelle option dans pg_dump
  - `--exclude-extension`
  - exclut les extensions correspondantes au motif indiqué
- Nouvelle option pour `pg_restore`
  - `--transaction-size`
  - Permet de gagner en performance
  - … tout en évitant les problèmes mémoires si en une seule transaction

L'option `--exclude-extension` de pg_dump permet d'exclure les extensions indiquées d'une sauvegarde logique. Par défaut, toutes les extensions sont prise en compte(ici au nombre de 3) :

```
$ pg_dump -U postgres -Fc > backup.pgdump

$ pg_restore -l backup.pgdump | grep EXTENSION
3; 3079 16508 EXTENSION - bloom
3451; 0 0 COMMENT - EXTENSION bloom
2; 3079 16427 EXTENSION - pg_trgm
3452; 0 0 COMMENT - EXTENSION pg_trgm
4; 3079 16519 EXTENSION - postgres_fdw
3453; 0 0 COMMENT - EXTENSION postgres_fdw
```

Pour en exclure certaines, l'option `--exclude-extension` peut être utilisée comme cela :

```
$ pg_dump -U postgres -Fc --exclude-extension=pg* > backup_no_pg_extension.pgdump

$ pg_restore -l backup_no_pg_extension.pgdump | grep EXTENSION
3; 3079 16508 EXTENSION - bloom
3451; 0 0 COMMENT - EXTENSION bloom
4; 3079 16519 EXTENSION - postgres_fdw
3452; 0 0 COMMENT - EXTENSION postgres_fdw
```

Toutes les extensions commençant par `pg*` ont bien été retirées de la sauvegarde logique.

Par défaut, l'outil `pg_restore` n'utilise pas de transactions explicites. Donc toutes les requêtes sont exécutées dans leur propre transaction implicite. Ceci est assez coûteux pour les performances mais a l'avantage de demander peu de mémoire.

Une solution revenait à utiliser l'option `--single-transaction` qui permettait d'englober l'intégralité du script SQL de restauration dans une transaction explicite. Autrement dit, `pg_restore` ajoutait automatiquement un `BEGIN` en début de restauration et un `COMMIT` en fin de restauration. Le problème qui survenait était l'utilisation d'un grand nombre de verrous conservés pendant toute la durée de la restauration. Parfois, la mémoire partagée ne suffisait pas et la restauration échouait.

Les développeurs de PostgreSQL proposent donc en version 17 un entre-deux qui permet de regrouper un certain nombre de requêtes du script SQL dans une transaction explicite. Ce nombre est indiqué par le paramètre en ligne de commande `--transaction-size`.

L'exemple suivant se limite à rechercher les `COMMIT` car il ne serait pas judicieux de placer toutes les requêtes dans ce support.

```
$ pg_restore --file=- b1.dump | grep COMMIT
-- sans option, nous obtenons l'ancien comportement par défaut
-- donc pas de transaction explicite

$ pg_restore --file=- --single-transaction b1.dump | grep COMMIT
COMMIT;
-- avec l'ancienne option --single-transaction,
-- tout est englobé dans une seule transaction

$ pg_restore --file=- --transaction-size=5 b1.dump | grep COMMIT
COMMIT;
COMMIT;
COMMIT;
COMMIT;
-- avec la nouvelle option, chaque fois que pg_restore atteint de nombre de requêtes,
-- il fait un COMMIT suivi d'un BEGIN
-- avec 20 requêtes et 5 pour --transaction-size, on se retrouve avec 4 transactions

$ pg_restore --file=- --transaction-size=10 b1.dump | grep COMMIT
COMMIT;
COMMIT;
-- alors qu'avec 10 pour --transaction-size, on se retrouve avec 2 transactions
```

4/ Réplication

4.1 FAILOVER DES SLOTS DE RÉPLICATION LOGIQUE

- Option `failover`
    - `pg_create_logical_replication_slot()`,
    - `CREATE SUBSCRIPTION`
- Synchronisation des slots de réplication
    - `pg_sync_replication_slots()`
- Fonction `sync_replication_slots`
    - automatiser la synchronisation

Cette version de PostgreSQL introduit une fonctionnalité qui manquait à la réplication logique depuis longtemps : la possibilité de maintenir la réplication logique malgré un switchover de la réplication physique sur la grappe de serveurs *provider*. Nous allons détailler ici les différents éléments qui constituent cette fonctionnalité.

Note : pour pouvoir exécuter les commandes ci-dessous, il faut configurer le paramètre `wal_level` à `logical`.

Propriété failover sur les slots de réplications

La fonction `pg_create_logical_replication_slot()` s'est vu ajoutée un nouveau paramètre `failover` :

```
\df pg_create_logical_replication_slot()
```

```
                                List of functions
                Name                 |                  Argument data types
-------------------------------------+-----------------------------------------------
 ↪  ---------------
 pg_create_logical_replication_slot | slot_name name, plugin name, temporary boolean
 ↪  DEFAULT false,
                                    | twophase boolean DEFAULT false, failover DEFAULT
 ↪  false,
                                    | OUT slot_name name, OUT lsn pg_lsn
(1 row)
```

Cette propriété indique que le slot de réplication logique sera synchronisé sur l'instance secondaire

d'un dispositif de réplication physique, ce qui permettra de reprendre la réplication logique en cas de switchover.

La vue `pg_replication_slots` a également été mise à jour afin de refléter la présence de cette propriété.

```
SELECT slots.*
  FROM (VALUES ('rl_test_f', true),('rl_test_nof', false)) AS slot(name,
   ↪  has_failover)
       CROSS JOIN LATERAL pg_create_logical_replication_slot(name, 'test_decoding',
        ↪  failover=>has_failover) AS slots;

  slot_name   |    lsn
--------------+-----------
 rl_test_f    | 0/1AFD380
 rl_test_nof  | 0/1AFD3B8
(2 rows)

SELECT slot_name, slot_type, failover FROM pg_replication_slots ;

  slot_name   | slot_type | failover
--------------+-----------+----------
 rl_test_f    | logical   | t
 rl_test_nof  | logical   | f
(2 rows)

SELECT pg_drop_replication_slot(name)
  FROM (VALUES ('rl_test_f'),('rl_test_nof')) AS slot(name);
```

L'option a également été ajoutée au protocole de réplication.

```
psql "dbname=postgres replication=database" <<_EOF_
CREATE_REPLICATION_SLOT test LOGICAL test_decoding (FAILOVER true);
SELECT slot_name, slot_type, failover FROM pg_replication_slots ;
_EOF_

 slot_name | consistent_point |    snapshot_name    | output_plugin
-----------+------------------+---------------------+---------------
 test      | 0/1AFD460        | 00000074-00000002-1 | test_decoding
(1 row)

 slot_name | slot_type | failover
-----------+-----------+----------
 test      | logical   | t
(1 row)
```

Une nouvelle commande a été ajoutée au protocole de réplication pour mettre à jour un slot :

```
psql "dbname=postgres replication=database" <<_EOF_
ALTER_REPLICATION_SLOT test (FAILOVER false);
```

```
SELECT slot_name, slot_type, failover FROM pg_replication_slots ;
_EOF_

ALTER_REPLICATION_SLOT
 slot_name | slot_type | failover
-----------+-----------+----------
 test      | logical   | f
(1 row)

psql "dbname=postgres replication=database" <<_EOF_
DROP_REPLICATION_SLOT test;
_EOF_
```

Ces mises à jour du protocole permettent à une standby de demander au *walsender* de créer / mettre à jour un slot et définir sa propriété `failover`.

Capacité de synchroniser des slots depuis une standby

L'étape suivante dans le développement de cette fonctionnalité a été de mettre en place la synchronisation des slots de réplications. Cette fonctionnalité ajoute des contraintes à la mise en place de la réplication physique :

- la réplication physique doit utiliser un slot de réplication. Les efforts réalisés pour invalider les slots si les standbys ne consomment pas les slots et améliorer les vues de supervision, font que ce n'est pas un risque pour la disponibilité du service sur l'instance primaire ;
- le `hot_standby_feedback` doit être activé. Cela peut avoir un impact sur le nettoyage des lignes mortes sur la primaire si des requêtes longues sont lancées sur la standby.
- la chaine de connexion spécifiée dans `primary_conninfo` doit contenir un nom de base de données valide.

Pour utiliser la synchronisation des slots, il faut :

- créer un slot sur la primaire avec l'attribut `failover` configuré à `true` ;
- utiliser l'option `failover` de la commande `CREATE SUBSCRIPTION` sur le serveur souscripteur.

Il faudra ensuite exécuter la fonction `pg_sync_replication_slots()` pour forcer la synchronisation du slot.

Une nouvelle colonne `synched` a été ajoutée à la vue `pg_replication_slots` pour indiquer que le slot est un slot synchronisé avec la primaire. En situation normale, il ne devrait donc pas y avoir de slots `synched` sur une instance primaire.

Un slot synchronisé ne peut pas être supprimé et son contenu ne peut pas être consommé.

Si le slot de réplication logique de l'instance primaire est invalidé, le slot synchronisé sur la secondaire sera également invalidé.

Si le slot de réplication logique de l'instance secondaire est invalidé sur la standby, il sera supprimé. Il faudra relancer la commande `pg_sync_replication_slots()` pour le recréer. Cette invalidation peut être due au fait que le `max_slot_wal_keep_size` de la standby est insuffisant pour retenir les enregistrements de WAL nécessaire pour satisfaire le `restart_lsn` du slot ou que le `primary_slot_name` a été supprimé.

Voici un exemple réalisé sous FEDORA avec PGDATAS=~/`tmp/failover` et PGUSER=`postgres`, le logging collector est activé, les sockets sont créés dans '/tmp', les traces et la configuration sont dans le répertoire de données. Toutes les connexions sont faites via les sockets, pour qui la méthode d'authentification est `trust` (donc pas d'authentification).

