Workshop 15

Nouveautés de PostgreSQL 15

15

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 15

Workshop 15

TITRE : Nouveautés de PostgreSQL 15
SOUS-TITRE : Workshop 15

REVISION: 15

Table des Matières

1 ADMINISTRATION

1.1 FONCTIONNEMENT INTERNE

1.1.1 LANCEMENT DU BACKGROUND WRITER ET DU CHECKPOINTER LORS D'UNE RÉCUPÉRATION SUITE À UN CRASH

- Les processus checkpointer et bgwriter sont lancés dès la phase de *crash recovery*
  - simplifier le code en limitant la duplication
  - améliorer les performances dans certains cas

Le checkpointer et le bgwriter son désormais lancés pendant la phase de *crash recovery* de la même manière qu'on le fait pour la réplication. L'objectif est de limiter la duplication de code en supprimant ce cas particulier. Il est possible que, dans certains cas, cela améliore les performances. Par exemple quand la quantité de données à mettre en cache pour la *recovery* dépasse la taille des *shared buffers*.

Le comportement de l'instance reste inchangé en mode *single user* (option `--single` du *postmaster*).

1.1.2 PLUS DE CHECKPOINT LORS DE LA CRÉATION D'UNE DATABASE

- `CREATE DATABASE .. STRATEGY WAL_LOG` (valeur par défaut)
  - opération entièrement tracée dans les WAL
  - évite deux checkpoints potentiellement impactant pour les performances
  - manipulation plus sécurisée à la fois sur l'instance primaire et les instances qui rejouent les WAL par la suite, notamment les instances secondaires
- `CREATE DATABASE .. STRATEGY FILE_COPY`
  - méthode historique
  - génère moins de WAL
  - plus rapide quand la base modèle est très grosse

Précédemment, lors de la création d'une base de données, PostgreSQL devait réaliser un checkpoint, copier les fichiers de la base de référence, puis faire un nouveau checkpoint.

Le premier checkpoint permet de s'assurer que les données des buffers sales sont sur écrits sur disque, y compris ceux des tables *UNLOGGED*. Il permet aussi de s'assurer que les commandes de suppressions de fichiers ont été traitées, ce qui évite la disparition d'un fichier pendant sa copie.

La copie des fichiers de la base de référence est tracée dans les WAL sous forme d'un enregistrement unique pour chaque `TABLESPACE` utilisé par la base. Chacun de ces enregistrements représente l'écriture du répertoire associé.

Le second checkpoint permet de s'assurer qu'on ne rejouera pas les enregistrements de WAL du `CREATE DATABASE` en cas de *crash recovery*. La copie des fichiers pourrait en effet produire un résultat différent car des modifications ont été faites après la copie mais avant la fin des WAL. Cela causerait des erreurs dans le rejeu des enregistrements de WAL suivants.

Un nouveau mécanisme a été mis en place pour permettre de réaliser le `CREATE DATABASE` sans checkpoint.

Ce changement a plusieurs avantages :

- éviter deux checkpoints qui peuvent être très coûteux en performance à la fois pendant le checkpoint et après si `full_page_writes` est configuré à `on` (ce qui est la valeur par défaut). Ce problème peut arriver sur des systèmes avec une grosse activité. La nouvelle méthode permet également d'améliorer les performances de la commande lorsque la base de référence est petite ;
- sécuriser la copie en listant les fichiers copiés à partir des informations présentes dans le catalogue au lieu de se baser sur le contenu du système de fichier. Cela permet d'éviter de copier des fichiers qui ne devraient pas être là ;
- sécuriser le rejeu des WAL en rendant l'opération plus robuste. Les données copiées sont toutes tracées dans les WAL au lieu de n'enregistrer qu'un marqueur qui signale qu'il faut copier le répertoire de la base modèle ;
- permettre plus de flexibilité pour développer d'autres fonctionnalités, par exemple TDE (*Transparent Data Encryption*).

