Workshop 16

Nouveautés de PostgreSQL 16

16

Contents

0.1 Introduction 1

1/ Utilisation **3**

1.1 Prédicats IS JSON 4
1.2 Omission possible de l'alias d'une sous-requête 7
1.3 Gestion de triggers TRUNCATE sur des tables externes 9
1.4 Ajout de fonctions de vérification de types 12
1.5 Possibilité d'utiliser des tirets bas pour des entiers ou valeurs numériques 14

2/ Administration **15**

2.1 Ajout de la variable SYSTEM_USER 16
2.2 `archive_library` et `archive_command` ne peuvent plus être renseignés en même temps 19
2.3 Réservation de slots de connexion 20
2.4 Ajout du paramètre `scram_iterations` 23
2.5 Ajout de la possibilité d'inclure d'autres fichiers ou dossier dans pg_hba.conf et pg_ident.conf 24
2.6 Ajout du support des expressions régulières dans le fichier pg_hba.conf 26
2.7 Ajout de la gestion des tables enfants et partitionnées dans pg_dump 28
2.8 lz4 et zstd peuvent être utilisés avec pg_dump 32
2.9 Contrôle de l'utilisation de la mémoire partagée par ANALYZE et VACUUM 34
2.10 Ajout des options `--schema` et `--exclude-schema` dans vacuumdb 37
2.11 Ajout des options SKIP_DATABASE_STATS et ONLY_DATABASE_STATS 38
2.12 Optimisation de ANALYZE avec postgres_fdw 40
2.13 Refonte du système de délégation de droits 42
2.14 Nouveau paramètre libpq : require_auth 46
2.15 Sélection aléatoire des hosts par libpq 48

3/ Réplication **53**

3.1 Décodage logique sur les instances secondaires 54
3.1.1 Modification de la structure des WAL 54
3.1.2 Mettre en place une gestion des conflits sur les standby 55
3.1.3 Création d'un slot de réplication sur une instance secondaire 55
3.1.4 Décodage logique sur les instances secondaires 56
3.1.5 Publications sur une instance secondaire 57
3.1.6 Conflits de réplication 60

3.1.7 Bascules et décodage logique . . . 61
3.2 Parallélisme de l'application des modifications . . . 64
3.3 Nouveau role pg_create_subscription . . . 65

4/ Performances **69**
4.1 Nouvelle option d'EXPLAIN . . . 70
4.2 Plus d'utilisation du Incremental Sort . . . 72
4.3 Amélioration des agrégats . . . 74
4.4 Parallélisation des agrégats string_agg et array_agg . . . 77
4.5 Parallélisation des FULL OUTER JOIN . . . 78

5/ Supervision **81**
5.1 Nouvelle vue pg_stat_io . . . 82
5.2 Horodatage du dernier parcours d'une relation . . . 84
5.3 Nombre d'UPDATE . . . 86
5.4 Amélioration de pg_stat_statements . . . 89
5.5 Amélioration de auto_explain . . . 90

6/ Régression **91**
6.1 Disparition des variables LC_COLLATE et LC_CTYPE . . . 92

7/ Autres régressions **93**
7.1 Questions ? . . . 94

Notes **95**

Notes **97**

Notes **99**

Nos autres publications **101**
Formations . . . 102
Livres blancs . . . 103
Téléchargement gratuit . . . 104

8/ DALIBO, L'Expertise PostgreSQL **105**

0.1 INTRODUCTION

- Développement depuis l'été 2022
- 3 versions beta, 1 version RC
- Version finale : 14 septembre 2023
- 16.1, le 9 novembre 2023
- Des centaines de contributeurs

Le développement de la version 16 a suivi l'organisation habituelle : un démarrage vers la mi-2022, des *Commit Fests* tous les deux mois, un *feature freeze*, trois versions beta, une version RC, et enfin la GA.

La version finale est parue le 14 septembre 2023. Une première version corrective est sortie le 16 novembre 2023.

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

1/ Utilisation

1.1 PRÉDICATS IS JSON

- Support du prédicat `IS JSON`
- `IS NOT JSON`
- option `WITH UNIQUE KEYS`

Le prédicat `IS JSON` est désormais implémenté dans la version 16 de PostgreSQL. Il peut être appliqué sur des champs `text` ou `bytea` et évidemment sur des champs `json` et `jsonb`.

Il existe quatre nouveaux predicats :

- IS JSON [VALUE]
- IS JSON ARRAY
- IS JSON OBJECT
- IS JSON SCALAR

Chacun d'eux possède également sa variante `IS NOT` qui renvoie `true` lorsque la valeur testée ne respecte pas le standard `JSON`.

L'option `WITH UNIQUE KEYS` permet de renvoyer `false` si il existe des clés en doublon dans la valeur testée.

Créons un petit jeu de test pour manipuler ces nouveautés :

```
CREATE TABLE doc (id SERIAL PRIMARY KEY, content text not NULL);

INSERT INTO doc (content) VALUES ('{"auteur": "Melanie", "titre": "Mon livre",
↪  "prix": "25", "date": "01-05-2023"}');

INSERT INTO doc (content) VALUES ('{"auteur": "Thomas", "auteur": "Thomas", "titre":
↪  "Le livre de Thomas","prix": "8", "date": "07-08-2022"}');

INSERT INTO doc (content) VALUES ('{"auteur": "Melanie", "titre": "Mon second
↪  livre", "prix": "30", "date": "10-08-2023}');
```

Regardons ce que nous renvoie `IS JSON` :

```
postgres=# select id, content IS JSON as valid from doc;
 id | valid
----+-------
  1 | t
  2 | t
```

```
  3 | f
(3 rows)
```

La dernière ligne ne semble pas être du `JSON` … en effet, il manque un " après la date.

Regardons maintenant ce que retourne la même commande avec l'option `WITH UNIQUE KEYS`.

```
select id, content IS JSON WITH UNIQUE KEYS as valid from doc;
 id | valid
----+-------
  1 | t
  2 | f
  3 | f
(3 rows)
```

Nous retrouvons bien la dernière ligne qui n'est pas du `JSON` mais également la deuxième qui ne respecte pas la particularité d'avoir des clés uniques. En effet, la clé `auteur` a été ajoutée deux fois.

Les autres prédicats servent à valider le contenu d'un `JSON`. Quelques exemples très simples :

- IS JSON ARRAY

```
SELECT '{"noms": [{"interne": "production", "externe": "prod"}],
↪  "version":"1.1"}'::json ->> 'noms' IS JSON ARRAY as valid;
 valid
-------
 t
(1 row)
SELECT '{"nom": "production", "version":"1.1"}'::json ->> 'nom' IS JSON ARRAY as
↪  valid;
 valid
-------
 f
(1 row)
```

- IS JSON OBJECT

```
SELECT '{"nom": "production", "version":"1.1"}'::json IS JSON OBJECT as valid;
 valid
-------
 t
(1 row)

SELECT '{"nom": "production", "version":"1.1"}'::json ->> 'nom' IS JSON OBJECT as
↪  valid;
 valid
-------
 f
(1 row)
```

- IS JSON SCALAR

```
SELECT '{"nom": "production", "version":"1.1"}'::json ->> 'version' IS JSON SCALAR
↪  as valid;
 valid
-------
 t
(1 row)

SELECT '{"nom": "production", "version":"RC1"}'::json ->> 'version' IS JSON SCALAR
↪  as valid;
 valid
-------
 f
(1 row)
```

1.2 OMISSION POSSIBLE DE L'ALIAS D'UNE SOUS-REQUÊTE

- Auparavant, l'alias était obligatoire

```
SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_datatase WHERE NOT datistemplate) tmp;
```

- Maintenant, c'est optionnel

```
SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_datatase WHERE NOT datistemplate);
```

- Améliore la lisibilité

Les versions antérieures à la 16 étaient très rigides sur ce point :

```
# En version 15, sans alias
postgres=# SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_database WHERE NOT datistemplate);

ERROR:  subquery in FROM must have an alias
LINE 2: FROM (SELECT * from pg_database WHERE NOT datistemplate);
             ^
HINT:  For example, FROM (SELECT ...) [AS] foo.

# En version 15, avec alias
postgres=# SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_database WHERE NOT datistemplate) tmp;

 datname  | pg_database_size
----------+------------------
 postgres |          7869231
(1 row)
```

En version 16, les deux écritures sont acceptées :

```
# En version 16, sans alias
postgres=# SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_database WHERE NOT datistemplate);
 datname  | pg_database_size
----------+------------------
 postgres |          7909859
```

```
(1 row)

# En version 16, avec alias
postgres=# SELECT datname, pg_database_size(datname)
FROM (SELECT * from pg_database WHERE NOT datistemplate) tmp;
 datname  | pg_database_size
----------+------------------
 postgres |          7909859
(1 row)
```

1.3 GESTION DE TRIGGERS TRUNCATE SUR DES TABLES EXTERNES

- TRUNCATE sur table externe possible
- Mais pas de trigger sur TRUNCATE pour ce type de table
- Même gestion que pour une table normale
- Intérêts
  - audit des opérations sur une table externe
  - interdiction de cette opération

La version 16 permet d'ajouter un trigger sur TRUNCATE pour des tables externes.

Voici un exemple complet sous la forme d'un script SQL :

```
DROP DATABASE IF EXISTS b1;
DROP DATABASE IF EXISTS b2;
CREATE DATABASE b1;
CREATE DATABASE b2;
\c b2
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1
  SELECT i, 'Ligne '||i FROM generate_series(1,5) i;
\c b1
CREATE EXTENSION postgres_fdw;
CREATE SERVER remote_b2
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (dbname 'b2');
CREATE USER MAPPING FOR CURRENT_ROLE SERVER remote_b2;
CREATE FOREIGN TABLE public.remote_t1 (c1 integer, c2 text)
  SERVER remote_b2
  OPTIONS (table_name 't1');

TABLE remote_t1;
TRUNCATE remote_t1;
TABLE remote_t1;

INSERT INTO remote_t1 SELECT i, 'Ligne '||i FROM generate_series(1,5) i;

CREATE FUNCTION deny_truncate_function() RETURNS trigger LANGUAGE plpgsql AS
$$
begin
```

```
  raise exception 'You shall not truncate foreign tables!';
  return null;
end
$$;

CREATE TRIGGER deny_truncate_trigger
  BEFORE TRUNCATE ON remote_t1
  FOR EACH STATEMENT
  EXECUTE FUNCTION deny_truncate_function();

TABLE remote_t1;
TRUNCATE remote_t1;
TABLE remote_t1;
```

Et voici le résultat suite à l'exécution de ce script avec *psql* :

```
DROP DATABASE
DROP DATABASE
CREATE DATABASE
CREATE DATABASE
You are now connected to database "b2" as user "postgres".
CREATE TABLE
INSERT 0 5
You are now connected to database "b1" as user "postgres".
CREATE EXTENSION
CREATE SERVER
CREATE USER MAPPING
CREATE FOREIGN TABLE
 c1 |   c2
----+---------
  1 | Ligne 1
  2 | Ligne 2
  3 | Ligne 3
  4 | Ligne 4
  5 | Ligne 5
(5 rows)

TRUNCATE TABLE
 c1 | c2
----+----
(0 rows)

INSERT 0 5
CREATE FUNCTION
CREATE TRIGGER
 c1 |   c2
----+---------
```

```
  1 | Ligne 1
  2 | Ligne 2
  3 | Ligne 3
  4 | Ligne 4
  5 | Ligne 5
(5 rows)

psql:script.sql:39: ERROR:  You shall not truncate foreign tables!
CONTEXT:  PL/pgSQL function deny_truncate_function() line 3 at RAISE
 c1 |   c2
----+---------
  1 | Ligne 1
  2 | Ligne 2
  3 | Ligne 3
  4 | Ligne 4
  5 | Ligne 5
(5 rows)
```

1.4 AJOUT DE FONCTIONS DE VÉRIFICATION DE TYPES

- Deux nouvelles fonctions
  - `pg_input_is_valid()`
  - `pg_input_error_info()`

Deux nouvelles fonctions sont disponibles et permettent de vérifier qu'une valeur est conforme à un type de données.

La fonction `pg_input_is_valid()` renvoie `true` / `false` selon si la valeur et le type coïncident.

La fonction `pg_input_error_info()` quant à elle renvoie plusieurs informations (message, detail, hint, sql_error_code) si les deux ne coïncident pas, NULL dans le cas contraire.

```
# valide
postgres=# select pg_input_is_valid('2005', 'integer');
 pg_input_is_valid
-------------------
 t
(1 row)

# invalide
postgres=# select pg_input_is_valid('dalibo', 'integer');
 pg_input_is_valid
-------------------
 f
(1 row)

# valide
postgres=# select * from  pg_input_error_info('2005', 'integer');
 message | detail | hint | sql_error_code
---------+--------+------+----------------
         |        |      |
(1 row)

# invalide
postgres=# select * from  pg_input_error_info('dalibo', 'integer');
                    message                        | detail | hint | sql_error_code
---------------------------------------------------+--------+------+----------------
```

```
 invalid input syntax for type integer: "dalibo" |         |       | 22P02
(1 row)
```

1.5 POSSIBILITÉ D'UTILISER DES TIRETS BAS POUR DES ENTIERS OU VALEURS NUMÉRIQUES

- Utilisation autorisée des _ dans les nombres
- Améliore la lisibilité
- `1000000 = 1_000_000`

Cette amélioration permet d'utiliser le tiret bas (ou *underscore*) lors de l'utilisation d'entiers ou de numériques. La lisibilité est alors grandement améliorée.

Par exemple, une insertion de `9_999_999` ne relève plus d'erreur avec PostgreSQL 16.

```
# En version 15
postgres=# insert into t1 values (9_999_999);
ERROR:  trailing junk after numeric literal at or near "9_"
LINE 1: insert into t1 values (9_999_999);

# En version 16
postgres=# insert into t1 values (9_999_999);
INSERT 0 1
```

Un autre exemple de ce qu'il est possible de faire avec des entiers dans deux bases différentes (base 10 et base 2) :

```
postgres=# select 1_000_000_000 + 0b_1000_0000 as result;
   result
------------
 1000000128
(1 row)
```

2/ Administration

2.1 AJOUT DE LA VARIABLE SYSTEM_USER

- Nouvelle fonction SYSTEM_USER du standard SQL
- Affiche l'utilisateur système utilisé et la méthode de connexion
- `auth_method:identity`
- Valeur NULL si la méthode `trust` est utilisée

La fonction SYSTEM_USER du standard SQL est désormais implémentée avec PostgreSQL 16. Les informations remontées par cette fonction permettent de connaitre l'utilisateur système et la manière dont il s'est connecté.