- mise en place d'une instance primaire :

```
initdb --pgdata=$PGDATAS/primaire \
       --set port=5495 \
       --set wal_level=logical \
       --set cluster_name=primaire \
       --username=postgres
pg_ctl start --pgdata $PGDATAS/primaire
createuser replicator --replication --port=5495
```

- mise en place d'une instance secondaire :

```
pg_basebackup --pgdata $PGDATAS/secondaire \
              --progress --checkpoint=fast \
              --create-slot \
              --slot=secondaire \
              --dbname="port=5495 user=replicator"
rm -f $PGDATAS/secondaire/log/*
touch $PGDATAS/secondaire/standby.signal
cat >> $PGDATAS/secondaire/postgresql.conf <<_EOF_
port = 5496
cluster_name = secondaire
primary_conninfo = 'port=5495 user=replicator dbname=postgres
 ↪  application_name=secondaire'
primary_slot_name = 'secondaire'
hot_standby_feedback = on
_EOF_
pg_ctl start --pgdata $PGDATAS/secondaire
```

- vérification de la réplication : les *walsenders* et *walreceiver* sont présents et la primaire est au statut streaming.

```
ps f -ee | grep -E "(primaire|secondaire)"
```

```
300207 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/primaire
300208 ?        Ss     0:00  |   \_ postgres: primaire: logger
300209 ?        Ss     0:00  |   \_ postgres: primaire: checkpointer
300210 ?        Ss     0:00  |   \_ postgres: primaire: background writer
300212 ?        Ss     0:00  |   \_ postgres: primaire: walwriter
300213 ?        Ss     0:00  |   \_ postgres: primaire: autovacuum launcher
300214 ?        Ss     0:00  |   \_ postgres: primaire: logical replication
 ↪  launcher
303235 ?        Ss     0:00  |   \_ postgres: primaire: walsender replicator
 ↪  [local] streaming 0/B000168
303229 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/secondaire
303230 ?        Ss     0:00      \_ postgres: secondaire: logger
303231 ?        Ss     0:00      \_ postgres: secondaire: checkpointer
303232 ?        Ss     0:00      \_ postgres: secondaire: background writer
303233 ?        Ss     0:00      \_ postgres: secondaire: startup recovering
 ↪  000000010000000000000000B
303234 ?        Ss     0:00      \_ postgres: secondaire: walreceiver streaming
 ↪  0/B000168
```

- création d'un slot de réplication synchronisé et synchronisation :

```
psql -p 5495 -c "SELECT pg_create_logical_replication_slot('test_sync',
 ↪  'pgoutput', failover=>'true');"
psql -p 5496 -c "SELECT pg_sync_replication_slots();"
```

- Vérifications :

```
psql <<_EOF_
\c postgres postgres /tmp 5495
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
\c postgres postgres /tmp 5496
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
_EOF_
```

```
You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5495".
 slot_name  | slot_type | failover | synced
------------+-----------+----------+--------
 secondaire | physical  | f        | f
 test_sync  | logical   | t        | f
(2 rows)

You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5496".
 slot_name | slot_type | failover | synced
-----------+-----------+----------+--------
```

```
 test_sync | logical   | t        | t
(1 row)
```

On voit que le slot est synchronisé sur la standby et que l'option `failover` est activée.

Comme promis, il n'est pas possible de supprimer le slot directement sur la standby.

```
psql <<_EOF_
-- serveur de secondaire
\c postgres postgres /tmp 5496
SELECT pg_drop_replication_slot('test_sync');
-- serveur de primaire
\c postgres postgres /tmp 5495
SELECT pg_drop_replication_slot('test_sync');
_EOF_
```

```
You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5496".
ERROR:  cannot drop replication slot "test_sync"
DETAIL:  This replication slot is being synchronized from the primary server.
You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5495".
 pg_drop_replication_slot
--------------------------
```

Une fois le slot supprimé sur la primaire, le slot est encore visible sur la standby. Il faut utiliser la fonction `pg_sync_replication_slots()` sur la standby pour le faire disparaitre.

```
psql <<_EOF_
-- serveur de primaire
\c postgres postgres /tmp 5495
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
-- serveur de secondaire
\c postgres postgres /tmp 5496
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
_EOF_
```

```
You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5495".
 slot_name  | slot_type | failover | synced
------------+-----------+----------+--------
 secondaire | physical  | f        | f
(1 row)

You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5496".
 slot_name | slot_type | failover | synced
-----------+-----------+----------+--------
```

```
 test_sync | logical   | t        | t
(1 row)

psql -p 5496 <<_EOF_
SELECT pg_sync_replication_slots();
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
_EOF_

You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5496".
 slot_name | slot_type | failover | synced
-----------+-----------+----------+--------
(0 rows)
```

- mise en place de la publication :

```
psql -p 5495 -c "CREATE PUBLICATION pub FOR TABLES IN SCHEMA public;"
```

- création d'une souscription et de l'instance associée :

```
initdb --pgdata=$PGDATAS/souscription \
       --set port=5497 \
       --set cluster_name=souscription \
       --username=postgres
pg_ctl start --pgdata $PGDATAS/souscription
psql <<_EOF_
-- serveur de souscription
\c postgres postgres /tmp 5497
CREATE SUBSCRIPTION sub
  CONNECTION 'port=5495 dbname=postgres'
  PUBLICATION pub
  WITH (failover);
-- serveur de secondaire
\c postgres postgres /tmp 5496
SELECT pg_sync_replication_slots();
_EOF_
```

- vérifications :

```
psql <<_EOF_
\c postgres postgres /tmp 5497
\x
\dRs+
\x
\c postgres postgres /tmp 5495
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
\c postgres postgres /tmp 5496
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
_EOF_
```

```
List of subscriptions
-[ RECORD 1 ]------+-----------------------------------------
Name               | sub
Owner              | postgres
Enabled            | t
Publication        | {pub}
Binary             | f
Streaming          | off
Two-phase commit   | d
Disable on error   | f
Origin             | any
Password required  | t
Run as owner?      | f
Failover           | t
Synchronous commit | off
Conninfo           | port=5495 user=replicator dbname=postgres
Skip LSN           | 0/0
You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5495".
 slot_name  | slot_type | failover | synced
------------+-----------+----------+--------
 secondaire | physical  | f        | f
 sub        | logical   | t        | f
(2 rows)

You are now connected to database "postgres" as user "postgres" via socket in
 ↪  "/tmp" at port "5496".
 slot_name | slot_type | failover | synced
-----------+-----------+----------+--------
 sub       | logical   | t        | t
(1 row)
```

Si on supprime le slot, il faut aussi relancer `pg_sync_replication_slots()` sur la standby.

```
psql <<_EOF_
-- serveur de souscription
\c postgres postgres /tmp 5497
DROP SUBSCRIPTION sub;
-- serveur de secondaire
\c postgres postgres /tmp 5496
SELECT pg_sync_replication_slots();
_EOF_
```

Synchronisation automatique des slots

La synchronisation des slots peut être automatiquement réalisée en activant le paramètre

`sync_replication_slots` sur l'instance secondaire.

Nous allons l'activer sur les deux instances puisque l'objectif est de réaliser un switchover.

```
echo "sync_replication_slots = true" >> $PGDATAS/primaire/postgresql.conf
pg_ctl restart -D $PGDATAS/primaire
echo "sync_replication_slots = true" >> $PGDATAS/secondaire/postgresql.conf
pg_ctl restart -D $PGDATAS/secondaire
```

On peut observer qu'un processus `slotsync worker` est démarré sur la standby :

```
ps f -e | grep -E "(primaire|secondaire)"
```

```
316406 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/primaire
316407 ?        Ss     0:00  |   \_ postgres: primaire: logger
316409 ?        Ss     0:00  |   \_ postgres: primaire: checkpointer
316410 ?        Ss     0:00  |   \_ postgres: primaire: background writer
316412 ?        Ss     0:00  |   \_ postgres: primaire: walwriter
316413 ?        Ss     0:00  |   \_ postgres: primaire: autovacuum launcher
316414 ?        Ss     0:00  |   \_ postgres: primaire: logical replication launcher
316441 ?        Ss     0:00  |   \_ postgres: primaire: walsender replicator [local]
 ↪  streaming 0/B0063C0
316442 ?        Ss     0:00  |   \_ postgres: primaire: replicator postgres [local]
 ↪  idle
316434 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/secondaire
316435 ?        Ss     0:00      \_ postgres: secondaire: logger
316436 ?        Ss     0:00      \_ postgres: secondaire: checkpointer
316437 ?        Ss     0:00      \_ postgres: secondaire: background writer
316438 ?        Ss     0:00      \_ postgres: secondaire: startup recovering
 ↪  000000010000000000000000B
316439 ?        Ss     0:00      \_ postgres: secondaire: walreceiver streaming
 ↪  0/B0063C0
316440 ?        Ss     0:00      \_ postgres: secondaire: slotsync worker
```

Recréons un slot de réplication :

```
psql -p 5497 <<_EOF_
CREATE SUBSCRIPTION sub
  CONNECTION 'port=5495 dbname=postgres'
  PUBLICATION pub
  WITH (failover);
_EOF_
```

La synchronisation des slots de réplication logique n'est pas instantanée, le `slotsync worker` se réveille à intervalle régulier et récupère les informations concernant les slots de réplication pour les créer ou les mettre à jour. La durée de repos du processus est adaptée en fonction de l'activité sur la primaire.