Ce changement augmente cependant la volumétrie de WAL écrits, cela peut être un problème dans certains cas. De plus, si la base est grosse, la copie de fichier est plus performante.

Le changement n'étant pas sans pénalité, le choix de la stratégie de création est laissé à l'utilisateur. `CREATE DATABASE` se voit donc ajouter un paramètre supplémentaire, appelé `STRATEGY`, qui peut prendre la valeur `WAL_LOG` (valeur par défaut) ou `FILE_COPY`.

La différence de volume de WAL généré par chaque commande est facilement observable dans la vue `pg_stat_wal`.

Cas d'une création entièrement tracée dans les WAL :

```
SELECT pg_stat_reset_shared('wal');
CREATE DATABASE db_wal_log STRATEGY WAL_LOG;
SELECT wal_records, wal_bytes FROM pg_stat_wal;
```

```
 wal_records | wal_bytes
-------------+-----------
        1254 |   4519307
(1 row)
```

Cas d'une création en mode copie de fichier :

```
SELECT pg_stat_reset_shared('wal');
CREATE DATABASE db_file_copy STRATEGY FILE_COPY;
SELECT wal_records, wal_bytes FROM pg_stat_wal ;
```

```
 wal_records | wal_bytes
-------------+-----------
          11 |       849
(1 row)
```

La commande `ALTER DATABASE .. SET TABLESPACE` repose sur les mêmes mécanismes que `CREATE DATABASE .. STRATEGY FILE_COPY`. L'opération était initialement couverte par cette évolution mais la commande est plus complexe à modifier et le travail n'a pas pu être fait dans les temps pour la sortie de la version 15.

1.1.3 STATISTIQUES D'ACTIVITÉ EN MÉMOIRE PARTAGÉE

- Statistiques d'activité stockées en mémoire
- Données perdues en cas de crash.
- Disparition du processus `stats collector`
- Disparition du paramètre `stats_temp_directory`
- Nouveau paramètre `stats_fetch_consistency`

Dans les versions précédentes, le processus `stats collector` recevait des mises à jour des statistiques d'activité collectées par les autres processus via UDP. Il partageait ces statistiques en les écrivant à intervalle régulier dans des fichiers temporaires situés dans le répertoire pointé par `stats_temp_directory`. Ces fichiers pouvaient atteindre quelques dizaines de mégaoctets et être écrits jusqu'à deux fois par seconde, ce goulet d'étranglement a longtemps été un frein à l'ajout de statistiques pourtant utiles.

Désormais, les statistiques sont stockées en mémoire partagée, soit directement si le nombre de ces statistiques est fixe (`pg_stat_database`), soit dans une table de hachage dans le cas où leur nombre est variable (`pg_stat_users_tables`).

Une zone dédiée est désormais visible dans la mémoire partagée :

```
SELECT *
  FROM pg_shmem_allocations
 WHERE name = 'Shared Memory Stats';
```

```
        name          |    off    |  size  | allocated_size
----------------------+-----------+--------+----------------
 Shared Memory Stats | 147290112 | 263312 |         263424
(1 row)
```

Ce changement d'architecture se traduit par la disparition du processus `stats collector` et du paramètre `stats_temp_directory`. Le répertoire `pg_stat_tmp` existe toujours dans le répertoire de données de l'instance car certaines extensions dont `pg_stat_statements` l'utilisent toujours.

Les statistiques sont chargées depuis des fichiers situés dans le répertoire `pg_stat` lors du démarrage. Elles sont écrites lors de l'arrêt de l'instance par le processus `checkpointer`. Il y a deux exceptions à cela : en cas de crash les statistiques sont remises à zéro et en cas d'arrêt avec l'option `immediate` les données ne sont pas sauvées.

Les données sont la plupart du temps accumulées localement par les processus avant d'être écrites suite à un commit ou lors d'un timeout causé par l'inactivité de la session.