Si la méthode d'authentification `trust` est utilisée, cette fonction retourne NULL.

Dans l'exemple suivant, un utilisateur système `sysadmin` peut se connecter à l'instance en tant que `dalibo` grâce au fichier `pg_ident.conf` et à la configuration de `pg_hba.conf`.

Fichier `pg_ident.conf` :

```
# MAPNAME        SYSTEM-USERNAME         PG-USERNAME
sysdb            sysadmin                dalibo
```

Fichier `pg_hba.conf` :

```
[...]
local   all              all                             ident map=sysdb
[...]
```

La connexion s'effectue correctement et la variable `system_user` a bien pour valeur `sysadmin`.

```
$ whoami
sysadmin
$ psql -U dalibo -d postgres
Password for user dalibo:
psql (16.1)
Type "help" for help.

postgres=> select current_user, session_user, system_user;
 current_user | session_user | system_user
--------------+--------------+-------------
 dalibo       | dalibo       | peer:sysadmin
(1 row)
```

Regardons ce qu'il se passe lorsque la commande `SET ROLE` est utilisée. Celle-ci permet d'endosser un autre rôle (changement de `current_user`), par exemple, pour l'exécution d'une commande spécifique. Elle ne change en rien l'utilisateur de session ou du système.

```
$ whoami
sysadmin
$ psql -U dalibo -d postgres
psql (16.1)
Type "help" for help.

postgres=> set role admin;
SET
postgres=> select current_user, session_user, system_user;
 current_user | session_user | system_user
--------------+--------------+-------------
 admin        | dalibo       | peer:sysadmin
(1 row)
```

Comme expliqué au début, la valeur de `system_user` sera `NULL` lorsque la méthode `trust` est utilisée.

```
$ psql -U postgres
psql (16.1)
Type "help" for help.

postgres=# select current_user, session_user, system_user;
 current_user | session_user | system_user
--------------+--------------+-------------
 postgres     | postgres     |
(1 row)
```

Voici un autre exemple avec l'utilisation de la commande `SET ROLE`. Celle-ci permet d'endosser un autre rôle (changement de `current_user`), par exemple, pour l'exécution d'une commande spécifique. Elle ne change en rien l'utilisateur de session ou du système.

```
$ psql -U dalibo -d postgres
psql (16.1)
Type "help" for help.

postgres=> set role admin;
SET
postgres=> select current_user, session_user, system_user;
 current_user | session_user | system_user
--------------+--------------+-------------
 admin        | dalibo       | md5:dalibo
(1 row)
```

2.2 ARCHIVE_LIBRARY ET ARCHIVE_COMMAND NE PEUVENT PLUS ÊTRE RENSEIGNÉS EN MÊME TEMPS

- `archive_library` et `archive_command` ne peuvent pas être configurés en même temps
- Une erreur FATAL est renvoyée
- Avant `archive_library` prenait le dessus

Il n'est désormais plus possible de définir les paramètres `archive_library` et `archive_command` en même temps. Si c'est le cas, une erreur est remontée dans les traces. Par exemple :

```
2023-08-22 16:49:12.620 CEST [2082970] LOG:  database system was shut down at
 ↪  2023-08-22 16:49:12 CEST
2023-08-22 16:49:12.631 CEST [2082967] LOG:  database system is ready to accept
 ↪  connections
2023-08-22 16:49:12.631 CEST [2082973] FATAL:  both archive_command and
 ↪  archive_library set
2023-08-22 16:49:12.631 CEST [2082973] DETAIL:  Only one of archive_command,
 ↪  archive_library may be set.
```

L'archivage des fichiers ne se fera pas tant que les deux paramètres seront présents. Le fichier de transaction sera marqué comme `ready` et sera archivé lors d'un rechargement de la configuration une fois corrigée.

2.3 RÉSERVATION DE SLOTS DE CONNEXION

- Nouveau rôle pg_use_reserved_connections
  - permet d'utiliser des slots de connexions réservés
- Nouveau paramètre reserved_connections
  - pour configurer le nombre de slots réservés

Un nouveau rôle prédéfini a été ajouté dans cette version de PostgreSQL.

Les rôles, pour lesquels le rôle prédéfini pg_use_reserved_connections a été attribué, peuvent utiliser les connexions réservées par le paramètre de configuration reserved_connections.

Prenons un exemple très simpliste avec la configuration suivante. Un seul utilisateur normal peut se connecter à l'instance (6-3-2 = 1). Les cinq autres connexions étant réservées soit pour des utilisateurs privilégiés (reserved_connections), soit pour des administrateurs (superuser_reserved_connections).

```
postgres=# show max_connections ;
 max_connections
-----------------
 6
(1 row)

postgres=# show reserved_connections ;
 reserved_connections
----------------------
 2
(1 row)

postgres=# show superuser_reserved_connections ;
 superuser_reserved_connections
--------------------------------
 3
(1 row)
```

La création des rôles s'est faite de la manière suivante :

```
postgres=# create role r1 with login password 'role1';
CREATE ROLE
```

```
postgres=# create role a1 with login password 'admin1';
CREATE ROLE
```

Le nouveau rôle prédéfini a été attribué avec la commande GRANT.

```
postgres=# grant pg_use_reserved_connections to a1;
GRANT ROLE
```

Essayons désormais de nous connecter une fois avec l'utilisateur `r1`. Tout se passe bien.

```
$ psql -U r1 -d postgres
psql (16.1)
Type "help" for help.

postgres=>
```

Essayons une nouvelle fois avec ce même utilisateur … Il n'est pas possible de se connecter car nous avons atteint la limite de connexion possible pour des utilisateurs normaux.

```
$ psql -U r1 -d postgres
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
remaining connection slots are reserved for roles with privileges of the
"pg_use_reserved_connections" role
```

Cependant, il nous est possible de nous connecter avec l'utilisateur `a1`, qui lui en tant que membre de `pg_use_reserved_connections`, dispose des slots de connexions réservés de `pg_use_reserved_connections`.

```
$ psql -U a1 -d postgres
psql (16.1)
Type "help" for help.

postgres=>
```

Si la limite de `reserved_connections` est atteinte, le message d'erreur suivant sera affiché lors d'une connexion avec un rôle membre de `pg_use_reserved_connections`.

```
$ psql -U a1 -d postgres
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
↪  remaining connection slots are reserved for roles with the SUPERUSER attribute
```

Même si cette limite est atteinte, il restera encore la possibilité de se connecter avec un superutilisateur.

```
$ psql -U postgres -d postgres
psql (16.1)
Type "help" for help.

postgres=#
```

2.4 AJOUT DU PARAMÈTRE SCRAM_ITERATIONS

- Nouveau paramètre `scram_iterations`
  - détermine le nombre d'itérations à effectuer lors du chiffrement d'un mot de passe avec SCRAM
- Valeur par défaut
  - 4096

Il est désormais possible de configurer le nombre d'itérations effectuées par l'algorithme de hachage lors de l'utilisation du mécanisme d'authentification SCRAM. La valeur par défaut de 4096 itérations était écrite en dur dans le code, suivant ainsi la recommandation de la RFC 7677[1].

Augmenter ce paramètre permet d'obtenir des mots de passe plus résistants aux attaques par force brute, étant donné que le coût de calcul est plus important lors de la connexion. Si ce paramètre est réduit, le coût de calcul est logiquement réduit.

Ce paramètre nécessite uniquement un rechargement de la configuration de l'instance si il est modifié.

---

[1] https://datatracker.ietf.org/doc/html/rfc7677

2.5 AJOUT DE LA POSSIBILITÉ D'INCLURE D'AUTRES FICHIERS OU DOSSIER DANS PG_HBA.CONF ET PG_IDENT.CONF

- Trois nouveaux mots clés dans `pg_hba.conf` et `pg_ident.conf`
  - `include` : un fichier
  - `include_if_exists`: un fichier s'il existe, l'ignorer autrement
  - `include_dir`: un dossier
- Champs `file_name` dans `pg_hba_file_rules` et `pg_ident_file_mappings`
  - permet de savoir d'où est tirée la configuration

Les fichiers `pg_hba.conf` et `pg_ident.conf` supportent désormais l'utilisation des mots clés `include`, `include_if_exists` et `include_dir` afin d'inclure des fichiers de configuration supplémentaires. Si un fichier contient un espace, il doit être entouré de guillemets doubles. Les chemins des fichiers ou dossiers peuvent être relatifs ou absolus.

De plus, les vues `pg_hba_file_rules` et `pg_ident_file_mappings` voient un champ supplémentaire leur être attribués : `file_name`. Il permet de savoir d'où est tirée la configuration.

Voici un exemple avec le fichier `pg_hba.conf` qui inclue le fichier `auth_dba.conf`. Ce dernier contient les autorisations d'accès pour une certaine adresse IP uniquement :

```
include auth_dba.conf

# TYPE  DATABASE        USER            ADDRESS                 METHOD
# Base de production
host    production      dba            192.168.1.165/32         scram-sha-256

postgres=# select * from pg_hba_file_rules ;
[...]

-[ RECORD 7 ]-----------------------------------------------
rule_number | 7
file_name   | /etc/postgresql/16/main/auth_dba.conf
line_number | 3
type        | host
database    | {production}
```

```
user_name   | {dba}
address     | 192.168.1.165
netmask     | 255.255.255.255
auth_method | scram-sha-256
options     |
error       |
```

2.6 AJOUT DU SUPPORT DES EXPRESSIONS RÉGULIÈRES DANS LE FICHIER PG_HBA.CONF

- Préfixe /
  - rupture avec les versions inférieures
- Champs concernés : base et utilisateur

PostgreSQL supporte désormais l'utilisation d'expressions régulières dans les fichiers `pg_hba.conf` et `pg_ident.conf`. Elles peuvent être utilisées pour les champs correspondant aux bases de données et aux utilisateurs.

Il est nécessaire d'utiliser le caractère `/` en début de mot pour que PostgreSQL l'évalue comme une expression régulière. Si une virgule existe dans le mot, il doit être encadré avec des guillemets doubles pour être pris en compte. Il n'est pas possible d'utiliser une expression régulière pour les noms d'hôtes.

Ce changement est en rupture avec les anciennes versions de PostgreSQL et la manière de comprendre les paramètres de ces fichiers. Avant, le caractère `/` était compris comme un caractère normal pouvant faire partie du nom d'utilisateur ou de la base de données.

Des fichiers `pg_hba.conf` et `pg_ident.conf` écrits avec des expressions régulières pour une version 16, ne seront pas supportés par une version inférieure à 16.

Lors de l'authentification, l'utilisateur ainsi que la base de données sont vérifiés dans l'ordre suivant :

1. d'abord les mots clés qui n'auront jamais d'expressions régulières (comme `all` ou `replication`) ;
2. puis les expressions régulières ;
3. et enfin la correspondance exacte.

Voici un exemple simple où nous mettons à disposition des bases de données mutualisées sur une même instance. Nous avons un compte administrateur pour les bases de test (`admin_t`) et un pour celle de pré-production (`admin_p`). L'utilisation d'expressions particulières est très intéressante.

```
# Fichier pg_hba.conf
# TYPE  DATABASE              USER         ADDRESS                METHOD
host    /client[1-5]_test      admin_t      127.0.0.1/32          scram-sha-256
```

```
host    /client[1-5]_preprod    admin_p        127.0.0.1/32            scram-sha-256

# Accès à une base de test avec admin_t

$ psql -U admin_t -d client1_test -h 127.0.0.1
Password for user admin_t:
psql (16.1)
Type "help" for help.

client1_test=>

# Accès à une base de pré-production avec admin_t
$ psql -U admin_t -d client1_preprod -h 127.0.0.1
psql: error: connection to server at "127.0.0.1", port 5432 failed: FATAL:  no
↪  pg_hba.conf entry for host "127.0.0.1", user "admin_t", database
↪  "client1_preprod", no encryption

# Accès à une base de pré-production avec admin_p
psql -U admin_p -d client5_preprod -h 127.0.0.1
Password for user admin_p:
psql (16.1)
Type "help" for help.

client5_preprod=>
```

2.7 AJOUT DE LA GESTION DES TABLES ENFANTS ET PARTITIONNÉES DANS PG_DUMP

- Trois nouvelles options sont disponibles pour pg_dump
  - `--table-and-children`
  - `--exclude-table-and-children`
  - `--exclude-table-data-and-children`
- Inclusion ou exclusion de partitions lors d'une sauvegarde d'une table partitionnée

L'outil `pg_dump` permet désormais d'inclure ou d'exclure les tables enfants et les partitions de la sauvegarde logique. Pour cela, trois options ont été ajoutées :

- `--table-and-children` : permet de sauvegarder seulement les tables dont le nom correspond au motif ainsi que leurs tables enfants ou partitions qui existeraient.
- `--exclude-table-and-children` : permet d'exclure les tables dont le nom correspond au motif ainsi que leurs tables enfants ou partitions qui existeraient.
- `--exclude-table-data-and-children` : permet d'exclure les données des tables dont le nom correspond au motif ainsi que celles de leurs tables enfants ou partitions qui existeraient.

Les options `--exclude-table-data-and-children` et `--exclude-table-and-children` peuvent être appelées plusieurs fois dans la commande.

Imaginons une base de données `cave` d'un professionnel avec toutes ses références de bouteilles de vin, de cavistes, de récoltants. Imaginons ensuite la table `stock` qui contient les bouteilles disponibles. La table `stock` est partitionnée selon l'année de la bouteille. Particularité pour les bouteilles de 2001, elles sont également triées selon leur cru (1, 2 ou 3).