Après une période d'attente, le slot apparaît sur la secondaire :

```
psql <<_EOF_
\c postgres postgres /tmp 5497
\x
\dRs+
\x
\c postgres postgres /tmp 5495
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
\c postgres postgres /tmp 5496
SELECT slot_name, slot_type, failover, synced FROM pg_replication_slots;
_EOF_

You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5497".
Expanded display is on.
List of subscriptions
-[ RECORD 1 ]------+----------------------------------------
Name               | sub
Owner              | postgres
Enabled            | t
Publication        | {pub}
Binary             | f
Streaming          | off
Two-phase commit   | d
Disable on error   | f
Origin             | any
Password required  | t
Run as owner?      | f
Failover           | t
Synchronous commit | off
Conninfo           | port=5495 user=replicator dbname=postgres
Skip LSN           | 0/0

Expanded display is off.
You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5495".
 slot_name  | slot_type | failover | synced
------------+-----------+----------+--------
 secondaire | physical  | f        | f
 sub        | logical   | t        | f
(2 rows)

You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5496".
 slot_name | slot_type | failover | synced
-----------+-----------+----------+--------
```

```
 sub       | logical   | t        | t
(1 row)
```

Insertions de données

```
psql <<_EOF_
-- primaire / publication
\c postgres postgres /tmp 5495
CREATE TABLE junk(i int, t text);
INSERT INTO junk SELECT x, 'texte '||x FROM generate_series(1,10000) AS F(x);
-- souscription
\c postgres postgres /tmp 5497
CREATE TABLE junk(i int, t text);
ALTER SUBSCRIPTION sub REFRESH PUBLICATION;
SELECT pg_sleep_for(INTERVAL '2s');
SELECT count(*) FROM junk;
_EOF_
```

```
psql <<_EOF_
-- primaire / publication
\c postgres postgres /tmp 5495
SELECT slot_name, slot_type, failover, synced, restart_lsn FROM pg_replication_slots;
-- secondaire
\c postgres postgres /tmp 5496
SELECT slot_name, slot_type, failover, synced, restart_lsn FROM pg_replication_slots;
_EOF_
```

```
You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5495".
 slot_name  | slot_type | failover | synced | restart_lsn
------------+-----------+----------+--------+-------------
 secondaire | physical  | f        | f      | 0/3129A40
 sub        | logical   | t        | f      | 0/3129A08
(2 rows)

You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5496".
 slot_name | slot_type | failover | synced | restart_lsn
-----------+-----------+----------+--------+-------------
 sub       | logical   | t        | t      | 0/3129A08
(1 row)
```

Bascule controlée de la réplication (switchover)

Arrêter la primaire :

```
# primaire
pg_ctl stop -D $PGDATAS/primaire -m fast -w
```

```
pg_controldata -D $PGDATAS/primaire |\
  grep -E "(Database cluster state|Latest checkpoint's REDO location)"
```

Vérifier que la standby a tout reçu et la promouvoir :

```
# secondaire > nouvelle primaire
psql -p 5496 -c "CHECKPOINT;"
pg_controldata -D $PGDATAS/secondaire |\
  grep -E "(Database cluster state|Latest checkpoint's REDO location)"
pg_ctl promote -D $PGDATAS/secondaire
echo "cluster_name = 'nouvelle primaire'" >> $PGDATAS/secondaire/postgresql.conf
psql -p 5496 -c "SELECT pg_create_physical_replication_slot('ancienne_primaire');"
pg_ctl restart -D $PGDATAS/secondaire # pour màj les titres de processus pour la démo
psql -p 5496 -c "INSERT INTO junk SELECT x, 'texte '||x FROM
 ↪  generate_series(1,10000) AS F(x);"
```

Reconnecter l'ancienne primaire :

```
# ancienne primaire
touch $PGDATAS/primaire/standby.signal
cat >> $PGDATAS/primaire/postgresql.conf <<_EOF_
primary_conninfo = 'port=5496 user=replicator dbname=postgres
 ↪  application_name=ancienne_primaire'
primary_slot_name = 'ancienne_primaire'
hot_standby_feedback = on
cluster_name = 'ancienne primaire'
_EOF_
pg_ctl start -D $PGDATAS/primaire
```

Il faut ensuite mettre à jour la souscription :

```
psql -p 5497 -c "ALTER SUBSCRIPTION sub CONNECTION 'port=5496 dbname=postgres';"
```

… et supprimer les slot sur l'ancienne primaire :

```
psql -p 5495 -c "SELECT pg_drop_replication_slot('sub');"
psql -p 5495 -c "SELECT pg_drop_replication_slot('secondaire');"
```

La réplication logique a bien basculé :

```
ps f -e | grep -E "(primaire|secondaire|souscription)"
```

```
330608 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/souscription
330609 ?        Ss     0:00  |   \_ postgres: souscription: logger
330610 ?        Ss     0:00  |   \_ postgres: souscription: checkpointer
330611 ?        Ss     0:00  |   \_ postgres: souscription: background writer
330613 ?        Ss     0:00  |   \_ postgres: souscription: walwriter
330614 ?        Ss     0:00  |   \_ postgres: souscription: autovacuum launcher
```

```
330615 ?        Ss     0:00  |   \_ postgres: souscription: logical replication
 ↪  launcher
335150 ?        Ss     0:00  |   \_ postgres: souscription: logical replication apply
 ↪  worker for subscription 16384
333983 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/primaire
333984 ?        Ss     0:00  |   \_ postgres: ancienne primaire: logger
333985 ?        Ss     0:00  |   \_ postgres: ancienne primaire: checkpointer
333986 ?        Ss     0:00  |   \_ postgres: ancienne primaire: background writer
333987 ?        Ss     0:00  |   \_ postgres: ancienne primaire: startup recovering
 ↪  000000020000000000000003
334439 ?        Ss     0:00  |   \_ postgres: ancienne primaire: walreceiver
 ↪  streaming 0/31E1B50
336165 ?        Ss     0:00  |   \_ postgres: ancienne primaire: slotsync worker
334371 ?        Ss     0:00  \_ /usr/pgsql-17/bin/postgres -D
 ↪  /home/benoit/tmp/failover/secondaire
334372 ?        Ss     0:00      \_ postgres: nouvelle primaire: logger
334373 ?        Ss     0:00      \_ postgres: nouvelle primaire: checkpointer
334374 ?        Ss     0:00      \_ postgres: nouvelle primaire: background writer
334376 ?        Ss     0:00      \_ postgres: nouvelle primaire: walwriter
334377 ?        Ss     0:00      \_ postgres: nouvelle primaire: autovacuum launcher
334378 ?        Ss     0:00      \_ postgres: nouvelle primaire: logical replication
 ↪  launcher
334440 ?        Ss     0:00      \_ postgres: nouvelle primaire: walsender replicator
 ↪  [local] streaming 0/31E1B50
335156 ?        Ss     0:00      \_ postgres: nouvelle primaire: walsender postgres
 ↪  postgres [local] START_REPLICATION
336169 ?        Ss     0:00      \_ postgres: nouvelle primaire: replicator postgres
 ↪  [local] idle
```

Les données ont bien été synchronisées :

```
psql <<_EOF_
-- nouvelle primaire / publication
\c postgres postgres /tmp 5496
INSERT INTO junk SELECT x, 'texte '||x FROM generate_series(1,10000) AS F(x);
SELECT count(*) FROM junk;
-- souscription
\c postgres postgres /tmp 5497
SELECT pg_sleep_for(INTERVAL '2s');
SELECT count(*) FROM junk;
_EOF_
```

```
_EOF_
You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
 ↪  at port "5496".
INSERT 0 10000
 count
```

```
-------
 30000
(1 row)

You are now connected to database "postgres" as user "postgres" via socket in "/tmp"
↪  at port "5497".
 pg_sleep_for
--------------

(1 row)

 count
-------
 30000
(1 row)
```

Bascule non programmée (failover)

Si une bascule non programmée est réalisée et que pour une raison ou pour une autre les deux instances divergent, il est possible que la souscription reçoive des informations qui ne sont pas arrivées sur la nouvelle instance primaire. Cela pourrait causer des conflits de réplication logique par la suite et/ou provoquer des bugs au niveau applicatif.

4.2 NOUVELLES COLONNES POUR PG_REPLICATION_SLOTS

- `inactive_since`
  - depuis combien de temps le slot est-il inactif ?
- `invalidation_reason`
  - raison pour laquelle un slot est invalidé

La colonne `inactive_since` a été ajoutée à la vue `pg_replication_slot`, elle permet de déterminer depuis quand un slot est inactif. Cette fonctionnalité est utile pour identifier les slots inactifs, et pour diagnostiquer ou mesurer les indisponibilités des clients de réplication. Elle pourra servir de base pour une éventuelle future fonctionnalité permettant d'invalider les slots de réplication en se basant sur un timeout.

La seconde colonne ajoutée à cette vue est `invalidation_reason`. Elle permet d'avoir des informations sur la raison pour laquelle un slot a été invalidé :

- pour tous les types de réplication :
  - `NULL` : le slot n'a pas été invalidé ;
  - `wal_removed` : les WAL requis ont été supprimés (voir `wal_status` et `safe_wal_size`) ;
- seulement pour la réplication logique :
  - `rows_removed` : les lignes requises ont été supprimées ;
  - `wal_level_insufficient` : le `wal_level` de la source ne permet pas de décoder les WAL.

Les colonnes `synced` et `failover` ont également été ajoutées et sont discutées dans le chapitre sur la synchronisation des slots de réplication logique et leur *failover*.

4.3 NOUVEAUX MESSAGES DES *WALSENDERS*

- Ajoute les messages suivants dans les trace lorsqu'un *walsender* acquiert ou relâche un slot de réplication :

```
LOG:  acquired logical replication slot "nom-slot"
LOG:  acquired physical replication slot "nom-slot"
LOG:  eleased logical replication slot "nom-slot"
LOG:  released physical replication slot "nom-slot"
```

- Nécessite
  - d'activer `log_replication_commands`
  - ou d'avoir le niveau de trace DEBUG1 actif

4.4 OUTIL PG_CREATESUBSCRIBER

- Part d'une instance primaire démarrée et une standby arrêtée
- Met en place les publications, slots et souscriptions
- Utile pour grosses publications

L'outil `pg_createsubscriber` a pour objet de faciliter la mise en place de souscription pour de grosses publications. En effet, lorsque de gros volumes de données doivent être transférés lors de la phase de copie initiale, l'opération est longue et peut mobiliser beaucoup de ressources sur l'instance de la publication. De plus, en raison de la durée de la copie initiale, un gros retard peut s'accumuler ce qui force les slots de réplication logique à conserver de gros volumes de données.