Un nouveau paramètre a été introduit : `stats_fetch_consistency`. Il permet de déterminer le comportement lorsque les statistiques sont accédées dans une transaction. Il peut être changé dans la session et a trois valeurs possibles :

- `none` : chaque accès récupère les données depuis la mémoire partagée. Les valeurs ramenées sont donc différentes à chaque fois. C'est le mode le moins coûteux. Il est adapté pour les systèmes de supervision qui accèdent aux données une seule fois ;
- `cache` : le premier accès à une statistique la met en cache pour le restant de la transaction à moins que `pg_stat_clear_snapshot()` ne soit appelée. C'est utile pour des requêtes qui font des auto-jointures. C'est la valeur par défaut ;
- `snapshot` : le premier accès aux statistiques met en cache toutes les statistiques accessibles pour la base de données en cours. Elles seront conservées jusqu'à la fin de la transaction à moins que `pg_stat_clear_snapshot()` ne soit appelée. C'est le mode le plus coûteux.

On observe quatre contextes mémoires relatifs aux statistiques d'activité pour ces trois modes de fonctionnement :

```
SELECT name, parent, level, total_bytes, free_bytes, used_bytes
  FROM pg_backend_memory_contexts
 WHERE name LIKE ANY(ARRAY['PgStat%','CacheMemoryContext']);
```

```
          name           |       parent       | level | total_bytes | free_bytes | used_bytes
-------------------------+--------------------+-------+-------------+------------+------------
 CacheMemoryContext      | TopMemoryContext   |     1 |     1048576 |     501176 |     547400
 PgStat Shared Ref Hash | CacheMemoryContext |     2 |        7224 |        680 |       6544
 PgStat Shared Ref       | CacheMemoryContext |     2 |        8192 |       3568 |       4624
```

```
 PgStat Pending          | CacheMemoryContext |     2 |        8192 |       6944 |       1248
(4 rows)
```

On peut noter la présence de la zone mémoire `PgStat Pending` qui est celle utilisée pour les statistiques en cours de mise à jour dans la session.

Avec les modes `cache` et `snapshot`, on voit l'apparition d'une autre zone. Sa dimension est importante dans le cas du mode `snapshot`.

```
  mode    |         name           |       parent       | level | total_bytes | free_bytes | used_b
----------+------------------------+--------------------+-------+-------------+------------+-------
snapshot | PgStat Snapshot         | TopMemoryContext   |     1 |       57400 |       4488 |      5
cache    | PgStat Snapshot         | TopMemoryContext   |     1 |       25656 |        680 |      2
```

Les vues `pg_backend_memory_contexts`, `pg_shmem_allocations` sont normalement accessibles uniquement aux utilisateurs dotés de l'attribut super utilisateur. En version 15, les membres du groupe `pg_read_all_stats` peuvent aussi y accéder.

1.1.4 PRÉSERVATION DE L'OID DES RELFILENODES, TABLESPACES, ET BASES DE DONNÉES APRÈS UNE MIGRATION PG_UPGRADE

- `pg_upgrade` préserve désormais :
  - les relfilenode
  - les oid de tablespaces
  - les oid de base de données
- pour :
  - faciliter les analyses post upgrade
  - économiser de la bande passante quand on resynchronise une instance post upgrade avec rsync

`pg_upgrade` préserve désormais les *relfilenodes*, *tablespace oid* et *database oid*.

Le relfilenode est le nom utilisé par le fichier qui contient les données d'une relation. Les différents segments et forks de la relation ajoutent un suffixe au relfilenode (ex: `_vm` pour la *visibility map*). Il peut être différent de l'oid de l'objet (identifiant unique d'un objet) car certaines opérations peuvent conduire à la recréation des fichiers de la relation comme un `VACUUM FULL`.

Le *tablespace oid* est l'identifiant unique d'un tablespace. Il est utilisé pour le lien symbolique placé dans le répertoire `pg_tblspc` et qui pointe vers le répertoire qui contient les données du tablespace.

Le *database oid* est l'identifiant unique d'une base de données. Il est utilisé pour nommer le répertoire qui regroupe toutes les données d'une base de données dans un tablespace.