Dans un premier temps, voyons ce que l'option `--table-and-children` permet de faire. Pour sauvegarder le stock tout entier, il est possible d'utiliser cette nouvelle option. `pg_restore --list` nous confirme bien que les tables enfants ont été prises en compte.

```
# Sauvegarde
$ pg_dump -d cave -U postgres -Fc --table-and-children=stock* > stock.pgdump

# Inspection
$ pg_restore --list stock.pgdump
```

```
;
; Archive created at 2023-11-10 09:47:10 CET
;     dbname: cave
;     TOC Entries: 32
[...]

# Les définitions des tables sont bien sauvegardées ...
228; 1259 41285 TABLE public stock postgres
233; 1259 41311 TABLE public stock_2001 postgres
234; 1259 41314 TABLE public stock_2001_1 postgres
236; 1259 41320 TABLE public stock_2001_2 postgres
235; 1259 41317 TABLE public stock_2001_3 postgres
229; 1259 41291 TABLE public stock_2002 postgres
230; 1259 41294 TABLE public stock_2003 postgres
231; 1259 41297 TABLE public stock_2004 postgres
232; 1259 41300 TABLE public stock_2005 postgres

# ... ainsi que leurs données
3440; 0 41314 TABLE DATA public stock_2001_1 postgres
3442; 0 41320 TABLE DATA public stock_2001_2 postgres
3441; 0 41317 TABLE DATA public stock_2001_3 postgres
3436; 0 41291 TABLE DATA public stock_2002 postgres
3437; 0 41294 TABLE DATA public stock_2003 postgres
3438; 0 41297 TABLE DATA public stock_2004 postgres
3439; 0 41300 TABLE DATA public stock_2005 postgres
[...]
```

Prenons le cas maintenant d'un très bon acheteur qui demanderait un export de toutes les bouteilles du stock sauf celle de l'année 2001. Il souhaite intégrer ces données dans sa propre base.

L'option `--exclude-table-and-children` de pg_dump peut être utilisée pour satisfaire sa demande. Cette option permet d'exclure les données de la table stock_2001 ainsi que ses partitions. L'option `-T` de pg_dump n'aurait pas permis cela.

```
# Sauvegarde
$ pg_dump -d cave -U postgres -Fc --table-and-children=stock*
↪   --exclude-table-and-children=stock_2001 > stock_pour_client.pgdump

# Inspection
$ pg_restore --list stock_meilleures_annees.pgdump

[...]
228; 1259 41285 TABLE public stock postgres
229; 1259 41291 TABLE public stock_2002 postgres
230; 1259 41294 TABLE public stock_2003 postgres
231; 1259 41297 TABLE public stock_2004 postgres
232; 1259 41300 TABLE public stock_2005 postgres
```

```
[...]
3433; 0 41291 TABLE DATA public stock_2002 postgres
3434; 0 41294 TABLE DATA public stock_2003 postgres
3435; 0 41297 TABLE DATA public stock_2004 postgres
3436; 0 41300 TABLE DATA public stock_2005 postgres
[...]
```

À noter que l'option `--table-and-children=stock` est encore nécessaire, sans quoi, toute la base de données serait exportée.

Voici le résultat que nous aurions obtenu avec l'option -T. Toutes les partitions de stock_2001 auraient été sauvegardées.

```
# Sauvegarde
$ pg_dump -d cave -U postgres -Fc --table-and-children=stock* -T stock_2001 >
  ↪  stock_pour_client.pgdump

# Inspection
$ pg_restore --list stock_meilleures_annees.pgdump

[...]
228; 1259 41285 TABLE public stock postgres
234; 1259 41314 TABLE public stock_2001_1 postgres  <-- la définition est gardée
236; 1259 41320 TABLE public stock_2001_2 postgres  <-- la définition est gardée
235; 1259 41317 TABLE public stock_2001_3 postgres  <-- la définition est gardée
229; 1259 41291 TABLE public stock_2002 postgres
230; 1259 41294 TABLE public stock_2003 postgres
231; 1259 41297 TABLE public stock_2004 postgres
232; 1259 41300 TABLE public stock_2005 postgres
[...]
3440; 0 41314 TABLE DATA public stock_2001_1 postgres  <-- les données sont
  ↪  conservées
3442; 0 41320 TABLE DATA public stock_2001_2 postgres  <-- les données sont
  ↪  conservées
3441; 0 41317 TABLE DATA public stock_2001_3 postgres  <-- les données sont
  ↪  conservées
3436; 0 41291 TABLE DATA public stock_2002 postgres
3437; 0 41294 TABLE DATA public stock_2003 postgres
3438; 0 41297 TABLE DATA public stock_2004 postgres
3439; 0 41300 TABLE DATA public stock_2005 postgres
[...]
```

Enfin la dernière option `--exclude-table-data-and-children` permet de ne pas sauvegarder le contenu de la table et de ses partitions, mais uniquement la définition. Par exemple, si notre caviste s'est rendu compte d'une erreur sur toute l'année 2001 qui doit entièrement être reprise, une commande de sauvegarde pourrait être :

```
# Sauvegarde
$ pg_dump -d cave -U postgres -Fc --table-and-children=stock*
 ↪  --exclude-table-data-and-children=stock_2001 > stock_reset_2001.pgdump

# Inspection
$ pg_restore --list stock_reset_2005.pgdump

[...]
228; 1259 41285 TABLE public stock postgres
233; 1259 41311 TABLE public stock_2001 postgres    <-- la définition est gardée
234; 1259 41314 TABLE public stock_2001_1 postgres  <-- la définition est gardée
236; 1259 41320 TABLE public stock_2001_2 postgres  <-- la définition est gardée
235; 1259 41317 TABLE public stock_2001_3 postgres  <-- la définition est gardée
229; 1259 41291 TABLE public stock_2002 postgres
230; 1259 41294 TABLE public stock_2003 postgres
231; 1259 41297 TABLE public stock_2004 postgres
232; 1259 41300 TABLE public stock_2005 postgres
[...]
3436; 0 41291 TABLE DATA public stock_2002 postgres  <-- les données conservées
 ↪  débutent en 2002
3437; 0 41294 TABLE DATA public stock_2003 postgres
3438; 0 41297 TABLE DATA public stock_2004 postgres
3439; 0 41300 TABLE DATA public stock_2005 postgres
```

2.8 LZ4 ET ZSTD PEUVENT ÊTRE UTILISÉS AVEC PG_DUMP

- Deux nouveaux algorithmes de compression supportés par pg_dump :
  - `zstd`
  - `lz4`
- Option `-Z / --compress`

Les algorithmes de compression `zstd` et `lz4` sont désormais supportés par l'utilitaire pg_dump. Le choix de l'algorithme se fait grâce à l'option `-Z / --compress` de la commande. Elle peut prendre les valeurs `gzip`, `lz4`, `zstd` ou none.

Voici à titre d'exemple trois exports compressés d'une base de 19Go ainsi que le temps d'exécution nécessaire pour les obtenir.

```
# gzip
$ time pg_dump -U postgres -h 127.0.0.1 > /tmp/gzip.pgdump
real    0m52,381s
user    0m50,106s
sys   0m2,108s

# lz4
$ time pg_dump -U postgres -Z lz4 -h 127.0.0.1 > /tmp/lz4.pgdump
real    0m47,370s
user    0m13,372s
sys   0m5,894s

# zstd
$ time pg_dump -U postgres -Z zstd -h 127.0.0.1 > /tmp/zstd.pgdump
real    0m48,629s
user    0m15,789s
sys 0m4,957s

# tailles
$ ls -hl /tmp/*.pgdump
-rw-rw-r-- 1 dalibo dalibo 406M sept. 11 16:37 /tmp/gzip.pgdump
-rw-rw-r-- 1 dalibo dalibo 743M sept. 11 16:38 /tmp/lz4.pgdump
-rw-rw-r-- 1 dalibo dalibo 131M sept. 11 16:39 /tmp/zstd.pgdump
```

Cet exemple nous montre que les exports sont moins volumineux avec l'option `zstd` et se font plus

rapidement avec les options `lz4` et `zstd`.

Des détails pour la compression peuvent être spécifiés. Par exemple, avec un entier, cela définit le niveau de compression. Le format d'archive `tar` ne supporte pas du tout la compression.

```
$ time pg_dump -U postgres -Z gzip:1 -h 127.0.0.1 > /tmp/gzip1.pgdump
real    0m7,402s

$ time pg_dump -U postgres -Z gzip:6 -h 127.0.0.1 > /tmp/gzip6.pgdump
real    0m10,373s

$ time pg_dump -U postgres -Z gzip:9 -h 127.0.0.1 > /tmp/gzip9.pgdump
real    0m15,967s

$ ls -hl /tmp/gzip*
-rw-rw-r-- 1 dalibo dalibo 83M sept. 28 17:05 /tmp/gzip1.pgdump
-rw-rw-r-- 1 dalibo dalibo 82M sept. 28 17:05 /tmp/gzip6.pgdump
-rw-rw-r-- 1 dalibo dalibo 79M sept. 28 17:05 /tmp/gzip9.pgdump
```

Les tailles et les temps sont purement indicatifs. Les tailles et les durées seront différentes selon les bases traitées, leurs volumétries, leurs contenus ou encore le système sous-jacent.

Prendre le temps de choisir l'algorithme de compression est donc essentiel mais peut apporter de nombreux bénéfices.

2.9 CONTRÔLE DE L'UTILISATION DE LA MÉMOIRE PARTAGÉE PAR ANALYZE ET VACUUM

- Nouvelle option BUFFER_USAGE_LIMIT
  - VACUUM
  - ANALYZE
- Nouveau paramètre de configuration
  - vacuum_buffer_usage_limit

Une nouvelle option est désormais disponible pour contrôler la stratégie d'accès aux buffers de la mémoire partagée par les commandes VACUUM et ANALYZE. Cette option est nommée BUFFER_USAGE_LIMIT. Elle permet de limiter, ou non, la quantité de buffers accédés par ces commandes.

Une grande valeur permettra, par exemple, une exécution plus rapide de VACUUM, mais pourra avoir un impact négatif sur les autres requêtes qui verront la mémoire partagée disponible se réduire. La plus petite valeur configurable est de 128 ko et la plus grande est de 16 Go.

En plus de cette option, le nouveau paramètre de configuration vacuum_buffer_usage_limit voit le jour. Il indique si une stratégie d'accès à la mémoire est utilisée ou non. Si ce paramètre est initialisé à 0, cela désactive la stratégie d'accès à la mémoire partagée. Il n'y a alors aucune limite en terme d'accès aux buffers. Autrement, ce paramètre indique le nombre maximal de buffers accessibles par les commandes VACUUM, ANALYZE et le processus d'autovacuum. La valeur par défaut est de 256 ko.

La valeur passée en argument est par défaut comprise en kilo-octets si aucune unité n'est précisée. L'utilisation de cette option se fait de la manière suivante :

```
ANALYZE (BUFFER_USAGE_LIMIT 1024);
```

Pour essayer de montrer l'incidence de cette configuration sur le comportement d'un VACUUM, voici un script qui effectue une opération de VACUUM avec quatre valeurs différentes pour l'option BUFFER_USAGE_LIMIT :

- 0 : qui permet de désactiver la stratégie d'accès à la mémoire partagée ;
- 256 : qui est la valeur par défaut ;

  - 1024 ;
  - et 4096.

```
#!/bin/bash

#echo "\timing" >> .psqlrc # décommenter si le \timing n'est pas présent dans votre
↪  fichier .psqlrc

export PGUSER=postgres
psql -c "alter system set track_wal_io_timing to on";

for i in 0 256 1024 4096
do
        pgbench --quiet -i -s 300 -d postgres
        psql --quiet -c "create index on pgbench_accounts (abalance);"
        psql --quiet -c "update pgbench_accounts set bid = 0 where aid <= 10000000;"
        systemctl stop postgresql-16
        systemctl start postgresql-16
        psql --quiet -c "select pg_stat_reset_shared('wal');"
        echo "### Test BUFFER_USAGE_LIMIT $i ###"
        psql -c "VACUUM (BUFFER_USAGE_LIMIT $i);"
        psql -c "select wal_sync, wal_sync_time from pg_stat_wal;"
done
```

Les résultats suivants sont obtenus :

BUFFER_USAGE_LIMIT (ko)	VACUUM (ms)	wal_sync	wal_sync_time (ms)
0	7612	71	1578
256	12756	12004	6763
1024	10137	3031	4371
4096	8280	789	2855

Quelles conclusions en tirer ?

Par défaut, `BUFFER_USAGE_LIMIT` limite l'accès à la mémoire partagée en autorisant l'accès à 256 ko de mémoire à l'opération exécutée (voir la documentation officielle[2] pour connaitre la liste des opérations concernées). Celle-ci ne pourra utiliser que cette quantité de `buffers` pour ses opérations. Si de la mémoire supplémentaire est nécessaire, elle devra recycler certains `buffers`. Ce recyclage entraine une écriture de WAL sur disque, augmentant dès lors le temps d'exécution.

[2] https://www.postgresql.org/docs/current/glossary.html#GLOSSARY-BUFFER-ACCESS-STRATEGY

L'effet de la taille du `BUFFER_USAGE_LIMIT` se voit très clairement dans le tableau ci-dessous : plus la mémoire est grande, moins d'écritures de fichiers de transactions sont nécessaires et plus le temps d'exécution est rapide.

Lorsque `BUFFER_USAGE_LIMIT` est à 0, il n'y a pas de limitation quant au nombre de `buffers` que peut utiliser l'opération exécutée. Il y a alors très peu de recyclage nécessaire. Nous avons donc de meilleurs temps d'exécution et moins d'écritures sur disque. Pour autant, il ne faut pas oublier qu'avec cette configuration là, les autres requêtes pourront utiliser moins de mémoire et verrons donc leurs performances être dégradées.

Il est imaginable de positionner ce paramètre à 0 dans le cas d'une plage de maintenance où il serait possible d'utiliser le maximum de mémoire partagée.