L'outil permet donc de se baser sur une instance mise en réplication avec la réplication physique pour créer une souscription. L'instance secondaire aura été créée avec `pg_basebackup` ou, mieux, un outil de sauvegarde tiers, ce qui permet de se protéger contre les problèmes listés ci-dessus.

Voici un exemple:

Une instance secondaire a été créée avec `pg_basebackup`, un slot a été déclaré sur la primaire.

```
-- sur l'instance secondaire
\dconfig primary_conninfo|primary_slot_name
```

```
                        List of configuration parameters
     Parameter     |                              Value
-------------------+-----------------------------------------------------------------
 primary_conninfo  | port=5443 user=replication application_name=pgsql-17-replica
 primary_slot_name | pgsql_17_replica
(2 rows)
```

La réplication fonctionne :

```
-- sur l'instance primaire
SELECT r.pid,
       r.application_name,
       r.state,
       r.sync_state,
       r.sent_lsn,
       r.write_lsn,
       s.slot_name,
       s.slot_type,
       s.active,
```

```
       s.restart_lsn,
       s.wal_status
FROM pg_stat_replication AS r
     INNER JOIN pg_replication_slots AS s ON r.pid = s.active_pid

  -[ RECORD 1 ]----+-----------------
pid              | 49908
application_name | pgsql-17-replica
state            | streaming
sync_state       | async
sent_lsn         | 0/3000928
write_lsn        | 0/3000928
slot_name        | pgsql_17_replica
slot_type        | physical
active           | t
restart_lsn      | 0/3000928
wal_status       | reserved
```

La base primaire contient deux bases applicatives que l'on souhaite répliquer via la réplication logique :

```
  datname
-----------
 postgres
 app1
 template1
 template0
 app2
(5 rows)
```

Afin de pouvoir mettre en place la réplication, il faut s'assurer que le paramétrage est correct :

```
-- sur l'instance primaire
\dconfig+   wal_level|max_replication_slots|max_wal_senders

                      List of configuration parameters
       Parameter       |  Value  |  Type   |  Context   | Access privileges
-----------------------+---------+---------+------------+-------------------
 max_replication_slots | 10      | integer | postmaster | ¤
 max_wal_senders       | 10      | integer | postmaster | ¤
 wal_level             | logical | enum    | postmaster | ¤
(3 rows)
```

Le nombre de slots de réplication et de processus *wal senders* doit suffire pour mettre en place une publication par base à répliquer. Bien entendu, il faut aussi faire en sorte que les WAL contiennent toutes les informations nécessaires à l'utilisation de la réplication logique (`wal_level = logical`).

```
-- sur l'instance primaire
\dconfig+
 ↪  max_replication_slots|max_logical_replication_workers|max_worker_processes

                          List of configuration parameters
            Parameter             | Value |  Type   |  Context   | Access privileges
----------------------------------+-------+---------+------------+-------------------
 max_logical_replication_workers | 4     | integer | postmaster | ¤
 max_replication_slots           | 10    | integer | postmaster | ¤
 max_worker_processes            | 8     | integer | postmaster | ¤
(3 rows)
```

L'instance secondaire a assez de processus *worker* pour démarrer les processus de réplication. Il faut à minima un processus par souscription, voire plus si l'application parallélisée est configurée. L'outil vérifie que :

- `max_logical_replication_workers` : supérieur ou égal au nombre de bases à répliquer ;
- `max_worker_processes` : supérieur au nombre de bases à répliquer.

Il ne faut pas oublier que si le parallélisme est utilisé pour les requêtes ou des tâches de maintenances, les workers sont pris dans le même pool de workers défini par `max_worker_processes`.

Il faut aussi un nombre de slots de réplication supérieur ou égal au nombre de bases à répliquer. Nous verrons pourquoi plus loin.

L'outil nécessite de spécifier un utilisateur pour la réplication et pour la connexion de l'instance de publication. Ici, par souci de simplicité, nous avons réutilisé l'utilisateur de réplication physique pour les deux types de connexions. Nous lui avons rajouté le groupe `pg_read_all_data`, et, temporairement, l'attribut SUPERUSER.

SUPERUSER est nécessaire pour l'utilisateur qui se connecte à la future instance de publication afin de pouvoir créer une publication `FOR ALL TABLES`.

La commande de mise en place de la réplication logique se lance comme suit :

```
pg_createsubscriber \
  --pgdata $PGDATA \
  --subscriber-port 5445 \
  --subscriber-username replication \
  --publisher-server "port=5443 user=replication application_name=pgsql-17-replica" \
  --database app1 --database app2 \
  --publication pub_app1 --publication pub_app2 \
  --subscription sub_app1 --subscription sub_app2 \
  --verbose
```

Il est possible de :

- faire un essai à blanc avec `--dry-run` (les traces ne sont pas très claires à ce sujet) ;
- spécifier une durée d'attente maximale afin d'attendre que la réplication rattrape un éventuel retard (`--recovery-timeout`) ;
- spécifier où se trouve le fichier de configuration de l'instance, ceci est intéressant pour des installations où la configuration n'est pas dans le répertoire de données (notamment sur Debian) avec `--config-file`. C'est nécessaire car l'outil doit redémarrer l'instance standby. Pour cela, il utilise `pg_ctl`.

La commande se déroule en plusieurs phases :

- vérification que l'instance secondaire est bien arrêtée ;
- validation des prérequis :

```
pg_createsubscriber: validating publisher connection string
pg_createsubscriber: validating subscriber connection string
pg_createsubscriber: checking if directory
 ↪  "/home/benoit/var/lib/postgres/pgsql-17-replica" is a cluster data
 ↪  directory
pg_createsubscriber: getting system identifier from publisher
pg_createsubscriber: system identifier is 7442627096879726047 on publisher
pg_createsubscriber: getting system identifier from subscriber
pg_createsubscriber: system identifier is 7442627096879726047 on subscriber
pg_createsubscriber: starting the standby server with command-line options
2024-11-29 11:14:55.763 CET [56874] LOG:  redirecting log output to logging
 ↪  collector process
2024-11-29 11:14:55.763 CET [56874] HINT:  Future log output will appear in
 ↪  directory "log".
pg_createsubscriber: server was started
pg_createsubscriber: checking settings on subscriber
pg_createsubscriber: checking settings on publisher
pg_createsubscriber: stopping the subscriber
pg_createsubscriber: server was stopped
```

- création des publications et slots de réplication sur l'instance de primaire :
  - la publication est créée avec `FOR ALL TABLES`.
  - si aucun nom n'est fourni pour la publication et les slots sont créés avec le nom : `pg_createsubscriber_<database oid>_<numéro aléatoire>`
  - le dernier lsn de réplication du slot est conservé pour démarrer l'instance secondaire avec un `recovery_target_lsn` correspondant ce qui permet de garantir qu'aucune donnée ne sera perdue.

```
pg_createsubscriber: creating publication "pub_app1" in database "app1"
pg_createsubscriber: creating the replication slot "sub_app1" in database "app1"
pg_createsubscriber: create replication slot "sub_app1" on publisher
```

```
pg_createsubscriber: creating publication "pub_app2" in database "app2"
pg_createsubscriber: creating the replication slot "sub_app2" in database "app2"
pg_createsubscriber: create replication slot "sub_app2" on publisher
```

- démarrage de l'instance secondaire avec le `recovery_target_lsn`, et `recovery_target_action = promote` si `--recovery-timeout` a été spécifié, cette étape peut échouer.

  Comme le précisent les traces, en cas d'erreur après ce stade, la standby doit être recréée.

```
pg_createsubscriber: starting the subscriber
2024-11-29 11:14:56.257 CET [56892] LOG:  redirecting log output to logging
 ↪  collector process
2024-11-29 11:14:56.257 CET [56892] HINT:  Future log output will appear in
 ↪  directory "log".
pg_createsubscriber: server was started
pg_createsubscriber: waiting for the target server to reach the consistent state
pg_createsubscriber: target server reached the consistent state
pg_createsubscriber: hint: If pg_createsubscriber fails after this point, you
 ↪  must recreate the physical replica before continuing.
```

- Création des souscriptions : si aucun nom de souscription n'est fourni, un nom est généré sur le même modèle que les publications et slots.

  Lors de la création des publications et slots sur l'instance primaire, la réplication physique était active, ils sont donc présents sur l'instance de souscription. C'est la raison pour laquelle `max_replication_slots` doit être supérieur ou égal au nombre de bases à répliquer.

```
pg_createsubscriber: dropping publication "pub_app1" in database "app1"
pg_createsubscriber: creating subscription "sub_app1" in database "app1"
pg_createsubscriber: setting the replication progress (node name "pg_24626",
 ↪  LSN 0/D00F088) in database "app1"
pg_createsubscriber: enabling subscription "sub_app1" in database "app1"
pg_createsubscriber: dropping publication "pub_app2" in database "app2"
pg_createsubscriber: creating subscription "sub_app2" in database "app2"
pg_createsubscriber: setting the replication progress (node name "pg_24627",
 ↪  LSN 0/D00F088) in database "app2"
pg_createsubscriber: enabling subscription "sub_app2" in database "app2"
```

- Si un slot de réplication était utilisé pour la réplication physique, il est supprimé :

```
pg_createsubscriber: dropping the replication slot "pgsql_17_replica" in
 ↪  database "app1"
```

- L'instance de souscription est ensuite arrêtée et l'identifiant du système est changé.

```
pg_createsubscriber: stopping the subscriber
pg_createsubscriber: server was stopped
```

```
     pg_createsubscriber: modifying system identifier of subscriber
     pg_createsubscriber: system identifier is 7442642731418717731 on subscriber
     pg_createsubscriber: running pg_resetwal on the subscriber
     pg_createsubscriber: subscriber successfully changed the system identifier
     pg_createsubscriber: Done!
```

À ce stade l'instance de souscription est arrêtée.