Ce changement permet donc limiter les changements de noms de fichiers, répertoires et lien symboliques suite à une montée de version avec `pg_upgrade`. Les bénéfices sont multiples :

- faciliter l'analyse en cas de problème lors de la mise à jour ;
- économiser de la bande passante quand on utilise rsync pour faire une mise à jour différentielle des fichiers d'une instance après une mise à jour ;
- faciliter l'implémentation de futures fonctionnalités comme le chiffrement des blocs pour lesquels le sel pourrait se baser sur le refilenode.

Pour permettre cette modification, il a été décidé que l'oid des bases système serait fixé et que les bases de données utilisateurs auront un oid supérieur ou égale à `16384`.

```
SELECT datname, oid FROM pg_database
```

```
   datname    |  oid
--------------+-------
 formation    | 16384
 postgres     |     5
 template1    |     1
 template0    |     4
(4 rows)
```

La commande `CREATE DATABASE` se voit ajouter une nouvelle clause `OID` qui permet de spécifier manuellement un oid. Cet ajout est principalement destiné à l'usage de `pg_upgrade` qui est par ailleurs le seul à pouvoir assigner des oid inférieurs à `16384`.

1.2 PSQL

1.2.1 OPTIMISATION DES PERFORMANCES DE LA COMMANDE `\COPY`

- optimisation de la méta-commande psql `\copy from`

L'utilisation de plus larges segments de données par la commande psql `\copy from` permet d'effectuer plus rapidement l'import de données dans des tables.

Le gain observé approche les 10% sur un fichier de données contenant 20 millions d'entrées.

PostgreSQL 14 :

```
postgres=# \copy t1 from '~/data.txt';
COPY 20000000
Time: 9755.384 ms (00:09.755)
```

PostgreSQL 15 :

```
postgres=# \copy t1 from '~/data.txt';
COPY 20000000
Time: 8920.834 ms (00:08.921)
postgres=#
```

Si ce transfert passe par une connexion distante, la quantité de trafic réseau est également réduite.

1.2.2 NOUVELLES COMMANDES `\GETENV` ET `\DCONFIG`

- Ajout de nouvelles méta-commandes `psql`
- commande `\dconfig` pour afficher la configuration de l'instance
- commande `\getenv` pour récupérer la valeur d'une variable d'environnement

Commande `\dconfig`

La commande `\dconfig` permet d'afficher les paramètres de configuration de l'instance.

Son appel sans argument permet d'afficher les paramètres dont les valeurs ne sont pas celles par défaut :

```
postgres=# \dconfig
     List of non-default configuration parameters
         Parameter          |          Value
----------------------------+-------------------------
 application_name           | psql
```

```
 client_encoding            | UTF8
 config_file                | /data/15/postgresql.conf
 data_directory             | /data/15
 default_text_search_config | pg_catalog.english
 hba_file                   | /data/15/pg_hba.conf
 ident_file                 | /data/15/pg_ident.conf
 lc_messages                | en_US.UTF-8
 lc_monetary                | en_US.UTF-8
 lc_numeric                 | en_US.UTF-8
 lc_time                    | en_US.UTF-8
 log_filename               | postgresql-%a.log
 logging_collector          | on
 log_rotation_size          | 0
 log_timezone               | UTC
 log_truncate_on_rotation   | on
 TimeZone                   | UTC
```