2.10 AJOUT DES OPTIONS --SCHEMA ET --EXCLUDE-SCHEMA DANS VACUUMDB

- Deux nouvelles options à `vacuumdb`
  - `--schema`
  - `--exclude-schema`

Deux nouvelles options sont maintenant disponibles dans l'utilitaire `vacuumdb`. `--schema` et `--exclude-schema` permettent soit d'effectuer l'opération de `VACUUM` sur toutes les tables des schémas indiqués, soit, à l'inverse, de les exclure de l'opération.

Ces options peuvent respectivement être appelées avec les options `-n` et `-N`. Il n'est pas possible d'utiliser ces nouvelles options avec les options `-a` et `-t`. Un message d'erreur explicite sera renvoyé.

```
$ vacuumdb --schema public -U postgres -d postgres -t pgbench_accounts
vacuumdb: error: cannot vacuum all tables in schema(s) and specific table(s) at the
↪  same time
```

Une des raisons qui est à l'origine de cette amélioration est la trop forte fragmentation du schéma `pg_catalog` lorsque de nombreux objets temporaires sont créés. Jusqu'à présent, il n'y avait pas de moyen simple pour lancer des opérations de `VACUUM` sur ce schéma, il fallait donc passer sur chacune des tables. Dans un contexte de production, si on sait que de nombreuses opérations sont faites sur la majorité des tables d'un schéma, cette option permet de gagner du temps en indiquant le schéma sur lequel effectuer le `VACUUM` et non plus les tables une à une.

2.11 AJOUT DES OPTIONS SKIP_DATABASE_STATS ET ONLY_DATABASE_STATS

- Gestion de la mise à jour des statistiques pour VACUUM
- Nouvelles options de VACUUM
  - SKIP_DATABASE_STATS
  - ONLY_DATABASE_STATS
- Intégré à vacuumdb
  - SKIP_DATABASE_STATS activé par défaut en v16
  - ONLY_DATABASE_STATS si pas d'ANALYZE par étapes

Durant l'exécution d'un VACUUM, ou d'un autovacuum, une fonction particulière est appelée. Elle permet de mettre à jour l'entrée `datfrozenxid` de la table `pg_database` pour chaque base de données présente dans l'instance.

Cette entrée permet de connaitre l'identifiant de transaction le plus petit de la base et est utilisée pour déterminer si une table de cette base doit être nettoyée ou non.

Cette fonction passe en revue toutes les lignes de la table `pg_class` pour une base donnée. Elle le fait de manière séquentiel. Les performances se voyaient être dégradées sur des bases de données avec des dizaines de milliers de tables.

De plus, des outils comme `vacuumdb` exécutent les commandes de `VACUUM` sur chaque table. Ainsi à chaque passage sur une table, la fonction est appelée. On comprend bien que plus il y a de tables, plus le temps et les performances seront dégradés.

L'option `SKIP_DATABASE_STATS` (true ou false) permet d'indiquer si `VACUUM` doit ignorer la mise à jour de l'identifiant de transaction.

L'option `ONLY_DATABASE_STATS` (true ou false) permet d'indiquer que `VACUUM` ne doit rien faire d'autre à part mettre à jour l'identifiant.

L'outil `vacuumdb` a été mis à jour pour utiliser automatiquement l'option `SKIP_DATABASE_STATS` si le serveur est au minimum en version 16. Il utilise ensuite, tout aussi automatiquement, l'option `ONLY_DATABASE_STATS` une fois qu'il a traité toutes les tables à condition que l'option `--analyze-in-stages` ne soit pas indiquée.

2.12 OPTIMISATION DE ANALYZE AVEC POSTGRES_FDW

- postgres_fdw
- ANALYZE plus efficace sur des tables distantes
- Option analyze_sampling
  - SERVER
  - FOREIGN TABLE

Le calcul de statistiques sur des tables distantes avec l'extension postgres_fdw est nettement amélioré. Jusqu'à présent, lorsque ANALYZE était exécuté sur une table distante, l'échantillonnage était effectué localement à l'instance. Les données étaient donc intégralement rapatriées avant que ne soient effectuées les opérations d'échantillonnage. Pour des grosses tables, cette manière de faire était tout sauf optimisée.

Il est désormais possible d'effectuer l'échantillonnage sur le serveur distant grâce à l'option analyze_sampling. La volumétrie transférée est alors bien plus basse. Le calcul des statistiques des données sur cet échantillon se fait toujours sur l'instance qui lance ANALYZE. Cette option peut prendre les valeurs off, auto, system, bernoulli et random. La valeur par défaut est auto qui permettra d'utiliser soit bernoulli soit random. Elle peut être appliquée soit sur l'objet SERVER soit sur la FOREIGN TABLE.

Prenons l'exemple d'une table de 20 millions de lignes avec une seule colonne uuid. Les différences entre les temps d'exécution sont notables. Lorsque les données sont récupérées, il faut presque 7 secondes pour y arriver, moins de 1 secondes dans tous les autres cas. Le test a été fait avec deux instances sur un même poste. Dans le cas d'instances séparées sur des datacenters ou VLAN différents, les temps de latence pourraient être encore plus impactant.

```
-- Sur le serveur distant :
CREATE TABLE t1_fdw AS SELECT gen_random_uuid() AS id FROM
↪  generate_series(1,20000000);
```

```
-- Sur l'instance locale :
postgres=# CREATE EXTENSION postgres_fdw;
postgres=# CREATE SERVER serveur2
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '10.0.3.114',
           port '5432',
           dbname 'postgres') ;
```

```
postgres=# CREATE USER MAPPING FOR postgres SERVER serveur2 OPTIONS (user
 ↪  'postgres');
CREATE USER MAPPING

postgres=# CREATE FOREIGN TABLE t1_fdw (id uuid) SERVER serveur2 OPTIONS (
 ↪  analyze_sampling 'off');
CREATE FOREIGN TABLE

-- off
postgres=# ANALYZE VERBOSE t1_fdw;
INFO:  analyzing "public.t1_fdw"
INFO:  "t1_fdw": table contains 20000000 rows, 30000 rows in sample
ANALYZE
Time: 6922,019 ms (00:06,922)

-- random
postgres=# ALTER FOREIGN TABLE t1_fdw OPTIONS ( SET analyze_sampling 'random');
ALTER FOREIGN TABLE
postgres=# ANALYZE VERBOSE t1_fdw;
INFO:  analyzing "public.t1_fdw"
INFO:  "t1_fdw": table contains 19998332 rows, 29969 rows in sample
ANALYZE
Time: 629,190 ms

-- system
postgres=# ALTER FOREIGN TABLE t1_fdw OPTIONS ( SET analyze_sampling 'system');
ALTER FOREIGN TABLE
postgres=# ANALYZE VERBOSE t1_fdw;
INFO:  analyzing "public.t1_fdw"
INFO:  "t1_fdw": table contains 19998332 rows, 30000 rows in sample
ANALYZE
Time: 82,832 ms

-- bernoulli
postgres=# ALTER FOREIGN TABLE t1_fdw OPTIONS ( SET analyze_sampling 'bernoulli');
ALTER FOREIGN TABLE
postgres=# ANALYZE VERBOSE t1_fdw;
INFO:  analyzing "public.t1_fdw"
INFO:  "t1_fdw": table contains 19998332 rows, 29875 rows in sample
ANALYZE
Time: 303,548 ms
```

2.13 REFONTE DU SYSTÈME DE DÉLÉGATION DE DROITS

- Améliorations sur le droit ADMIN OPTION
- Retourne une erreur s'il est réappliqué au donneur du droit
- REVOKE ADMIN OPTION ... CASCADE

La déléguation de droits correspond à la capacité pour un utilisateur d'attribuer à un autre utilisateur un droit qu'on lui aurait octroyé avec la clause ADMIN OPTION. Dans les versions précédentes, la table pg_auth_members ne permettait pas de gérer plusieurs donneurs d'un même droit à un même utilisateur.

Ainsi, lorsqu'un utilisateur se voyait octroyer un droit avec l'option ADMIN, il ne lui était pas interdit de retirer ce droit à celui qui le lui avait donné, sans avoir besoin d'être superutilisateur.

```
v15=# CREATE ROLE role_adm;
v15=# GRANT role_adm TO user1 WITH ADMIN OPTION;
GRANT ROLE

v15=# SET ROLE = user1;
v15=> GRANT role_adm TO user2 WITH ADMIN OPTION;
GRANT ROLE
v15=> SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
 ↪   WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user1  | postgres | t
 user2  | user1    | t

v15=> SET ROLE = user2;
v15=> REVOKE ADMIN OPTION FOR role_adm FROM user1;
REVOKE ROLE
v15=> SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
 ↪   WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user2  | user1    | t
 user1  | postgres | f
(2 rows)

v15=> SET ROLE = user1;
```

```
v15=> GRANT role_adm TO user3;
ERROR:  must have admin option on role "role_adm"
```

On constate que la ligne de la table `pg_auth_members` a été modifiée avec le changement de la colonne `admin_option` passée de `true` à `false` alors même que ce droit `ADMIN` lui avait été octroyé par un rôle plus puissant que `user2`. La version 16 étend la contrainte d'unicité de la table `pg_auth_members` à la colonne `grantor`. Ainsi, un `REVOKE` par un tiers ne supprimera pas la délégation d'un droit entre deux autres rôles.

```
v16=> SET ROLE = user2;
v16=> REVOKE ADMIN OPTION FOR role_adm FROM user1;
REVOKE ROLE
v16=> SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
 ↪  WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user1  | postgres | t
 user2  | user1    | t
 user1  | user2    | f

v16=> SET ROLE = user1;
v16=> GRANT role_adm TO user3;
GRANT ROLE
```

Ce changement a également été l'occasion d'enrichir les exceptions possibles inhérentes à la relation de délégation entre plusieurs rôles. L'exemple ci-dessous montre qu'un rôle ne peut pas réattribuer un droit `ADMIN` à son propre donneur.

```
v16=# CREATE ROLE role_adm;
v16=# GRANT role_adm TO user1 WITH ADMIN OPTION;
GRANT ROLE

v16=# SET ROLE = user1;
v16=> GRANT role_adm TO user2 WITH ADMIN OPTION;

v16=> SET ROLE = user2;
v16=> GRANT role_adm TO user1 WITH ADMIN OPTION;
ERROR:  ADMIN option cannot be granted back to your own grantor
```

Ce comportement était permis avant la version 16 et renvoyait simplement un avertissement.

```
v15=# SET ROLE = user2;
v15=> GRANT role_adm TO user1 WITH ADMIN OPTION;
NOTICE:  role "user1" is already a member of role "role_adm"
GRANT ROLE
```

Puisque la notion de donneur est maintenue entre plusieurs niveaux hiérarchiques de rôles, la nouvelle version permet d'empêcher la révocation d'un droit lorsque celui-ci a été octroyé à d'autres utilisateurs. L'exemple suivant montre qu'une erreur est renvoyée au client avec pour conseil d'utiliser l'instruction `REVOKE ... CASCADE`.

```
v16=# GRANT role_adm TO user1 WITH ADMIN OPTION;
v16=# GRANT role_adm TO user2 WITH ADMIN OPTION GRANTED BY user1;

v16=# SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
↪  WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user1  | postgres | t
 user2  | user1    | t

v16=# REVOKE ADMIN OPTION FOR role_adm FROM user1;
ERROR:  dependent privileges exist
HINT:  Use CASCADE to revoke them too.

v16=# REVOKE ADMIN OPTION FOR role_adm FROM user1 CASCADE;
REVOKE ROLE

v16=# SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
↪  WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user1  | postgres | f
```

Ainsi, le rôle `user1` dispose encore du droit octroyé par `postgres` mais n'est plus en capacité de le donner à d'autres utilisateurs. L'action `REVOKE ... CASCADE` est rétroactive, avec le retrait définitif des droits pour les rôles qui en ont bénéficié (ici, `user2` n'a plus le droit que `user1` lui a donné). Dans les versions précédentes, une telle opération aboutissait et l'utilisateur intermédiare ne disposait plus de son droit `ADMIN`, sans que cela ne retire le moindre droit aux rôles en bas de la hiérarchie.

```
v15=# SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
↪  WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user1  | postgres | t
 user2  | user1    | t
 user3  | user2    | t

v15=# REVOKE ADMIN OPTION FOR role_adm FROM user1;
REVOKE ROLE
```

```
v15=# SELECT member::regrole, grantor::regrole, admin_option FROM pg_auth_members
↪  WHERE roleid = 'role_adm'::regrole;
 member | grantor  | admin_option
--------+----------+--------------
 user2  | user1    | t
 user3  | user2    | t
 user1  | postgres | f
```

2.14 NOUVEAU PARAMÈTRE LIBPQ : REQUIRE_AUTH

- Nouveau paramètre de la `libpq`
  - `require_auth`
- Liste de mots clés séparés par une virgule
- Mots clés
  - password, md5, gss, sspi, scram-sha-256, creds, none

Le paramètre de connexion `require_auth` permet à un client libpq de définir une liste de méthodes d'authentification qu'il accepte. Si le serveur ne présente pas une de ces méthodes d'autentification, les tentatives de connexion échoueront.

La liste des paramètres utilisables est :

- password
- md5
- gss
- sspi
- scram-sha-256
- creds
- none (utile pour contrôler si le serveur accepte des connexions non authentifiées)

Il est également possible d'utiliser `!` avant la méthode pour indiquer que le serveur ne doit pas utiliser le paramètre en question, comme par exemple `!md5`.