Après son démarrage, on peut voir la réplication démarrer sur l'instance de publication :

```
-- sur l'instance primaire
SELECT r.pid,
       r.application_name,
       r.state,
       r.sync_state,
       r.sent_lsn,
       r.write_lsn,
       s.slot_name,
       s.slot_type,
       s.active,
       s.restart_lsn,
       s.wal_status
FROM pg_stat_replication AS r
     INNER JOIN pg_replication_slots AS s ON r.pid = s.active_pid
```

```
-[ RECORD 1 ]----+-----------------
pid              | 57672
application_name | pgsql-17-replica
state            | streaming
sync_state       | async
sent_lsn         | 0/D01C778
write_lsn        | 0/D01C778
slot_name        | sub_app1
slot_type        | logical
active           | t
restart_lsn      | 0/D01C638
wal_status       | reserved
-[ RECORD 2 ]----+-----------------
pid              | 57674
application_name | pgsql-17-replica
state            | streaming
sync_state       | async
sent_lsn         | 0/D01C778
write_lsn        | 0/D01C778
slot_name        | sub_app2
slot_type        | logical
active           | t
restart_lsn      | 0/D01C638
```

```
wal_status       | reserved
```

Les publications sont visibles **dans leurs bases respectives** (la capture suivante est éditée) :

```
-- sur l'instance de publication, dans chaque base
\dRp
```

```
                                List of publications
   Name   |    Owner    | All tables | Inserts | Updates | Deletes | Truncates | Via
 ↪  root
----------+-------------+------------+---------+---------+---------+-----------+----
 ↪  ------
 pub_app1 | replication | t          | t       | t       | t       | t         | f
 pub_app2 | replication | t          | t       | t       | t       | t         | f
(1 row)
```

De même, sur l'instance de souscription, les souscriptions sont visibles **dans leurs bases respectives** (la capture suivante est éditée) :

```
-- sur l'instance de souscription, dans chaque base
\dRs
```

```
             List of subscriptions
   Name   |    Owner    | Enabled | Publication
----------+-------------+---------+-------------
 sub_app1 | replication | t       | {pub_app1}
 sub_app2 | replication | t       | {pub_app2}
```

L'état de la réplication peut aussi être observé sur l'instance de souscription aussi :

```
TABLE pg_stat_subscription;
```

```
-[ RECORD 1 ]---------+------------------------------
subid                 | 24626
subname               | sub_app1
worker_type           | apply
pid                   | 57671
leader_pid            | ¤
relid                 | ¤
received_lsn          | 0/D01C778
last_msg_send_time    | 2024-11-29 11:27:18.930714+01
last_msg_receipt_time | 2024-11-29 11:27:18.930843+01
latest_end_lsn        | 0/D01C778
latest_end_time       | 2024-11-29 11:27:18.930714+01
-[ RECORD 2 ]---------+------------------------------
subid                 | 24627
subname               | sub_app2
worker_type           | apply
pid                   | 57673
```

```
leader_pid            | ¤
relid                 | ¤
received_lsn          | 0/D01C778
last_msg_send_time    | 2024-11-29 11:27:18.94146+01
last_msg_receipt_time | 2024-11-29 11:27:18.941589+01
latest_end_lsn        | 0/D01C778
latest_end_time       | 2024-11-29 11:27:18.94146+01
```

```
TABLE pg_stat_subscription_stats ;
```

```
 subid | subname  | apply_error_count | sync_error_count | stats_reset
-------+----------+-------------------+------------------+-------------
 24626 | sub_app1 |                 0 |                0 | ¤
 24627 | sub_app2 |                 0 |                0 | ¤
(2 rows)
```

À ce stade, il faut supprimer l'attribut SUPERUSER de l'utilisateur réplication **sur les deux instances**.

5/ Supervision

5.1 NOUVELLE VUE PG_STAT_CHECKPOINTER

- Nouvelle vue de statistiques sur le checkpointer
- Auparavant, `pg_stat_bgwriter` pour les statistiques sur
  - bgwriter
  - checkpointer
- Maintenant
  - vue pour bgwriter : `pg_stat_bgwriter`
  - vue pour checkpointer : `pg_stat_checkpointer`

La vue `pg_stat_bgwriter` avait des colonnes pour le `bgwriter` et pour le `checkpointer`. La version 17 améliore cela en proposant une vue pour chaque processus, avec plus d'informations.

La vue `pg_stat_bgwriter` se voit donc amputée de plusieurs colonnes, qui sont maintenant dans la nouvelle vue `pg_stat_checkpointer`. Cette dernière se voit ajouter aussi de nouvelles colonnes pour les restart points.

```
postgres=# select * from pg_stat_bgwriter \gx
-[ RECORD 1 ]----+------------------------------
buffers_clean    | 0
maxwritten_clean | 0
buffers_alloc    | 540
stats_reset      | 2024-07-10 15:15:57.164507+02

postgres=# select * from pg_stat_checkpointer \gx
-[ RECORD 1 ]-------+------------------------------
num_timed           | 3
num_requested       | 0
restartpoints_timed | 0
restartpoints_req   | 0
restartpoints_done  | 0
write_time          | 4537
sync_time           | 20
buffers_written     | 45
```

```
stats_reset          | 2024-07-10 15:15:57.164507+02
```

Les développeurs en ont profité pour supprimer les deux colonnes sur les backends (`buffers_backend` et `buffers_backend_fsync`). EN effet, il est aussi possible de calculer ces valeurs à partir de la vue `pg_stat_io`. Par exemple, pour la colonne `buffers_backend` :

```
SELECT sum(writes*op_bytes/8192) AS buffers_backend
FROM stat_io
WHERE backend_type='client backend'
  AND object='relation'
```

Enfin, il est possible de réinitialiser les statistiques de la nouvelle vue avec un appel à la fonction `pg_stat_reset_shared()` en lui fournissant l'argument `checkpointer`.

5.2 NOUVELLES COLONNES POUR PG_STAT_PROGRESS_VACUUM

- `indexes_total`, nombre d'index à traiter
- `indexes_processed`, nombre d'index traités
- S'incrémentent pendant les phases
    - `vacuuming indexes`
    - `cleaning up indexes`

La vue de progression du `VACUUM`, appelée `pg_stat_progress_vacuum`, indiquait le nombre de lignes traitées dans la table, ce qui permettait de suivre finement certaines phases du `VACUUM`. Cependant, les phases de nettoyage des index de cette table ne permettait pas cette observation fine. Il était possible de voir qu'il traitait les index, mais pas combien étaient déjà traités. Dans le cas de tables très volumineuses, on pouvait avoir l'impression que le traitement était bloqué.

La situation s'améliore un peu avec la version 17. Nous pouvons maintenant savoir combien d'index ont été traités et combien d'index en total doivent être traités. Nous aurions certainement préféré voir aussi sa progression dans chaque index, mais ce sera certainement pour une prochaine version.

```
postgres=# \d pg_stat_progress_vacuum
            View "pg_catalog.pg_stat_progress_vacuum"
        Column         |  Type   | Collation | Nullable | Default
-----------------------+---------+-----------+----------+---------
 pid                   | integer |           |          |
 datid                 | oid     |           |          |
 datname               | name    |           |          |
 relid                 | oid     |           |          |
 phase                 | text    |           |          |
 heap_blks_total       | bigint  |           |          |
 heap_blks_scanned     | bigint  |           |          |
 heap_blks_vacuumed    | bigint  |           |          |
 index_vacuum_count    | bigint  |           |          |
 max_dead_tuple_bytes  | bigint  |           |          |
 dead_tuple_bytes      | bigint  |           |          |
 num_dead_item_ids     | bigint  |           |          |
```

```
 indexes_total        | bigint  |           |          |
 indexes_processed    | bigint  |           |          |
```

5.3 NOUVELLES TRACES D'UNE RECOVERY

- Trois nouveaux messages dans les traces

```
Recovery has started from a backup_label.
Recovery is restarting from a backup start LSN, without a backup_label.
Recovery has completed from a backup.
```

Ces messages permettent d'avoir une trace des conditions dans lesquelles une instance a démarré sa *recovery* lors d'une restauration. Ils permettent de confirmer si un fichier *backup label* a été utilisé.

5.4 NOUVELLES TRACES DES CONNEXIONS TRUST

- Nouvelle ligne dans les traces
  - `log_connections = 'on'`
  - Connexions utilisant la méthode `trust`

Une modification dans le code de l'authentification permet de tracer les connexions qui se font alors qu'aucun `authn ID` n'est renseigné. C'est par exemple le cas de la méthode de connexion `trust`. Désormais, lorsque le paramètre `log_connections` est à on, une ligne sera affichée dans les traces indiquant la connexion en question :

```
LOG:  connection received: host=[local]
LOG:  connection authenticated: user="postgres" method=trust
↪  (/home/pierrick/.pgenv/pgsql/data/pg_hba.conf:117)
LOG:  connection authorized: user=postgres database=postgres application_name=psql
```

Rien ne change pour les autres méthodes de connexion.

5.5 NOUVELLE VUE PG_WAIT_EVENTS

- Informations sur les `wait_events`
- `type`, `name`, `description`

Cette nouvelle vue permet de récupérer des informations concernant les `wait events`. Pour le moment, trois champs existent : `type`, `name`, `description`. Les descriptions sont les mêmes que celles qui se trouvent dans la documentation officielle. À l'heure actuelle, les informations sont très limitées, mais cette vue sera très probablement complétée dans le futur.