L'ajout d'un + à la commande permet d'obtenir plus d'informations :

```
postgres=# \dconfig+
                        List of non-default configuration parameters
         Parameter          |          Value           |  Type   |  Context   | Access privileges
----------------------------+--------------------------+---------+------------+-------------------
 application_name           | psql                     | string  | user       |
 client_encoding            | UTF8                     | string  | user       |
 config_file                | /data/15/postgresql.conf | string  | postmaster |
 data_directory             | /data/15                 | string  | postmaster |
 default_text_search_config | pg_catalog.english       | string  | user       |
 hba_file                   | /data/15/pg_hba.conf     | string  | postmaster |
 ident_file                 | /data/15/pg_ident.conf   | string  | postmaster |
 lc_messages                | en_US.UTF-8              | string  | superuser  |
 lc_monetary                | en_US.UTF-8              | string  | user       |
 lc_numeric                 | en_US.UTF-8              | string  | user       |
 lc_time                    | en_US.UTF-8              | string  | user       |
 log_filename               | postgresql-%a.log        | string  | sighup     |
 logging_collector          | on                       | bool    | postmaster |
 log_rotation_size          | 0                        | integer | sighup     |
 log_timezone               | UTC                      | string  | sighup     |
 log_truncate_on_rotation   | on                       | bool    | sighup     |
 TimeZone                   | UTC                      | string  | user       |
```

La commande accepte également l'utilisation de *wild card* :

```
postgres=# \dconfig *work_mem*
 List of configuration parameters
         Parameter          | Value
----------------------------+-------
 autovacuum_work_mem        | -1
```

```
logical_decoding_work_mem | 64MB
maintenance_work_mem      | 64MB
work_mem                  | 4MB
```

> l'appel `\dconfig *` permet ainsi de lister l'ensemble des paramètres de l'instance.

Commande `\getenv`

La commande `\getenv` permet d'enregistrer la valeur d'une variable d'environnement dans une variable sql.

```
[postgres@pg15 ~]$ export ENV_VAR='foo'
[postgres@pg15 ~]$ psql
psql (15beta2)
Type "help" for help.

postgres=# \getenv sql_var ENV_VAR

postgres=# \echo :sql_var
foo
```

1.2.3 DIVERSES AMÉLIORATIONS SUR L'AUTO-COMPLÉTION

- Recherche insensible à la casse
- Affichage des noms complets des commandes plutôt que leurs abréviations
- Amélioration de l'auto-complétion de différentes commandes *SQL* :
  - `EXPLAIN EXECUTE`
  - `LOCK TABLE ONLY | NOWAIT`
  - `ALTER TABLE ... ADD`
  - `CREATE`, `ALTER`, `DROP`

L'auto-complétion dans `psql` a été améliorée à différents niveaux.

Recherche insensible à la casse

L'auto-complétion est désormais capable de suggérer ou compléter une commande même si la casse n'est pas respectée.

La complétion par une double tabulation de la saisie suivante permet d'afficher la liste des paramètres des traces, alors que les versions précédentes ne renvoyait rien :

```
postgres=# set LOG_
```

La saisie est automatiquement transformée en minuscule, et les différentes suggestions apparaissent :

```
postgres=# set log_
log_duration                    log_lock_waits                  log_min_messages
log_error_verbosity             log_min_duration_sample         log_parameter_max_length
log_executor_stats              log_min_duration_statement      log_parameter_max_length_on_
logical_decoding_work_mem       log_min_error_statement         log_parser_stats
```

Noms de paramètres

La complétion d'un `\` via une double tabulation permet de lister les commandes disponibles. Cette liste affiche désormais le nom complet de chaque commande, alors que certaines commandes apparaissaient sous la forme d'abréviations. La commande `\l` devient ainsi `\list`, `\o` devient `\out`, `\e` devient `\echo`, etc.

```
postgres=# \
Display all 106 possibilities? (y or n)
\!                \dAp              \dFp              \dRs              \errverbose       \lo
\?                \db               \dFt              \ds               \ev               \lo
\a                \dc               \dg               \dt               \f                \lo
\C                \dC               \di               \dT               \g                \lo
\cd               \dconfig          \dl               \du               \gdesc            \ou
\connect          \dd               \dL               \dv               \getenv           \pa
\conninfo         \dD               \dm               \dx               \gexec            \pr
\copy             \ddp              \dn               \dX               \gset             \pr
\copyright        \dE               \do               \dy               \gx               \ps
\crosstabview     \des              \dO               \echo             \help             \qe
\d                \det              \dp               \edit             \html             \qu
\da               \deu              \dP               \ef               \if               \re
\dA               \dew              \dPi              \elif             \include          \s
\dAc              \df               \dPt              \else             \include_relative \se
\dAf              \dF               \drds             \encoding         \ir               \se
\dAo              \dFd              \dRp              \endif            \list             \sf
```