Prenons l'exemple d'une instance PostgreSQL ayant le contenu suivant dans le fichier `pg_hba.conf`. Il autorise les connexions uniquement en local avec comme méthode d'authentification `scram-sha-256` :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
```

Regardons le comportement d'une connexion avec `psql` et avec des valeurs différentes de `require_auth` :

```
# Une connexion sans spécifier require_auth fonctionne bien
$ psql -U postgres -h 127.0.0.1
```

```
Password for user postgres:

# Avec require_auth=md5, la connexion échoue car cette méthode n'est pas prise en
↪  compte par l'instance
$ psql -U postgres -h 127.0.0.1 "require_auth=md5"
psql: error: connection to server at "127.0.0.1", port 5432 failed: authentication
↪  method requirement "md5" failed: server requested SASL authentication

# Dès lors que scram-sha-256 est renseigné, la connexion fonctionne
psql -U postgres -h 127.0.0.1 "require_auth=scram-sha-256"
Password for user postgres:

# Ou encore
psql -U postgres -h 127.0.0.1 "require_auth=md5,scram-sha-256"
Password for user postgres:

# Si pour une raison, scram-sha-256 ne peut pas être utilisé par le client
↪  (utilisation de !)
# la connexion échoue
$ psql -U postgres -h 127.0.0.1 "require_auth=\!scram-sha-256"
psql: error: connection to server at "127.0.0.1", port 5432 failed: authentication
↪  method requirement "!scram-sha-256" failed: server requested SASL authentication
```

Bien que ce paramètre permette à un client de restreindre les méthodes d'authentification qu'il souhaite utiliser, il n'est reste pas moins que c'est bien le fichier `pg_hba.conf` de l'instance qui va forcer la méthode utilisée.

2.15 SÉLECTION ALÉATOIRE DES HOSTS PAR LIBPQ

- Répartiton de la charge de connexions entre plusieurs instances
- Nouveau paramètre `libpq`
  - `load_balance_hosts=<string>`

Un nouveau paramètre de connexion voit le jour au niveau de `libpq`. Il permet de faire de la répartition de charge au niveau des connexions à plusieurs instances PostgreSQL. Le paramètre `load_balance_hosts=<string>` peut prendre plusieurs valeurs :

- `disable` (valeur par défaut)
- `random`

Dans le premier cas, les tentatives de connexions se font de manière séquentielle, les adresses sont testées dans l'ordre. Si des noms DNS sont indiqués, ils seront résolus puis les connexions se feront selon l'ordre de la ou les adresses IP obtenues par la résolution DNS.

Lorsque `random` est utilisé, l'ordre de prise en compte est aléatoire. Si une résolution DNS est nécessaire, l'ordre des adresses IP obtenues sera lui aussi mélangé pour ne pas toujours se connecter à la même adresse IP pour un nom de domaine donné.

Il est à noter que cette répartition de charge se fait au niveau des connexions et non pas au niveau des transactions. Cela signifie qu'un contrôle est tout de même nécessaire sur les transactions qui sont effectuées après la connexion.

Par exemple, dans le cas suivant, nous avons trois instances PostgreSQL dont deux qui se trouvent être des secondaires en lecture seule. Il est donc possible d'effectuer des `SELECT` sur toutes les instances. Dans cet exemple, le `SELECT` renvoie l'adresse IP de l'instance, mais il est facile d'étendre cet exemple à des requêtes plus fonctionnelles.

```
$ cat /etc/hosts
...
10.0.3.114  pg16_1
10.0.3.19   pg16_2
10.0.3.97   pg16_3
```

```
$ for i in {1..10}; do PGPASSWORD=dalibo psql -At 'user=dalibo dbname=dalibo
↪  host=pg16_1,pg16_2,pg16_3 load_balance_hosts=random' -c "select
↪  inet_server_addr();" ; done
```

```
10.0.3.97
10.0.3.114
10.0.3.114
10.0.3.19
10.0.3.19
10.0.3.114
10.0.3.97
10.0.3.114
10.0.3.19
10.0.3.97
```

Toutes les instances ont répondues correctement.

Maintenant, si la requête passée est un `INSERT` sur la base `dalibo`, des messages d'erreurs apparaissent lors de l'exécution sur les secondaires. Ceci est logique puisque ces instances là sont en lecture seule. Par exemple :

```
$ for i in {1..10}; do PGPASSWORD=dalibo psql -At 'user=dalibo dbname=dalibo
↪  host=pg16_1,pg16_2,pg16_3 load_balance_hosts=random' -c "select
↪  inet_server_addr(); insert into test_random values ('1');" ; done

10.0.3.19 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.97 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.114 # <-- primaire
INSERT 0 1
10.0.3.19 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.97 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.19 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.114 # <-- primaire
INSERT 0 1
10.0.3.19 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
10.0.3.114 # <-- primaire
INSERT 0 1
10.0.3.19 # secondaire
ERROR:  cannot execute INSERT in a read-only transaction
```

Rappelons au passage que l'option `target_session_attrs` permet de spécifier à quel type d'instance le client peut se connecter. Par exemple `target_session_attrs=primary` permet au client de se connecter uniquement sur des instances primaires. Le test précédent ne remonte plus d'erreur.

```
$ for i in {1..10}; do PGPASSWORD=dalibo psql -At 'user=dalibo dbname=dalibo
↪  host=pg16_1,pg16_2,pg16_3 load_balance_hosts=random
↪  target_session_attrs=primary' -c "select inet_server_addr(); insert into
↪  test_random values ('1');" ; done

10.0.3.114 # <-- primaire
INSERT 0 1
10.0.3.114 # <-- primaire
INSERT 0 1
...
```

Cette nouvelle option permet de manière très simple d'effectuer de la répartition de charge sur plusieurs secondaires de manière équilibrée ou pondérée. Attirons néanmoins l'attention sur le fait que la répartition se fait au niveau des requêtes, et non pas au niveau de la charge réelle.

Reprenons l'exemple avec cette fois-ici uniquement nos deux serveurs secondaires. Pour avoir une répartition équilibrée, rien de plus simple, il suffit d'indiquer les deux serveurs. On peut facilement estimer la répartition entre les deux instances avec une combinaison de commandes `grep 97` (97 faisant parti de l'adresse IP de pg16_3) et wc.

```
$ for i in {1..10}; do PGPASSWORD=dalibo psql -At 'user=dalibo dbname=dalibo
↪  host=pg16_2,pg16_3 load_balance_hosts=random' -c "select inet_server_addr(); " ;
↪  done | grep 97 | wc
```

En modifiant le nombre de passages dans la boucle, on obtient le tableau suivant, montrant une répartition équilibrée des requêtes :

itérations	10	100	1000	10000
pg16_2	6	46	489	5009
pg16_3	4	54	511	4991

Si désormais, on veut favoriser l'utilisation du secondaire `pg16_3`, il suffit de le rajouter une seconde fois dans la ligne de commande, afin d'obtenir, par exemple un ratio 1/3 - 2/3 en terme d'utilisation des secondaires `pg16_2` et `pg16_3`.

```
$ for i in {1..10}; do PGPASSWORD=dalibo psql -At 'user=dalibo dbname=dalibo
↪  host=pg16_2,pg16_3,pg16_3 load_balance_hosts=random' -c "select
↪  inet_server_addr();" ; done | grep 97 | wc
```

itérations	10	100	1000	10000
pg16_2	3	34	332	3322
pg16_3	7	66	668	6678

De manière très simple, il est désormais possible de faire de la répartion de charge en lecture sur plusieurs secondaires avec `libpq`.

3/ Réplication

3.1 DÉCODAGE LOGIQUE SUR LES INSTANCES SECONDAIRES

- Permet de :
  - créer un slot de réplication logique sur une standby
  - lancer le décodage logique sur une standby (!= réplication logique)
  - souscrire à une publication créée sur le primaire depuis une standby
- Invalidation du slot de réplication logique en cas de :
  - conflit de réplication : utiliser un slot de réplication physique et le `hot_standby_feedback`
  - réduction du `wal_level` sur l'instance principale
- Nouveau champ `confl_active_logicalslot` dans `pg_stat_database_conflicts`
- Nouveau champ `conflicting` dans `pg_replication_slots`

Dans cette version de PostgreSQL, il est désormais possible de :

- créer un slot de réplication logique sur une standby ;
- lancer le décodage logique sur une standby ;
- souscrire à une publication créée sur le primaire depuis une standby.

Pour cela un certain nombre de changements ont dû être faits au niveau de l'infrastructure de PostgreSQL.

3.1.1 Modification de la structure des WAL

Sur une instance primaire, pour éviter de rejouer des modifications du catalogue qui sont nécessaires à la réplication logique, PostgreSQL utilise l'information `catalog_xmin` associée au slot de réplication (et que l'on retrouve dans `pg_replication_slots`).

Si l'on utilise le décodage logique sur une instance secondaire, cette information ne sera pas toujours disponible depuis l'instance primaire. Cette dernière risque donc de nettoyer les lignes du catalogue système qui sont nécessaires au décodage logique.

Deux stratégies de mise en place de la réplication sont concernées par ce problème :

- `hot_standby_feedback` est désactivé ;
- `hot_standby_feedback` est activé sans slot de réplication, dans ce cas à la première déconnexion, la valeur du `catalog_xmin` est perdue.

Il a donc fallu ajouter des informations dans les journaux de transactions pour marquer les modifications qui concernent le catalogue système et sont nécessaires au décodage logique.

3.1.2 Mettre en place une gestion des conflits sur les standby

Deux sources de conflits sont identifiées sur une instance secondaire :

1. des lignes du catalogue nécessaires au décodage logique sont supprimées ;
2. le paramètre `wal_level` est passé de `logical` à `replica` sur la primaire.

Dans ces deux cas, le slot de réplication doit être invalidé.

La colonne `confl_active_logicalslot` a été ajoutée à la vue `pg_stat_replication_conflicts` pour détecter cette nouvelle source de conflits.

3.1.3 Création d'un slot de réplication sur une instance secondaire

Grâce aux modifications décrites précédemment, il est désormais possible d'activer le décodage logique sur les instances secondaires. Pour cela, il faut créer un slot de réplication logique.

Lorsque l'on passe la commande suivante, qui crée un slot de réplication logique avec le plugin `test_decoding`, il est possible que la commande mette un certain temps à être prise en compte.

```sql
SELECT pg_create_logical_replication_slot('slot_standby', 'test_decoding');
```

```
 pg_create_logical_replication_slot
------------------------------------
 (slot_standby,0/205A658)
(1 row)
```

Cette attente est due au fait que, pour créer le slot, l'instance secondaire doit traiter un enregistrement de WAL de type `xl_running_xact`. Cet enregistrement contient des informations sur le prochain numéro de transaction, le numéro de transaction active le plus ancien, le numéro de la dernière transaction qui s'est terminée et un tableau de transactions actives. Pour qu'il soit envoyé, il faut qu'il y ait de l'activité sur l'instance primaire.

La commande suivante peut être exécutée sur le primaire afin de forcer l'écriture d'un tel enregistrement et ainsi débloquer la création du slot.

```sql
SELECT pg_log_standby_snapshot();
```

```
 pg_log_standby_snapshot
-------------------------
 0/205A658
(1 row)
```

Le slot est bien visible sur l'instance secondaire :

```
TABLE pg_replication_slots \gx
```

```
-[ RECORD 1 ]-------+--------------
slot_name           | slot_standby
plugin              | test_decoding
slot_type           | logical
datoid              | 5
database            | postgres
temporary           | f
active              | f
active_pid          | ¤
xmin                | ¤
catalog_xmin        | 777
restart_lsn         | 0/209E148
confirmed_flush_lsn | 0/209E180
wal_status          | reserved
safe_wal_size       | ¤
two_phase           | f
conflicting         | f
```

3.1.4 Décodage logique sur les instances secondaires

Si on crée de l'activité dans une table créée au préalable sur le primaire :

```
-- Définition de la table : CREATE TABLE matable(i int);
INSERT INTO matable VALUES (1),(2),(3);
DELETE FROM matable WHERE i = 1;
TRUNCATE matable ;
```

Les informations du décodage logique peuvent être consommées, soit :

- avec la fonction `pg_logical_slot_get_changes()` :

  ```
  SELECT *
    FROM pg_logical_slot_get_changes('slot_standby', NULL, NULL, 'include-xids',
     ↪ '0');
  ```

  ```
      lsn     | xid |                          data
  -----------+-----+-------------------------------------------------
   0/20935A8 | 769 | BEGIN
  ```

```
 0/20935A8 | 769 | table public.matable: INSERT: i[integer]:1
 0/20935E8 | 769 | table public.matable: INSERT: i[integer]:2
 0/2093628 | 769 | table public.matable: INSERT: i[integer]:3
 0/2093698 | 769 | COMMIT
 0/2093698 | 770 | BEGIN
 0/2093698 | 770 | table public.matable: DELETE: (no-tuple-data)
 0/2093700 | 770 | COMMIT
 0/2093700 | 771 | BEGIN
 0/2093978 | 771 | table public.matable: TRUNCATE: (no-flags)
 0/2093A20 | 771 | COMMIT
(11 rows)
```

- en ligne de commande avec l'outil `pg_recvlogical` :

```
pg_recvlogical --slot slot_standby --dbname postgres --start --file -
```

```
BEGIN 772
table public.matable: INSERT: i[integer]:1
table public.matable: INSERT: i[integer]:2
table public.matable: INSERT: i[integer]:3
COMMIT 772
BEGIN 773
table public.matable: DELETE: (no-tuple-data)
COMMIT 773
BEGIN 774
table public.matable: TRUNCATE: (no-flags)
COMMIT 774
```

- ou avec un outil développé par vos soins.

3.1.5 Publications sur une instance secondaire

Il n'est pas encore possible de créer de publication sur une instance secondaire, car l'instance est ouverte en lecture seule. Mais dans ce cas, pourquoi peut-on créer un slot de réplication ?

La création d'un slot est possible, car le slot est représenté par un fichier dans l'arborescence de PostgreSQL. Dans notre cas, c'est `$PGDATA/pg_replslot/slot_standby/state`. `pg_replication_slot` est une vue qui permet de visualiser les données de ce fichier.