L'intérêt est de pouvoir faire une jointure entre cette vue et la vue `pg_stat_activity`. De cette manière, il sera possible de savoir pourquoi une requête est en attente.

Voici un exemple de jointure entre ces deux vues :

```
SELECT a.pid, a.query, a.wait_event, w.description
   FROM pg_stat_activity a JOIN
        pg_wait_events w ON (a.wait_event_type = w.type AND
                             a.wait_event = w.name)
   WHERE wait_event is NOT NULL and a.state = 'active';
```

Prenons le cas d'une opération de VACUUM qui semble être bloquée. Avec la requête précédente on aurait comme résultat :

```
postgres=# SELECT a.pid, a.query, a.wait_event, w.description
   FROM pg_stat_activity a JOIN
        pg_wait_events w ON (a.wait_event_type = w.type AND
                             a.wait_event = w.name)
   WHERE wait_event is NOT NULL and a.state = 'active';
  pid   |   query    | wait_event |              description
--------+------------+------------+---------------------------------------
 172181 | vacuum t1; | relation   | Waiting to acquire a lock on a relation
(1 row)
```

On sait donc que l'opération de VACUUM attend de pouvoir acquérir un verrou sur la relation `t1`.

6/ Performance

6.1 REGROUPEMENT DES I/O

- Un premier pas vers une gestion plus optimale des I/O
- Vectored I/O, Direct I/O, et Asynchronous I/O
- Nouveau paramètre `io_combine_limit`
  - 128 ko par défaut
  - configurable entre 8 ko et 256 ko avec des blocs de taille standard

Une nouvelle infrastructure est mise en place afin de gérer de façon plus optimale les I/O. Pour le moment, PostgreSQL n'utilise pas de *direct I/O*, contrairement à Oracle par exemple. Il y a donc un problème de double *buffering* : des blocs de tables ou index peuvent se trouver à la fois dans le cache interne de PostgreSQL et dans le cache système (*page cache*). Dans certaines conditions, on observe une nette dégradation de performance à cause d'un va-et-vient permanent entre les deux caches. Et dans tous les cas, la double copie des données en mémoire a un coût qu'il serait avantageux d'économiser. Depuis la version 16, le paramètre `debug_io_direct` permet à PostgreSQL d'utiliser les *direct I/O* pour les données (tables et index) ou pour les WAL, ou encore pour les deux. Ce paramètre est avant tout destiné aux tests et il est déconseillé de l'activer en production. En effet, les performances sont généralement moins bonnes, car une utilisation efficace de ce mode « direct » implique préalablement de pouvoir gérer les I/O de façon asynchrone [1], ce qui n'est pas encore possible avec PostgreSQL.

Mais une infrastructure permettant cela est progressivement mise en place. La version 17 expose une nouvelle API, avec les fonctions `read_stream_*()`, qui permet au code client (par exemple le code de ANALYZE) de fournir un *callback* qui va énumérer la suite des numéros de blocs à lire, plutôt que de faire un appel à la fonction `ReadBuffer()` pour chacun d'eux. Cela permet d'implémenter, de façon générique, un mécanisme qui peut faire beaucoup de choses intéressantes : *prefetching*, *read-ahead*, lectures asynchrones, etc. Une des fonctionnalités déjà implémentées est le fait de pouvoir regrouper plusieurs lectures de blocs en un seul appel système `preadv()`. Cela a pour principal avantage de réduire le nombre d'appels système. Avec la valeur par défaut de `io_combine_limit` et des blocs de taille standard, on a donc jusqu'à 16 fois moins d'appels système pour la lecture des blocs en de-

[1] avec les *direct I/O*, le noyau ne fait plus de *read-ahead* (ça n'aurait pas de sens sans *page cache*). Le *prefetch* demandé explicitement par PostgreSQL (avec l'appel système `posix_fadvise()`), utilisé notamment par `analyze`, `vacuum`, et le *bitmap heap scan* devient impossible lui aussi. Il faut donc un vrai système d'I/O asynchrone pour retrouver de bonnes performances. Une piste en cours de test serait d'utiliser la relativement récente interface noyau `io_uring`. Une autre piste un peu plus avancée est d'utiliser un *background worker* dédié pour ces lectures asynchrones.

hors du cache PostgreSQL. Nous conseillons de ne pas changer la valeur de ce paramètre à moins d'avoir compris très précisément ce mécanisme et que des tests aient démontré qu'un changement était bénéfique. En version 17, seule l'implémentation du parcours séquentiel, celle de `ANALYZE`, et celle de `pg_prewarm` utilisent cette nouvelle API.

6.2 SUPPRESSION DE LA LIMITE MÉMOIRE DE VACUUM

- Configuration de `maintenance_work_mem`
- Plus de limite à 1 Go pour le VACUUM

Auparavant, les identifiants de tuples morts étaient stockés dans un tableau limité à 1 Go. De ce fait, lorsque la valeur du paramètre `maintenance_work_mem` était supérieure à 1 Go, cela n'avait pas d'effet pour le VACUUM. Il en va de même pour le paramètre `autovacuum_work_mem`, lui aussi limité silencieusement à 1 Go.

Désormais, une nouvelle structure de données, le TIDStore, permet le stockage des TID (identifiants de tuples) en utilisant un arbre radix. Comme l'arbre radix effectue de petites allocations en fonction des besoins, la limite de 1 Go a maintenant été supprimée. En outre, la mémoire totale utilisée est souvent réduite de manière significative. La recherche des TID pendant le nettoyage d'index est plus rapide, surtout lorsque l'ordre des index correspond à l'ordre des tuples.

Étant donné qu'il n'existe plus de relation prévisible entre le nombre de tuples morts nettoyés par le VACUUM et l'espace occupé par leurs TIDs, le nombre de tuples ne fournit plus d'informations significatives pour les utilisateurs, et le nombre maximum n'est plus prévisible. Pour cette raison, les colonnes `max_dead_tuples` et `num_dead_tuples` de la vue `pg_stat_progress_vacuum` sont renommées `max_dead_tuple_bytes` et `dead_tuple_bytes`.

6.3 NOUVELLES OPTIONS POUR EXPLAIN

- SERIALIZE
  - temps et la quantité de données retournés au client
- MEMORY
  - mémoire allouée et utilisée par le planificateur

Option SERIALIZE

Cette nouvelle option d'EXPLAIN permet de connaitre le temps ainsi que la quantité de données qui a été transmisse au client. La quantité de données correspond à ce qui a été envoyé au client. Le temps correspond à la conversion des données dans le format d'échange du le protocole utilisé par PostgreSQL.

```
postgres=# explain (analyze) select * from tab1;
                                                  QUERY PLAN

-----------------------------------------------------------
 Seq Scan on tab1  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.037..0.08
3 rows=1000 loops=1)
 Planning Time: 0.070 ms
 Execution Time: 0.124 ms
(3 rows)
```

Une nouvelle ligne apparait dans la sortie de la commande.

```
postgres=# explain (analyze, serialize) select * from tab1;
                                                  QUERY PLAN

-----------------------------------------------------------
 Seq Scan on tab1  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.019..0.12
3 rows=1000 loops=1)
 Planning Time: 0.053 ms
 Serialization: time=0.070 ms  output=6kB  format=text
 Execution Time: 0.303 ms
(4 rows)
```

Cette option doit nécessairement être utilisée avec ANALYZE, autrement un message d'erreur apparaîtra.

```
postgres=# explain (serialize) select * from tab1;
ERROR:  EXPLAIN option SERIALIZE requires ANALYZE
```

MEMORY

Cette nouvelle option d'`EXPLAIN` permet de connaitre la quantité de mémoire qui a été utilisée par le planificateur lors de l'exécution de la requête (`used`). Cette information est complétée par la quantité de mémoire qui a été initialement allouée (`allocated`).

Exemple de sortie obtenue :

```
postgres=# explain (analyse, memory) select * from tab1;
                                                QUERY PLAN
-----------------------------------------------------------------------------------
 Seq Scan on tab1  (cost=0.00..22.70 rows=1270 width=36)
                   (actual time=0.023..0.141 rows=1000 loops=1)
 Planning:
   Memory: used=7kB  allocated=8kB
 Planning Time: 0.087 ms
 Execution Time: 0.224 ms
(5 rows)
```

6.4 MERGEAPPEND POUR UNION SANS ALL

- UNION sans ALL impose une déduplication des lignes (lourd)
- Passait uniquement par trois nœuds : Append, Sort puis Unique
- Peut maintenant utiliser Merge Append suivi d'Unique
- Permet d'éviter le tri en utilisant des index

Prenons l'exemple d'une table t1 avec un million de lignes et un index :

```
CREATE UNLOGGED TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
CREATE INDEX ON t1(id);
```

Et prenons la requête suivante :

```
SELECT id FROM t1
UNION
SELECT id FROM t1;
```

Comme nous utilions un UNION sans ALL, les lignes doivent être toutes distinctes. Pour cela, une solution simple, utilisée par toutes les versions précédentes de PostgreSQL, revenait à faire deux parcours séquentiels sur t1, de joindre leur résultat avec un nœud Append, puis de distinguer les lignes en les triant puis en identifiant les lignes uniques. Cela donne donc ce plan d'exécution (ici sur une version 16) :

```
Unique (actual rows=1000000 loops=1)
 Buffers: shared hit=8850, temp read=2941 written=2952
 I/O Timings: temp read=4.295 write=16.620
 ->  Sort (actual rows=2000000 loops=1)
       Sort Key: t1.id
       Sort Method: external merge  Disk: 23528kB
       Buffers: shared hit=8850, temp read=2941 written=2952
       I/O Timings: temp read=4.295 write=16.620
       ->  Append (actual rows=2000000 loops=1)
             Buffers: shared hit=8850
```

```
             ->  Seq Scan on t1 (actual rows=1000000 loops=1)
                   Buffers: shared hit=4425
             ->  Seq Scan on t1 t1_1 (actual rows=1000000 loops=1)
                   Buffers: shared hit=4425
Settings: jit = 'off', max_parallel_workers_per_gather = '0'
Planning:
 Buffers: shared hit=5
Planning Time: 0.094 ms
Execution Time: 533.984 ms
```

À partir de la version 17, un nœud MergeAppend peut être utilisé. Cela sera d'autant plus favorable s'il existe un index sur les colonnes concernées. Cela nous donne ce plan en version 17 :

```
Unique (actual rows=1000000 loops=1)
 Buffers: shared hit=5472
 ->  Merge Append (actual rows=2000000 loops=1)
       Sort Key: t1.id
       Buffers: shared hit=5472
     ->  Index Only Scan using t1_id_idx on t1 (actual rows=1000000 loops=1)
             Heap Fetches: 0
             Buffers: shared hit=2736
     ->  Index Only Scan using t1_id_idx on t1 t1_1 (actual rows=1000000 loops=1)
             Heap Fetches: 0
             Buffers: shared hit=2736
Settings: jit = 'off', max_parallel_workers_per_gather = '0'
Planning:
 Buffers: shared hit=5
Planning Time: 0.249 ms
Execution Time: 396.779 ms
```

Les parcours d'index permettent d'avoir les données pré-triées. `MergeAppend` permet de conserver ce tri pendant l'assemblage des résultats. Il ne reste plus qu'à distinguer les lignes uniques.

6.5 OPTIMISATIONS DES CTE

- Par extraction des statistiques des CTE
- Par une meilleure information des nœuds précédents sur l'ordre de tri des données de la CTE

Prenons comme exemple la table t1 comprenant un million d'enregistrements :

```
CREATE UNLOGGED TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
CREATE INDEX ON t1(id);
VACUUM ANALYZE t1;
CHECKPOINT;
```

Et prenons cette requête pour le premier exemple :

```
EXPLAIN (ANALYZE,SETTINGS,BUFFERS,COSTS off,TIMING off)
WITH b AS MATERIALIZED (
    SELECT id FROM t1
)
SELECT *
FROM t1
WHERE id IN (SELECT id FROM b);
```

Voici comment elle est exécutée en version 16 :

```
Nested Loop  (rows=500000 width=4) (actual rows=1000000 loops=1)
 Buffers: shared hit=3004426, temp read=4763 written=8732
 CTE b
  -> Seq Scan on t1 t1_1  (rows=1000000 width=4) (actual rows=1000000 loops=1)
        Buffers: shared hit=4425
 ->  HashAggregate  (rows=200 width=4) (actual rows=1000000 loops=1)
       Group Key: b.id
       Batches: 21  Memory Usage: 10305kB  Disk Usage: 23512kB
       Buffers: shared hit=4425, temp read=4763 written=8732
```

```
     ->  CTE Scan on b  (rows=1000000 width=4) (actual rows=1000000 loops=1)
           Buffers: shared hit=4425, temp written=1708
 ->  Index Only Scan using t1_id_idx on t1  (rows=1 width=4) (actual rows=1 loops=100
       Index Cond: (id = b.id)
       Heap Fetches: 0
       Buffers: shared hit=3000001
Planning:
 Buffers: shared hit=3
Planning Time: 0.293 ms
Execution Time: 2771.565 ms
```

La CTE ne fournit pas d'informations aux autres nœuds. Un nœud Hash Aggregate est choisi, car l'optimiseur estimait qu'il allait traiter 200 lignes, alors qu'en réalité, il en récupère un million. Cela a pour conséquence de nécessiter tant mémoire pour la table de hachage qu'il est nécessaire de la stocker sur disque, ce qui ralentit considérablement la requête.

Et maintenant, voici le plan d'exécution pour une version 17 :

```
Hash Semi Join  (rows=1000000 width=4) (actual rows=1000000 loops=1)
 Hash Cond: (t1.id = b.id)
 Buffers: shared hit=8850, temp read=5134 written=6842
 CTE b
  ->  Seq Scan on t1 t1_1  (rows=1000000 width=4) (actual rows=1000000 loops=1)
        Buffers: shared hit=4425
 ->  Seq Scan on t1  (rows=1000000 width=4) (actual rows=1000000 loops=1)
       Buffers: shared hit=4425
 ->  Hash  (rows=1000000 width=4) (actual rows=1000000 loops=1)
       Buckets: 262144  Batches: 8  Memory Usage: 6446kB
       Buffers: shared hit=4425, temp written=4268
     ->  CTE Scan on b  (rows=1000000 width=4) (actual rows=1000000 loops=1)
           Buffers: shared hit=4425, temp written=1708
Planning:
 Buffers: shared hit=157
Planning Time: 1.461 ms
Execution Time: 738.725 ms
```

Cette fois, la CTE peut donner suffisamment d'informations statistiques pour que l'optimiseur trouve un meilleur plan. Nous passons d'une durée de 2,7 secondes en version 16 à une durée de 0,7 secondes en version 17.

Comme on le voit dans le plan d'exécution, l'optimiseur a accès aux statistiques de certaines colonnes de la CTE matérialisée et peut les utiliser pour optimiser le plan. Ceci amène des estimations de cardinalité plus précises, et donc à un plan plus adéquat.

Testons maintenant une autre requête, comportant un tri :

```
EXPLAIN (ANALYZE,SETTINGS,BUFFERS,COSTS off)
WITH tmp AS MATERIALIZED (
    SELECT * FROM t1 ORDER BY id
)
SELECT * FROM tmp ORDER BY id;
```

Voici le plan en version 16 :

```
Sort  (rows=1000000 width=4) (actual time=359.858..428.068 rows=1000000 loops=1)
 Sort Key: tmp.id
 Sort Method: external merge  Disk: 11768kB
 Buffers: shared hit=2736, temp read=1471 written=3184
 CTE tmp
   ->  Index Only Scan using t1_id_idx on t1  (rows=1000000 width=4)
                    (actual time=0.034..88.719 rows=1000000 loops=1)
         Heap Fetches: 0
         Buffers: shared hit=2736
 ->  CTE Scan on tmp  (rows=1000000 width=4)
                     (actual time=0.037..282.122 rows=1000000 loops=1)
       Buffers: shared hit=2736, temp written=1708
Planning Time: 0.134 ms
Execution Time: 461.300 ms
```

La CTE n'a pas de moyen d'indiquer aux autres nœuds que ses données sont déjà triées par le parcours d'index. L'optimiseur croit nécessaire d'ajouter un nœud Sort, qui ralentit considérablement la requête.

Ce n'est pas le cas en version 17 :

```
 CTE Scan on tmp  (rows=1000000 width=4)
                  (actual time=0.062..281.560 rows=1000000 loops=1)
   Buffers: shared hit=2736, temp written=1708
   CTE tmp
```

```
    ->  Index Only Scan using t1_id_idx on t1  (rows=1000000 width=4)
                  (actual time=0.058..90.009 rows=1000000 loops=1)
          Heap Fetches: 0
          Buffers: shared hit=2736
 Planning Time: 0.106 ms
 Execution Time: 314.712 ms
```

Il n'y a plus d'opération `Sort` car l'optimiseur sait que les données sont déjà correctement triées. Il faut toutefois que l'ordre des deux `ORDER BY` soit compatible (pas de `DESC` sur l'un uniquement).

6.6 OPTIMISATIONS DES CLAUSES IN

- `WHERE c1 IN (... liste de valeurs...)`
- `WHERE c1 = ANY (... liste de valeurs...)`
- Syntaxes couramment utilisées mais peu efficaces
- Avant la v17, N parcours d'index
- En v17, 1 seul parcours d'index

Nous allons de nouveau prendre un exemple. Une table `t1` contient un million d'enregistrements. Un index se trouve sur la colonne `c1`.

```
DROP TABLE IF EXISTS t1;
CREATE UNLOGGED TABLE t1 (c1 integer);
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
CREATE INDEX ON t1(c1);
VACUUM ANALYZE t1;
```

Cet index n'a jamais été utilisé pour l'instant :

```
SELECT idx_scan FROM pg_stat_user_tables WHERE relname='t1';
 idx_scan
----------
        0
(1 row)
```

Maintenant, nous allons exécuter la requête suivante :

```
SELECT *
FROM t1
WHERE c1 IN (1,2,3,4,...,995,996,997,998,999,1000);
```

La clause `IN` contient donc 1000 valeurs.

Voici le plan d'exécution en version 16 :

```
Index Only Scan using t1_c1_idx on t1 (actual rows=1000 loops=1)
  Index Cond: (c1 = ANY ('{1,2,3,4,5,6,...}'))
  Heap Fetches: 0
  Buffers: shared hit=2999 read=5
  I/O Timings: shared read=0.050
Planning:
  Buffers: shared hit=16 read=1
  I/O Timings: shared read=0.222
Planning Time: 1.348 ms
Execution Time: 2.854 ms
```

La partie intéressante ici est le nombre de blocs lus : plus de 3000. C'est beaucoup pour un index.

Voyons maintenant le plan en version 17 :

```
Index Only Scan using t1_c1_idx on t1 (actual rows=1000 loops=1)
  Index Cond: (c1 = ANY ('{1,2,3,4,5,6,...}'))
  Heap Fetches: 0
  Buffers: shared hit=1 read=5
  I/O Timings: shared read=0.025
Planning:
  Buffers: shared hit=16 read=1
  I/O Timings: shared read=0.010
Planning Time: 0.392 ms
Execution Time: 0.203 ms
```

La structure du plan est identique mais le nombre de blocs lus est très inférieur : uniquement 6 (pour la partie exécution). Cela ramène la durée de 2,8 ms à seulement 0,2 ms.

La question est : pourquoi si peu de blocs ?