```
-- description de la vue
\sv pg_replication_slots
-- ... et de la foncton associée
SELECT proname, description
  FROM pg_proc p
       INNER JOIN pg_description d ON p.oid = d.objoid
  WHERE proname = 'pg_get_replication_slots' \gx
```

```
CREATE OR REPLACE VIEW pg_catalog.pg_replication_slots AS
 SELECT l.slot_name,
    l.plugin,
    l.slot_type,
    l.datoid,
    d.datname AS database,
    l.temporary,
    l.active,
    l.active_pid,
    l.xmin,
    l.catalog_xmin,
    l.restart_lsn,
    l.confirmed_flush_lsn,
    l.wal_status,
    l.safe_wal_size,
    l.two_phase,
    l.conflicting
   FROM pg_get_replication_slots() l(slot_name, plugin, slot_type, datoid,
 ↪  temporary, active, active_pid, xmin, catalog_xmin, restart_lsn,
 ↪  confirmed_flush_lsn, wal_status, safe_wal_size, two_phase, conflicting)
     LEFT JOIN pg_database d ON l.datoid = d.oid
```

```
-[ RECORD 1 ]-----------------------------------------------------
proname     | pg_get_replication_slots
description | information about replication slots currently in use
```

Une publication est représentée par des méta-données écrites dans une table du catalogue (`pg_publication`) :

```
\dt pg_publication
```

```
              List of relations
   Schema   |      Name      | Type  |  Owner
------------+----------------+-------+----------
 pg_catalog | pg_publication | table | postgres
(1 row)
```

Il est par contre possible de créer une souscription qui pointe vers l'instance secondaire. Cette dernière connait les informations de la publication puisqu'elles sont disponibles dans le catalogue.

Créons une publication sur l'instance primaire :

```sql
CREATE PUBLICATION pub FOR TABLE matable;
```

On voit bien ses méta-données sur l'instance secondaire :

```sql
TABLE pg_publication_tables ;
TABLE pg_publication;
```

```
 pubname | schemaname | tablename | attnames | rowfilter
---------+------------+-----------+----------+-----------
 pub     | public     | matable   | {i}      | ¤
(1 row)

  oid  | pubname | pubowner | puballtables | pubinsert | pubupdate | pubdelete |
↪  pubtruncate | pubviaroot
-------+---------+----------+--------------+-----------+-----------+-----------+----
↪  ---------+------------
 16409 | pub     |       10 | f            | t         | t         | t         | t
↪   | f
(1 row)
```

Sur une troisième instance, créons une souscription (par soucis de simplicité la réplication utilise un socket sans authentification et l'utilisateur `postgres`):

```
CREATE TABLE matable(i int);
CREATE SUBSCRIPTION sub
       CONNECTION 'host=/var/run/postgresql port=5439 user=postgres dbname=postgres'
       PUBLICATION pub;
```

```
CREATE TABLE
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

Comme la souscription crée un slot sur l'instance secondaire, s'il n'y a pas d'activité sur le primaire, il faudra y exécuter la fonction `pg_log_standby_snapshot()` pour éviter l'attente.

Les modifications faites sur le primaire seront alors visibles dans la table sur la troisième instance.

On peut voir que le slot est bien créé sur l'instance secondaire et est actif :

```
TABLE pg_replication_slots \gx
```

```
-[ RECORD 1 ]-------+--------------
slot_name           | sub
plugin              | pgoutput
slot_type           | logical
datoid              | 5
database            | postgres
temporary           | f
active              | t
active_pid          | 610195
xmin                | ¤
catalog_xmin        | 777
restart_lsn         | 0/209E148
confirmed_flush_lsn | 0/209E180
wal_status          | reserved
```

```
safe_wal_size       | ¤
two_phase           | f
conflicting         | f
```

3.1.6 Conflits de réplication

Si l'on ne prend pas de précautions particulières, une modification du catalogue peut provoquer un conflit de réplication.

En guise d'exemple, ajoutons une clé primaire sur la table `matable` :

```
TRUNCATE matable;
ALTER TABLE matable PRIMARY KEY (i);
INSERT INTO matable(i) VALUES (1);
```

On constate que les informations n'atteignent pas la troisième instance.

En regardant dans les traces de l'instance, on voit le message.

```
ERROR:  could not receive data from WAL stream: ERROR:  canceling statement due to
 ↪  conflict with recovery
DETAIL:  User was using a logical replication slot that must be invalidated.
LOG:  background worker "logical replication worker" (PID 614482) exited with exit
 ↪  code 1
```

Sur l'instance secondaire, le slot est invalidé :

```
TABLE pg_replication_slots \gx
```

```
-[ RECORD 1 ]-------+----------
slot_name           | sub
plugin              | pgoutput
slot_type           | logical
datoid              | 5
database            | postgres
temporary           | f
active              | f
active_pid          | ¤
xmin                | ¤
catalog_xmin        | 803
restart_lsn         | 0/21A34E8
confirmed_flush_lsn | 0/21AFE88
wal_status          | lost
safe_wal_size       | ¤
two_phase           | f
conflicting         | t
```

Un conflit est aussi visible dans la vue `pg_stat_database_conflicts` :

```sql
SELECT datname, confl_active_logicalslot
  FROM  pg_stat_database_conflicts
 WHERE datname = 'postgres' \gx
```

```
-[ RECORD 1 ]------------+---------
datname                  | postgres
confl_active_logicalslot | 1
```

L'instance qui porte la souscription tente de se reconnecter en boucle, on trouve donc le message suivant dans les traces de l'instance secondaire.

```
STATEMENT:  START_REPLICATION SLOT "sub" LOGICAL 0/21AEEB8 (proto_version '4',
 ↪  origin 'any', publication_names '"pub"')
ERROR:  can no longer get changes from replication slot "sub"
DETAIL:  This slot has been invalidated because it was conflicting with recovery.
```

Il est possible de créer la souscription avec l'option `disable_on_error` afin d'éviter que la souscription ne se reconnecte en boucle :

```sql
CREATE SUBSCRIPTION sub
       CONNECTION 'host=/var/run/postgresql port=5439 user=postgres dbname=postgres'
       PUBLICATION pub
       WITH ( disable_on_error = true );
```

On voit alors le message suivant dans les traces de la troisième instance :

```
LOG:  subscription "sub" has been disabled because of an error
```

Mettre en place la réplication physique avec un slot de réplication et le `hot_standby_feedback` permet de se protéger contre ce genre de problème.

3.1.7 Bascules et décodage logique

Si on déclenche une bascule sur l'instance secondaire, la réplication logique continue de fonctionner sans interruption, comme le montre le schéma ci-dessous.

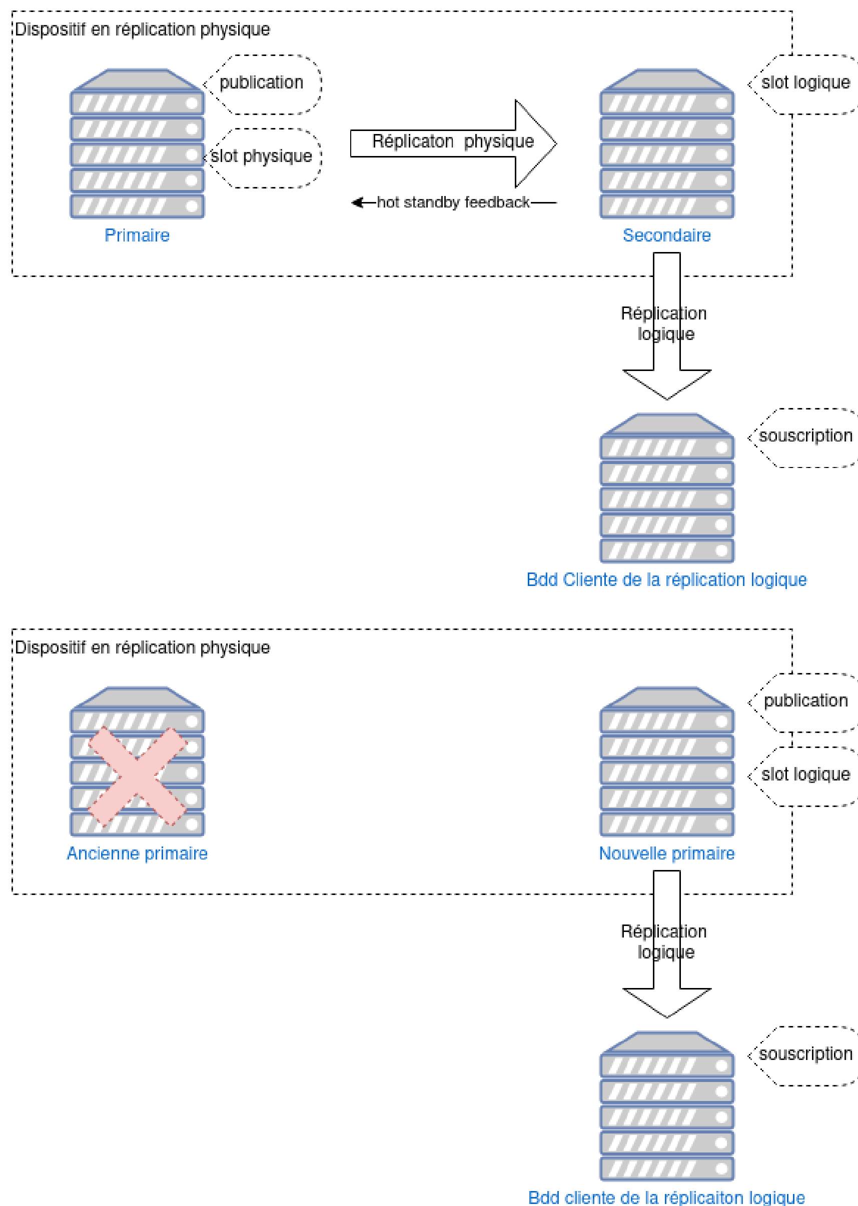

Figure 3/ .1: Bascules et décodage logique

Cette fonctionnalité ne permet donc pas de synchroniser les slots de réplication logique entre primaire et secondaire ce qui permettrait de faire basculer la réplication logique lors d'un failover (bascule non programmée). [Patroni] dispose d'une fonctionnalité qui permet de faire cela. Une [discussion] est en cours sur la mailing list hackers pour rendre cela possible directement dans PostgreSQL.

- Discussion[1]

[1] https://commitfest.postgresql.org/44/4423/

- Patroni[2]

---

[2] https://github.com/zalando/patroni/

3.2 PARALLÉLISME DE L'APPLICATION DES MODIFICATIONS

- Réplication logique
  - parallélisme lors de l'application
  - paramètre `streaming` d'une souscription

Avant la version 16, des transactions volumineuses étaient transmises par morceaux par le publieur et reçues par le souscripteur pour être appliquées. Avant d'être appliqués, les changements apportés par les transactions étaient écrits dans des fichiers temporaires, puis lorsque le commit était reçu, alors un worker lisait le contenu de ces fichiers temporaires pour les appliquer. Ceci était notamment fait afin d'éviter qu'un ROLLBACK inutile soit fait sur le souscripteur.

Ces changements peuvent désormais être appliqués de manière parallélisée et les fichiers temporaires ne sont plus utilisés. Un worker leader va recevoir les transactions à appliquer puis enverra les changements à un worker ou plusieurs workers qui travailleront en parallèle. L'échange des changements se fait désormais via la mémoire partagée. Dans le cas où le worker leader n'arrive plus à communiquer avec ses workers parallèles, il passera en mode "sérialisation partielle" et écrira les modifications dans un fichier temporaire pour conserver modifications à apporter.

Le parametre `streaming` d'un objet SUBSCRIPTION permet désormais de choisir si l'application des changements se fait de manière parallèle ou non grâce à la valeur `parallel` :

- `off` : Toutes les transactions sont décodées du côté du publieur puis envoyées entièrement au souscripteur.
- `on` : Les transactions sont décodées sur le publieur puis envoyées au fil de l'eau. Les changements sont écrits dans des fichiers temporaires du côté du souscripteur et ne sont appliqués que lorsque le commit a été fait sur le publieur et reçu par le souscripteur.
- `parallel` : Les changements sont directement appliqués en parallèle par des workers sur le souscripteur, si des workers sont disponibles.

3.3 NOUVEAU ROLE PG_CREATE_SUBSCRIPTION

- Nouveau rôle `pg_create_subscription`
  - prévenir des failles de sécurité
- Droit CREATE sur la base de données pour
  - `ALTER SUBSCRIPTION .. RENAME`
  - `ALTER SUBSCRIPTION .. OWNER TO`
- Mot de passe défini et utilisé lors de l'authentification
- Paramètre de souscription : `require_pasword`

Le rôle `pg_create_subscription` peut être donné à des utilisateurs ne bénéficiant pas de l'attribut `SUPERUSER` afin qu'ils puissent exécuter la commande `CREATE SUBSCRIPTION`. En plus de ce groupe, l'utilisateur doit avoir la permission `CREATE` sur la base de données où la souscription va être créée. Les commandes `ALTER SUBSCRIPTION .. RENAME` et `ALTER SUBSCRIPTION .. OWNER TO` nécessitent aussi ce privilège sur la base de données. Les autres versions de la commande `ALTER SUBSCRIPTION` nécessitent uniquement d'être le propriétaire de l'objet.

La raison de l'ajout de ce nouveau rôle est de prévenir des failles de sécurité où un utilisateur sans privilège pourrait exécuter du code en tant que super utilisateur en utilisant la réplication logique. L'origine du problème est liée à l'utilisation des *triggers*. Ils permettent d'exécuter du code en utilisant le *userid* de l'utilisateur qui déclenche le *trigger* plutôt que celui de l'utilisateur qui a créé le *trigger*. Un utilisateur qui a le droit de créer des tables, triggers, publications et souscriptions pourrait faire en sorte qu'un *logical replication worker* réalise des `INSERT`, `UPDATE` ou `DELETE` qui déclencheraient alors les *triggers* qui s'exécuteraient en tant que super utilisateur.

Afin de tester cette nouvelle fonctionnalité, créons une publication sur un premier serveur.

```
CREATE DATABASE tests_pg16;
\c tests_pg16 -
CREATE TABLE matable(i int);
CREATE ROLE user_pub_pg16 WITH LOGIN REPLICATION PASSWORD 'repli';
GRANT SELECT ON TABLE matable TO user_pub_pg16;
CREATE PUBLICATION pub_pg16 FOR TABLE matable;
```

On peut maintenant créer la souscription sur le serveur en version 16.

```
CREATE DATABASE tests_pg16;
CREATE ROLE sub_owner WITH LOGIN;
GRANT pg_create_subscription TO sub_owner ;
\c tests_pg16 -
GRANT CREATE, USAGE ON SCHEMA public TO sub_owner;
\c tests_pg16 sub_owner ;
CREATE TABLE matable(i int);
CREATE SUBSCRIPTION sub_pg16
       CONNECTION 'host=/var/run/postgresql port=5437 user=user_pub_pg16
 ↪  dbname=tests_pg16'
       PUBLICATION pub_pg16;
```

Comme promis, la création de la souscription est impossible sans avoir la permission CREATE sur la base `tests_pg16`.

```
ERROR:  permission denied for database tests_pg16
```

Une fois la permission donnée avec l'utilisateur **postgres**:

```
\c tests_pg16 postgres
GRANT CREATE ON DATABASE tests_pg16 TO sub_owner;
```

… une nouvelle subtilité de cette mise à jour pointe son nez.

```
ERROR:  password is required
DETAIL:  Non-superusers must provide a password in the connection string.
```

En effet, les utilisateurs ne bénéficiant pas de l'attribut SUPERUSER doivent fournir un mot de passe lors de la création de la souscription avec le mot clé `password` de la chaine de connexion.

```
\c tests_pg16 sub_owner
CREATE SUBSCRIPTION sub_pg16
       CONNECTION 'host=/var/run/postgresql port=5437 user=user_pub_pg16
 ↪  dbname=tests_pg16 password=repli'
       PUBLICATION pub_pg16;
```

Cette modification ne suffit pas, il faut également configurer la méthode d'authentification de sorte que le mot de passe soit utilisé lors de la connexion. Dans le cas contraire, on se voit gratifier du message suivant :

```
ERROR:  password is required
DETAIL:  Non-superuser cannot connect if the server does not request a password.
HINT:  Target server's authentication method must be changed, or set
 ↪  password_required=false in the subscription parameters.
```

Pour ce test, la ligne suivante doit être ajoutée au début du `pg_hba.conf` de l'instance portant la publication et la configuration rechargée :

```
local   tests_pg16     user_pub_pg16                            scram-sha-256
```

La commande précédente peut désormais s'exécuter sans erreur :

```
CREATE SUBSCRIPTION sub_pg16
       CONNECTION 'host=/var/run/postgresql port=5437 user=user_pub_pg16
 ↪  dbname=tests_pg16 password=repli'
       PUBLICATION pub_pg16;
```

```
NOTICE:  created replication slot "sub_pg16" on publisher
CREATE SUBSCRIPTION
```

Il est possible de faire en sorte qu'un utilisateur n'ayant pas l'attribut SUPERUSER soit propriétaire de la souscription sans fournir de mot de passe, en la créant avec l'attribut `password_required=false`. L'utilisation de cet attribut requiert d'être SUPERUSER.

```
\c tests_pg16 postgres
DROP SUBSCRIPTION sub_pg16;
CREATE SUBSCRIPTION sub_pg16
       CONNECTION 'host=/var/run/postgresql port=5437 user=user_pub_pg16
 ↪  dbname=tests_pg16'
       PUBLICATION pub_pg16
       WITH (password_required=false);
ALTER SUBSCRIPTION sub_pg16 OWNER TO sub_owner;
```

```
NOTICE:  created replication slot "sub_pg16" on publisher
CREATE SUBSCRIPTION
ALTER SUBSCRIPTION
```

Dans ce cas l'utilisateur **sub_owner** ne peut pas modifier la souscription :

```
\c tests_pg16 sub_owner
ALTER SUBSCRIPTION sub_pg16 DISABLE;
```

```
ERROR:  password_required=false is superuser-only
HINT:  Subscriptions with the password_required option set to false may only be
 ↪  created or modified by the superuser.
```

La seule action possible est le `DROP SUBSCRIPTION` :

```
DROP SUBSCRIPTION sub_pg16;
```

```
NOTICE:  dropped replication slot "sub_pg16" on publisher
DROP SUBSCRIPTION
```

La valeur par défaut de l'option `password_required` est `true`. Le paramétrage est ignoré pour un utilisateur bénéficiant de l'attribut SUPERUSER. Il est par conséquent possible de créer une souscription avec `password_required=true` et de la transférer à un utilisateur ne bénéficiant pas de l'attribut SUPERUSER. Dans ce cas, le comportement de la souscription est instable. Ce genre de manipulation est donc déconseillé.

4/ Performances

4.1 NOUVELLE OPTION D'EXPLAIN

- Nouvelle option `GENERIC_PLAN`
- Trace le plan générique d'une requête préparée
  - Accepte les *placeholders* comme $1 ou $2

Lors de la création d'une requête préparée, un plan générique est créé. Pendant les cinq premières exécutions, un plan personnalisé est aussi créé et les deux sont comparés pour savoir lequel est le plus intéressant. Par la suite, PostgreSQL utilisera tout le temps l'un ou l'autre, suivant lequel a été le plus intéressant pendant les cinq premières exécutions.

Ce plan générique n'était pas récupérable facilement auparavant. La version 16 ajoute une option `GENERIC_PLAN` qui permet de le récupérer. Par exemple :

```
CREATE TABLE t4(id integer);
INSERT INTO t4 SELECT generate_series(1, 1_000_000) i;
CREATE INDEX ON t4(id);

EXPLAIN (GENERIC_PLAN) SELECT * FROM t4 WHERE id<$1;

                              QUERY PLAN
---------------------------------------------------------------------------------
 Index Only Scan using t4_id_idx on t4  (cost=0.42..9493.75 rows=333333 width=4)
   Index Cond: (id < $1)
(2 rows)
```

Dans les versions précédentes, il était nécessaire d'activer les traces des requêtes pour obtenir les valeurs rattachées aux requêtes préparées à l'aide de la configuration `log_min_duration_statement`. Par exemple, pour simuler une requête préparée, nous utilisons l'outil `pgbench` et son option `--protocol=prepared`. Les traces pour une version 13 sont les suivantes :

```
LOG:  duration: 1.091 ms  parse P_0: SELECT abalance FROM pgbench_accounts WHERE aid
↪  = $1;
LOG:  duration: 1.974 ms  bind P_0: SELECT abalance FROM pgbench_accounts WHERE aid =
↪  $1;
DETAIL:  parameters: $1 = '5613613'
LOG:  duration: 0.322 ms  execute P_0: SELECT abalance FROM pgbench_accounts WHERE
↪  aid = $1;
DETAIL:  parameters: $1 = '5613613'
```

Il fallait ensuite substituer la valeur de paramètre pour obtenir le plan d'exécution :

```
EXPLAIN SELECT abalance FROM pgbench_accounts WHERE aid = 5613613;

                                        QUERY PLAN
-----------------------------------------------------------------------------------
↪  ----------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..8.45 rows=1
  ↪  width=4)
   Index Cond: (aid = 5613613)
(2 rows)
```

4.2 PLUS D'UTILISATION DU INCREMENTAL SORT

- Nœud *Incremental Sort* utilisé dans plus de cas
- Notamment pour `DISTINCT`

Le nœud *Incremental Sort* a été créé pour la version 13. Lors d'un tri de plusieurs colonnes, si la première colonne est indexée, PostgreSQL peut utiliser l'index pour réaliser rapidement un premier tri. Puis il utilise un nœud *Sort* pour trier sur les colonnes suivantes. Cela ne fonctionnait que pour les clauses `ORDER BY`.

La version 16 améliore cela en permettant son utilisation dans un plus grand nombre de cas, voici un exemple avec le cas d'une clause `DISTINCT` :

```
SET max_parallel_workers_per_gather TO 0;
DROP TABLE IF exists t1;
CREATE TABLE t1 (c1 integer, c2 integer);
INSERT INTO t1 SELECT i, i+1 FROM generate_series(1, 1000000) AS i;
VACUUM ANALYZE t1;

EXPLAIN SELECT DISTINCT c1 FROM t1;

                           QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=68175.00..85987.50 rows=1000000 width=4)
   Group Key: c1
   Planned Partitions: 16
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
(4 rows)

CREATE INDEX ON t1(c1);

EXPLAIN SELECT DISTINCT c1 FROM t1;

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

```
 Unique  (cost=0.42..28480.42 rows=1000000 width=4)
   ->  Index Only Scan using t1_c1_idx on t1
       (cost=0.42..25980.42 rows=1000000 width=4)
(2 rows)

EXPLAIN SELECT DISTINCT c1, c2 FROM t1;

                            QUERY PLAN
--------------------------------------------------------------------
 Unique  (cost=0.47..80408.43 rows=1000000 width=8)
   ->  Incremental Sort  (cost=0.47..75408.43 rows=1000000 width=8)
         Sort Key: c1, c2
         Presorted Key: c1
         ->  Index Scan using t1_c1_idx on t1
             (cost=0.42..30408.42 rows=1000000 width=8)
(5 rows)
```

En version 15, on aurait eu plutôt :

```
EXPLAIN SELECT DISTINCT c1, c2 FROM t1;

                            QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=70675.00..88487.50 rows=1000000 width=8)
   Group Key: c1, c2
   Planned Partitions: 16
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=8)
(4 rows)
```

4.3 AMÉLIORATION DES AGRÉGATS

- Pour les clauses ORDER BY et DISTINCT dans des agrégats
  - par exemple `string_agg(nom, ',' ORDER BY nom)`
- Possibilité d'utiliser
  - parcours d'index
  - tri incrémental

Certaines fonctions d'agrégat, comme `string_agg` ou `array_agg` peuvent indiquer les clauses `ORDER BY` et `DISTINCT`. Ces clauses nécessitent de trier les données et PostgreSQL a plusieurs moyens pour cela : le nœud *Sort* qui trie les données à l'exécution de la requête (et donc ralentit la récupération du résultat) et les nœuds *Index Scan* et *Index Only Scan* qui parcourent un index dans l'ordre et récupèrent donc les données pré-triées. Une clause `ORDER BY` en fin d'un `SELECT` peut utiliser ces différents nœuds. Par contre, une clause `ORDER BY` ne peut pas le faire si elle est utilisée dans un agrégat. La version 16 ajoute cette possibilité, comme le montre cet exemple :

```
create table t3(c1 integer, c2 text);
insert into t3 select i, 'ligne '||i from generate_series(1, 1000000) i;
analyze;

explain select string_agg(c2, ',' order by c2) from t3;

                              QUERY PLAN
----------------------------------------------------------------------
 Aggregate  (cost=138022.35..138022.36 rows=1 width=32)
   ->  Sort  (cost=133022.34..135522.34 rows=1000000 width=12)
         Sort Key: c2
         ->  Seq Scan on t3  (cost=0.00..16274.00 rows=1000000 width=12)
(4 rows)
```

Sans index, PostgreSQL ne peut que passer par un nœud *Sort*. Par contre, si on crée un index :

```
create index on t3(c2);

explain select string_agg(c2, ',' order by c2) from t3;
```

```
                              QUERY PLAN
--------------------------------------------------------------------
----
 Aggregate  (cost=45138.36..45138.37 rows=1 width=32)
   ->  Index Only Scan using t3_c2_idx on t3
       (cost=0.42..42638.36 rows=1000000 width=12)
(2 rows)
```

Cette fois, l'index est utilisé. Le coût estimé chûte fortement.

La version 16 permet aussi d'utiliser un tri incrémental :

```
explain select string_agg(c2, ',' order by c2,c1) from t3;
```

```
                              QUERY PLAN
--------------------------------------------------------------------
----
 Aggregate  (cost=90138.36..90138.37 rows=1 width=32)
   ->  Incremental Sort  (cost=0.48..87638.36 rows=1000000 width=16)
         Sort Key: c2, c1
         Presorted Key: c2
         ->  Index Scan using t3_c2_idx on t3
             (cost=0.42..42638.36 rows=1000000 width=16)
(5 rows)
```

Ce nouveau comportement dépend du paramètre `enable_presorted_aggregate`. En le désactivant, nous récupérons l'ancien fonctionnement.

```
show enable_presorted_aggregate;

 enable_presorted_aggregate
----------------------------
 on
(1 row)

set enable_presorted_aggregate to off;

explain select string_agg(c2, ',' order by c2,c1) from t3;
```

```
                                QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=18774.00..18774.01 rows=1 width=32)
   ->  Seq Scan on t3  (cost=0.00..16274.00 rows=1000000 width=16)
(2 rows)
```

4.4 PARALLÉLISATION DES AGRÉGATS STRING_AGG ET ARRAY_AGG

- Parallélisation possible de ces deux fonctions d'agrégat
- Comme d'habitude, un *Partial Aggregate*, suivi d'un *Full Aggregate*

Voici un exemple :

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

EXPLAIN (ANALYZE, TIMING OFF) SELECT string_agg(c2,',') FROM t1;

                                         QUERY PLAN
---------------------------------------------------------------------
----------
 Finalize Aggregate (actual time=1503.482..1503.671 rows=1 loops=1)
   ->  Gather (actual time=1450.959..1459.871 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
      ->  Partial Aggregate (actual time=1444.608..1444.618 rows=1 loops=3)
               ->  Parallel Seq Scan on t1
                   (actual time=0.023..716.970 rows=333333 loops=3)
 Planning Time: 0.110 ms
 Execution Time: 1514.484 ms
(8 rows)
```

La fonction `array_agg()` est aussi parallélisable.