En version 16, l'exécuteur fait un parcours d'index par valeur à récupérer. Cela se voit bien dans les statistiques :

```
SELECT idx_scan FROM pg_stat_user_tables WHERE relname='t1';
 idx_scan
----------
     1000
(1 row)
```

En version 17, l'exécuteur est plus malin. Il fait un seul parcours pour trouver toutes les valeurs. Là aussi, cela se voit dans les statistiques :

```
SELECT idx_scan FROM pg_stat_user_tables WHERE relname='t1';
 idx_scan
----------
        1
(1 row)
```

Notez que cela fonctionne aussi avec la syntaxe :

```
SELECT *
FROM t1
WHERE c1 = ANY (1,2,3,4,...,995,996,997,998,999,1000);
```

6.7 MEILLEURE GESTION DES CONTRAINTES NOT NULL

- Utilisation des contraintes NOT NULL lors de l'optimisation des requêtes
- Suppression des tests IS NOT NULL pour les colonnes NOT NULL
- Suppression des parcours avec filtre IS NULL sur les colonnes NOT NULL

Prenons comme exemple la table `t1` comprenant un million d'enregistrements :

```
CREATE UNLOGGED TABLE t1 (id integer NOT NULL);
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
```

Et prenons cette requête pour le premier exemple :

```
SELECT * FROM t1 WHERE id IS NULL
```

Voici son plan d'exécution en version 17 :

```
Result (actual rows=0 loops=1)
  One-Time Filter: false
Planning Time: 0.048 ms
Execution Time: 0.020 ms
```

Il n'y a pas de parcours de table et de filtre car la contrainte nous garantit qu'il ne peut pas y avoir de ligne correspondant à ce filtre.

Testons maintenant avec un filtre inverse :

```
SELECT * FROM t1 WHERE id IS NOT NULL
```

Voici son plan en version 17 :

```
Seq Scan on t1 (actual rows=1000000 loops=1)
Planning Time: 0.173 ms
Execution Time: 59.104 ms
```

L'exécuteur ne fait pas de filtre (pas de ligne `Filter`), ce qui rend la requête plus rapide. Ce filtre n'est pas nécessaire grâce à la présence de la contrainte.

Pour comparaison, en version 16, la première requête met 64 ms à s'exécuter et la deuxième 104 ms. La version 17 est donc plus rapide (respectivement 0,02 ms et 59 ms).

7/ Fonctionnement interne

7.1 NOUVELLE FONCTIONS UNICODE

- `unicode_version()` : version d'Unicode de PostgreSQL
- `icu_unicode_version()` : version d'Unicode de ICU
- `unicode_assigned()` : tous les caractères d'une chaîne encodée en UTF8 sont-ils des points de code Unicode assignés ?

La version 17 de PostgreSQL ajoute trois fonctions relatives à Unicode :

```
                              List of functions
        Name          | Result data type | Argument data types | Type
----------------------+------------------+---------------------+------
 icu_unicode_version | text             |                     | func
 unicode_assigned    | boolean          | text                | func
 unicode_version     | text             |                     | func
(3 rows)
```

Ces fonctions sont nées d'une longue discussion sur la normalisation des caractères UTF8 qui permettrait de comparer des caractères Unicodes avec `memcpy`. Un autre point abordé concernait l'insertion de points de code Unicode qui ne sont pas encore reconnus et qui pourraient provoquer des incohérences dans les index quand ils le seront. Un point de code Unicode est une valeur occupant un à quatre octets, et représentant un symbole (caractère d'une langue ou une autre, emoji, etc…).

La discussion n'est pas close et pourrait évoluer vers la possibilité de créer des bases de données qui :

- n'acceptent *que* des points de code Unicode assignés ;
- ne stockent que des valeurs Unicodes normalisées.

7.2 AT LOCAL

- Convertit un `timestamptz` en un `timestamp` dans le fuseau horaire de la session

Dans les versions précédentes de PostgreSQL, il était possible de récupérer un *timestamp* converti dans un fuseau horaire spécifique avec la clause `AT TIME ZONE`.

```
SELECT '2023-10-23 6:30:00+02'::timestamptz AT TIME ZONE 'europe/london';
```

```
      timezone
---------------------
 2023-10-23 05:30:00
(1 row)
```

Il est désormais possible de faire la conversion dans le fuseau horaire défini dans la session avec la clause `AT LOCAL`.

```
SET TIME ZONE "europe/paris";
SELECT '2023-10-23 6:30:00+02'::timestamptz AT LOCAL;
```

```
      timezone
---------------------
 2023-10-23 06:30:00
(1 row)
```

```
SET TIME ZONE "europe/helsinki";
SELECT '2023-10-23 6:30:00+02'::timestamptz AT LOCAL;
```

```
      timezone
---------------------
 2023-10-23 07:30:00
(1 row)
```

7.3 FONCTION PG_COLUMN_TOAST_CHUNK_ID()

- Récupérer le numéro de *chunk* d'une colonne toastée dans la table TOAST

Pour des détails sur le fonctionnement des tables TOAST, voir https://dali.bo/m4_html#m%C3%A9canisme-toast

Avant l'implémentation de la fonction `pg_column_toast_chunk_id()`, il n'était pas possible de récupérer le numéro de *chunk* correspondant à une entrée dans la relation toast associée à une table sans utiliser une extension comme `toastinfo`[1].

Exemple :

```
CREATE TABLE ttoast(i int GENERATED ALWAYS AS IDENTITY, t text);
CREATE TABLE notoast(i int GENERATED ALWAYS AS IDENTITY);
-- récupération de la table toast associée à la table ttoast
SELECT relname, reltoastrelid::regclass
  FROM pg_class
 WHERE relname IN ('ttoast','notoast');
```
```console
 relname |      reltoastrelid
---------+-------------------------
 ttoast  | pg_toast.pg_toast_16449
 notoast | -
(1 row)
```

On peut voir que dès lors que la table créée contient des données *toastables* (format de colonne EXTERNAL ou EXTENDED, ici la colonne `text`), la table TOAST associée est créée automatiquement.

Insérons maintenant des données dans la table `ttoast` :

```
INSERT INTO ttoast(t) VALUES (repeat('thestring', 10000));   -- non toastée
INSERT INTO ttoast(t) VALUES (repeat('thestring', 1000000)); -- toastée
```

La fonction `pg_column_toast_chunk_id()`, nous permet de savoir si une colonne d'une ligne est effectivement toastée et nous donne le numéro de *chunk* correspondant à la ligne dans la table TOAST.

```
SELECT i, pg_column_toast_chunk_id(t) FROM ttoast;
```
```
 i | pg_column_toast_chunk_id
---+--------------------------
```

[1] https://github.com/credativ/toastinfo

```
 1 |                          ¤
 2 |                      16458
(2 rows)
```

En regardant dans la table TOAST, on voit que la colonne est découpée en 52 morceaux :

```
SELECT chunk_id, count(*)
  FROM pg_toast.pg_toast_16449
 WHERE chunk_id = 16458
GROUP BY 1;
```

```
 chunk_id | count
----------+-------
    16458 |    52
(1 row)
```

8/ Régressions

- Paramètres supprimés
  - `old_snapshot_threshold`
  - `trace_recovery_messages`
- Extension supprimée
  - adminpack
- Fonctionnalité supprimée
  - utilisateur par base (`db_user_namespace`)

Le paramètre `old_snapshot_threshold`, ajouté à l'origine pour contrôler la durée de vie des snapshots permettait de limiter la durée de vie des transactions trop longues. Il est supprimé suite à de nombreux problèmes connus en termes de précision et de performance, car longtemps resté sans recherche active de corrections. Cette fonctionnalité reste cependant intéressante, et il n'est pas exclus qu'une nouvelle impémentation puisse être réintégrée.

Le paramètre `trace_recovery_messages` permettait d'obtenir les logs de l'état de la réplication lors des phases de RECOVERY. Ce paramètre est considéré comme obsolète, car il existe des outils pouvant fournir les mêmes informations, comme par exemple `pg_waldump`.

L'extension `adminpack` n'existait que pour le supoprt de pgAdmin 3, dont la fin de vie a été déclarée il y a plusieurs années. La suppression de l'extension permet également de supprimer les fonctions qui lui sont liées.

La fonctionnalité d'utilisateur par base, liée au paramètre `db_user_namespace`, permettait d'avoir des utilisateurs sur une base définie plutôt que sur l'instance complète. Implémentation à la base temporaire n'ayant pas reçu d'amélioration, ni de déclaration d'utilisation sur ses 21 années de service, elle a de ce fait été supprimée.

9/ Peut-être pour la prochaine fois ?

Ce qu'on ne trouvera finalement pas dans cette version :

- `SPLIT/MERGE PARTITIONS` : faille de sécurité
- clés primaires temporelles
- optimisation des auto-jointures

Attention aux articles sur le web !

La qualité et la sécurité passent avant la course aux fonctionnalités. Il arrive donc que certaines, intégrées pendant la phase de développement, soient finalement repoussées à une version ultérieure après découverte d'un problème majeur sans correction immédiate possible. On peut espérer que PostgreSQL 18 les intégrera.

Notamment, la syntaxe pour fusionner et scinder des partitions semblait prête, mais une faille de sécurité découverte tardivement n'a pu être corrigée proprement à temps.

Les clés primaires et étrangères temporelles, avec la syntaxe `PERIOD`, ont également été supprimées suite à des bugs.

Une optimisation améliorant les jointures d'une table avec elle-même a provoqué trop de bugs et a été supprimée.

On peut citer aussi l'annulation de tests, de fonctions utilitaires, ou de restructuration ou optimisation dans le code. Parfois les bugs ne se révèlent que sur certaines architectures, ou entraînent une incompatibilité.

Méfiez-vous de certains articles sur le web décrivant les nouveautés de PostgreSQL 17, parus trop tôt, et non corrigés.

10/ Questions

Merci de votre écoute !
Nouveautés de la version 17 :
https://dali.bo/workshop17_html
https://dali.bo/workshop17_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
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- 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.