4.5 PARALLÉLISATION DES FULL OUTER JOIN

- Nouveau nœud *Parallel Hash Full Join*
  - parallélisation des `FULL OUTER JOIN`
  - parallélisation des `RIGHT OUTER JOIN`
- Jointure par hachage dans ces deux cas

Voici un exemple :

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

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
  SELECT COUNT(*) FROM t1 a FULL OUTER JOIN t1 b USING (c1);

                                          QUERY PLAN
---------------------------------------------------------------------
-----------------
 Finalize Aggregate (actual rows=1 loops=1)
   ->  Gather (actual rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (actual rows=1 loops=3)
              ->  Parallel Hash Full Join (actual rows=333333 loops=3)
                     Hash Cond: (a.c1 = b.c1)
               -> Parallel Seq Scan on t1 a (actual rows=333333 loops=3)
                     ->  Parallel Hash (actual rows=333333 loops=3)
                      Buckets: 262144  Batches: 8  Memory Usage: 6976kB
                  -> Parallel Seq Scan on t1 b (actual rows=333333 loops=3)
 Planning Time: 0.300 ms
 Execution Time: 826.873 ms
(13 rows)
```

Sur les versions précédentes, le plan ressemblait à celui-ci :

```
                                  QUERY PLAN
------------------------------------------------------------------
 Aggregate (actual rows=1 loops=1)
   ->  Hash Full Join (actual rows=1000000 loops=1)
         Hash Cond: (a.c1 = b.c1)
         ->  Seq Scan on t1 a (actual rows=1000000 loops=1)
         ->  Hash (actual rows=1000000 loops=1)
               Buckets: 262144  Batches: 8  Memory Usage: 6446kB
               ->  Seq Scan on t1 b (actual rows=1000000 loops=1)
 Planning Time: 1.380 ms
 Execution Time: 1801.073 ms
(9 rows)
```

Sur cet exemple basique, nous divisons par deux le temps d'exécution, la table de test étant très peu volumineuse.

5/ Supervision

5.1 NOUVELLE VUE PG_STAT_IO

- Nouvelle vue de statistiques I/O
- Compteurs pour chaque combinaison de
  - type de backend ;
  - objet I/O cible ;
  - et contexte I/O.

La nouvelle vue `pg_stat_io` permet d'obtenir des informations sur les opérations faites sur disques. Les différents compteurs (*reads*, *writes*, *extends*, *hits*, etc) sont présents pour chaque combinaison de type de backend, objets I/O cible, et contexte I/O. Les définitions des colonnes et des compteurs peuvent être trouvées sur cette page de la documentation officielle[1].

Par exemple, la requête suivante permet d'obtenir le nombre de lectures faites par les processsus de type `client backend` (processus créé à la création d'une connexion sur l'instance), concernant des relations du type table ou index s'exécutant dans un contexte normal, c'est-à-dire des lectures et écritures utilisant les `shared_buffers`.

```
postgres=# SELECT reads
FROM pg_stat_io
WHERE backend_type = 'client backend' AND
      object = 'relation' AND
      context = 'normal';
-[ RECORD 1 ]
reads | 454
```

454 demandes de lecture ont été envoyées au noyau depuis la mise à zéro des statistiques. Cela signifie que PostgreSQL a effectué 454 demandes de lecture de blocs pour servir les données demandées par des `client backend`.

Si maintenant un `SELECT` est exécuté, le compteur peut augmenter si des données sont demandées au noyau.

```
postgres=# select * from pgbench_accounts ;
[...]
postgres=# SELECT reads
FROM pg_stat_io
```

[1] https://www.postgresql.org/docs/devel/monitoring-stats.html#MONITORING-PG-STAT-IO-VIEW

```
WHERE backend_type = 'client backend' AND
      object = 'relation' AND
      context = 'normal';
-[ RECORD 1 ]
reads | 454
```

Le compteur n'a pas évolué. Les données étaient bien dans le cache de PostgreSQL, aucune demande n'a été envoyé au noyau. Essayons avec un autre table :

```
postgres=# select * from pgbench_accounts ;
[...]
postgres=# SELECT reads
FROM pg_stat_io
WHERE backend_type = 'client backend' AND
      object = 'relation' AND
      context = 'normal';
-[ RECORD 1 ]
reads | 456
[...]
```

Le compteur a évolué. Deux demandes de lecture ont été faites au noyau pour ramener des données.

Une analyse de la vue `pg_stat_io` permettra d'extraire des explications sur le fonctionnement et la santé de l'instance. Par exemple :

- Un compteur `reads` élevé laisse penser que le paramètre `shared_buffers` est trop petit ;
- Un compteur `writes` plus élevé pour les `client backend` que pour le `background writer` laisse penser que les écritures en arrière plan ne sont pas correctement configurées.

5.2 HORODATAGE DU DERNIER PARCOURS D'UNE RELATION

- Donne la date et heure du dernier parcours de table et d'index
- Ajout de deux colonnes pour pg_stat_all_tables
  - last_seq_scan, dernier parcours séquentiel de table
  - last_idx_scan, dernier parcours d'index
- Ajout d'une colonne pour pg_stat_all_indexes
  - last_idx_scan, dernier parcours d'index

Avant cette version, il était possible de connaître le nombre total de parcours séquentiel et de parcours d'index par table, ainsi que le nombre total de parcours d'index pour chaque index. Ces informations sont intéressantes mais pas suffisantes.

En effet, si le nombre de parcours d'index est de 0, nous savons qu'il n'a jamais été utilisé depuis sa création ou depuis la dernière réinitialisation des statistiques de l'index. Cependant, s'il vaut, par exemple, 200, il est impossible de savoir quand ces 200 lectures ont eu lieu. Et notamment, il est impossible de savoir si la dernière lecture date d'hier ou d'il y a 3 ans. Dans ce dernier cas, la suppression de l'index serait correctement motivée.

Les développeurs de PostgreSQL ont donc ajouté deux colonnes dans la vue `pg_stat_all_tables` pour connaître la date et heure du dernier parcours de table (colonne `last_seq_scan`) et la date et heure du dernier parcours d'index pour cette table (colonne `last_idx_scan`).

La vue `pg_stat_all_indexes` contient elle aussi une colonne `last_idx_scan`.

Voici un exemple complet :

```
CREATE TABLE t1(id integer);

SELECT relname, seq_scan, last_seq_scan, idx_scan, last_idx_scan
FROM pg_stat_user_tables
WHERE relname='t1' \gx

-[ RECORD 1 ]--------------------------------
relname       | t1
seq_scan      | 0
last_seq_scan |
idx_scan      |
```

```
last_idx_scan |

SELECT * FROM t1;

 id
----
(0 rows)

SELECT relname, seq_scan, last_seq_scan, idx_scan, last_idx_scan
FROM pg_stat_user_tables
WHERE relname='t1' \gx

-[ RECORD 1 ]--------------------------------
relname       | t1
seq_scan      | 1
last_seq_scan | 2023-08-10 15:51:58.199368+02
idx_scan      |
last_idx_scan |

INSERT INTO t1 SELECT generate_series(1, 1000000);
CREATE INDEX ON t1(id);

SELECT * FROM t1 WHERE id=1;
 id
----
  1
(1 row)

SELECT relname, seq_scan, last_seq_scan, idx_scan, last_idx_scan
FROM pg_stat_user_tables
WHERE relname='t1' \gx

-[ RECORD 1 ]--------------------------------
relname       | t1
seq_scan      | 2
last_seq_scan | 2023-08-10 15:52:12.243123+02
idx_scan      | 1
last_idx_scan | 2023-08-10 15:52:18.068182+02
```

5.3 NOMBRE D'UPDATE

- Indique le nombre de lignes déplacées dans un autre bloc suite à une mise à jour
- Ajout d'une colonne pour pg_stat_all_tables
  - `n_tup_newpage_upd`
- Permet d'estimer les bons candidats à la configuration du `fillfactor`

Lors d'un UPDATE, PostgreSQL va dupliquer la ligne qui doit être modifiée. L'ancienne version est simplement indiquée comme morte, la nouvelle est modifiée. Cette nouvelle ligne sera enregistrée dans le même bloc que l'ancienne si l'espace y est suffisant. Sinon elle ira dans un autre bloc, ancien ou nouveau suivant la place disponible dans les blocs existants.

Auparavant, il était possible de connaître le nombre de lignes mises à jour ainsi que le nombre de lignes mises à jour dans le même bloc. Cependant, aucune colonne n'indiquait le nombre de lignes mises à jour dans un autre bloc. Ceci arrive en version 16 avec la nouvelle colonne `n_tup_newpage_upd` de la vue `pg_stat_all_tables`.

L'intérêt est que, si cette colonne augmente fortement, il y a de fortes chances que la table en question puisse bénéficier d'une configuration à la baisse du paramètre `fillfactor`.

Par exemple, voici une table de 1000 lignes. Nous désactivons l'autovacuum pour cette table, histoire qu'il ne nettoie pas la table automatiquement, et nous nous assurons d'avoir un facteur de remplissage à 100%.

```
-- preparation
drop table if exists t1;
create table t1(id integer);
alter table t1 set (autovacuum_enabled=false);
alter table t1 set (fillfactor = 100);
insert into t1 select generate_series(1, 1000);
select n_tup_ins, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd
  from pg_stat_user_tables
  where relname='t1';
```

n_tup_ins	n_tup_upd	n_tup_hot_upd	n_tup_newpage_upd
1000	0	0	0

Les statistiques indiquent bien les 1000 lignes insérées et aucune ligne mise à jour.

Faisons un premier UPDATE d'une ligne :

```
-- test #1
select ctid from t1 where id=1;
```

ctid
(0,1)

```
update t1 set id=id where id=1;
select ctid from t1 where id=1;
```

ctid
(4,97)

```
select n_tup_ins, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd
  from pg_stat_user_tables
  where relname='t1';
```

n_tup_ins	n_tup_upd	n_tup_hot_upd	n_tup_newpage_upd
1000	1	0	1

Comme cette table n'a pas de fragmentation et comme nous modifions la première ligne, cette nouvelle ligne va se retrouver en fin de fichier. Elle change donc de bloc. Le CTID l'indique bien (passage du bloc 0 au bloc 4). La nouvelle colonne de statistique `n_tup_newpage_upd` est bien mise à jour.

Modifions de nouveau la même ligne :

```
-- test #2
```

```
select ctid from t1 where id=1;
```

ctid
(4,97)

```
update t1 set id=id where id=1;
select ctid from t1 where id=1;
```

ctid
(4,98)

```
select n_tup_ins, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd
  from pg_stat_user_tables
  where relname='t1';
```

n_tup_ins	n_tup_upd	n_tup_hot_upd	n_tup_newpage_upd
1000	2	1	1

La nouvelle version de la ligne est ajoutée toujours en fin de fichier (pas de VACUUM entre les deux) mais il se trouve qu'il s'agit cette fois du même bloc. C'est donc l'ancien champ des statistiques qui est incrémenté. Nous voyons donc bien les deux informations séparément.

5.4 AMÉLIORATION DE PG_STAT_STATEMENTS

- Normalise la requête indiquée dans les ordres :
  - DECLARE
  - EXPLAIN
  - CREATE MATERIALIZED VIEW
  - CREATE TABLE AS
- Par exemple

```
CREATE TABLE pgss_ctas AS SELECT a, $1 b FROM generate_series($2, $3) a;
DECLARE cursor_stats_1 CURSOR WITH HOLD FOR SELECT $1;
```

Avant cette version, aucune requête DDL n'était normalisée. Avec la version 16, les ordres qui intègrent des requêtes SQL (comme la déclaration d'un curseur, la récupération d'un plan d'exécution, etc) sont aussi normalisés.

La requête :

```
CREATE TABLE pgss_ctas AS SELECT a, 'ctas' b FROM generate_series(1, 10) a;
```

devient donc

```
CREATE TABLE pgss_ctas AS SELECT a, $1 b FROM generate_series($2, $3) a;
```

5.5 AMÉLIORATION DE AUTO_EXPLAIN

- Trace le queryid en mode VERBOSE
- Gère le paramètre log_parameter_max_length

Bien que l'identifiant de requête soit disponible dans une commande EXPLAIN manuelle, il n'était pas récupéré par le module auto_explain. Il le fait à partir de la version 16. Par exemple :

```
2023-09-29 18:44:40.229 CEST [136206] LOG:  duration: 0.029 ms  plan:
        Query Text: select pg_is_in_recovery() as ro
        Result  (cost=0.00..0.01 rows=1 width=1)
          Output: pg_is_in_recovery()
        Query Identifier: 6937149974915068530
```

(Ne pas oublier que pour avoir cet identifiant, il faut soit avoir installé le module pg_stat_statements soit avoir configuré le paramètre query_compute_id à on.)

auto_explain dispose d'un nouveau paramètre, auto_explain.log_parameter_max_length, qui est à l'image du paramètre log_parameter_max_length, ajouté lui en version 15. Ce paramètre permet d'indiquer si le module doit tracer les arguments d'une requête à paramètres (par exemple une requête préparée). La valeur -1 permet de tracer toutes les requêtes, alors que la valeur 0 désactive cette trace. Les valeurs supérieures à zéro indiquent la longueur maximale des valeurs tracées.

6/ Régression

6.1 DISPARITION DES VARIABLES LC_COLLATE ET LC_CTYPE

- Suppression des variables en lecture seule :
  - `lc_collate`
  - `lc_ctype`

PostgreSQL 16 supprime ces deux variables. À l'origine valables pour l'instance toute entière, elles sont devenues locales à chaque base de données avec la sortie de la version 8.4. Rendues uniquement consultables, elles peuvent même porter à confusion étant donné que la valeur définie n'est pas nécessairement appliquée aux bases de l'instance.

Le message d'erreur suivant apparaitra lors d'une tentative de consultation :

```
psql (16.1)
Type "help" for help.

postgres=# show lc_collate ;
ERROR:  unrecognized configuration parameter "lc_collate"
postgres=# show lc_ctype;
ERROR:  unrecognized configuration parameter "lc_ctype"
```

7/ Autres régressions

- Paramètres supprimés
  - vacuum_defer_cleanup_age
  - promote_trigger_file
- Paramètre renommé
  - force_parallel_mode devient debug_parallel_query

PostgreSQL 16 supprime deux paramètres qui sont devenus inutiles. Dus aux récents changements sur la commande VACUUM, le paramètre vacuum_defer_cleanup_age est devenu inutile.

Le paramètre promote_trigger_file permettait d'indiquer le nom d'un fichier dont la présence demandait à une instance PostgreSQL secondaire de quitter le mode lecture seule et l'application de la réplication. Il existe deux autres moyens de le faire (un via le shell, un autre via une commande SQL), un paramètre comme celui-ci n'était donc pas vraiment utile.

Quant au paramètre renommé, il a été considéré qu'il fallait mettre l'accent sur le fait qu'il s'agit d'un paramètre de débogage, pas d'un paramètre d'utilisation normale. L'ajout du mot debug dans le nom du paramètre aide à cela.

7.1 QUESTIONS ?

Merci de votre écoute !

Nouveautés de la version 16 :

https://dali.bo/workshop16_html

https://dali.bo/workshop16_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.

8/ 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 fran- cophone 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.