EXPLAIN EXECUTE

La complétion de la commande `EXPLAIN` ajoute l'option `EXECUTE`.

```
postgres=# EXPLAIN
ANALYZE      DECLARE      DELETE FROM  EXECUTE      INSERT INTO  MERGE        SELECT       UPDATE
```

LOCK TABLE

La commande LOCK TABLE permet désormais la complétion de l'option `ONLY` avant le nom de la table :

```
postgres=# LOCK TABLE
information_schema.  ONLY                 public.              t1
```

Idem pour l'option NOWAIT, à préciser après le nom de la table :

```
postgres=# LOCK TABLE t1
IN      NOWAIT
```

CREATE, ALTER, DROP

Enfin, diverses améliorations ont été apportées aux options de complétion de plusieurs commandes CREATE, ALTER et DROP :

- CREATE CONVERSION, CREATE DOMAIN, CREATE LANGUAGE, CREATE SCHEMA, CREATE SEQUENCE, CREATE TRANSFORM
- ALTER DEFAULT PRIVILEGES,ALTER FOREIGN DATA WRAPPER, ALTER SEQUENCE, ALTER VIEW
- DROP MATERIALIZED VIEW, DROP OWNED BY, DROP POLICY, DROP TRANSFORM

1.3 SAUVEGARDE ET RESTAURATION

1.3.1 FIN DES BACKUPS EXCLUSIFS

- le mode *backup exclusive*:
  - risqué en cas de crash de l'instance
  - déprécié depuis la version 9.6
  - supprimé depuis la version 15
- renommage des fonctions de *backup* :
  - `pg_start_backup()` devient `pg_backup_start()`
  - `pg_stop_backup()` devient `pg_backup_stop()`

Le mode *backup_exclusive* pose problème car il crée un fichier `backup_label` dans le répertoire de données durant l'execution d'une sauvegarde. Avec ce mode, il n'y a aucun moyen de distinguer le répertoire de données d'un serveur en mode sauvegarde de celui d'un serveur interrompu pendant la sauvegarde. En cas de crash, l'instance cherche alors à se restaurer au lieu de poursuivre la sauvegarde inachevée.

En essayant de se restaurer sans `restore_command`, PostgreSQL cherche à rejouer les journaux disponibles dans *pg_wal* et peut échouer si une activité importante a entraîné la rotation desdits journaux. Dans certains cas, si le *checkpoint* écrit dans `backup_label` n'est pas bon, ou si le fichier lui-même n'est pas bon, l'instance tente de revenir à un état différent de celui précédent le backup, entraînant un risque de corruption des données.

Avec le mode de sauvegarde non exclusif, le fichier *backup_label* est renvoyé par la fonction `pg_backup_stop` au lieu d'être écrit dans le répertoire de données, protégeant ainsi le serveur en cas de crash pendant une sauvegarde. Une connexion avec le client de sauvegarde est nécessaire pendant toute la durée de celle-ci. En cas d'interruption, l'opération est abandonnée, sans risque pour l'intégrité des données. Ce mode de sauvegarde a été introduit avec PostgreSQL 9.6 et a supplanté le mode exclusif qui a été déprécié dans la foulée. Cependant, les sauvegardes exclusives ont continué à être présentes et utilisables jusqu'à la version 14.

PostgreSQL 15 supprime cette possibilité, et pour éviter toute confusion, les fonctions `pg_start_backup()` et `pg_stop_backup()`ont été renommées `pg_backup_start()` et `pg_backup_stop()`. Il est donc impératif de contrôler ses procédures de sauvegardes pour, d'une part, vérifier qu'elles n'utilisent plus les sauvegardes exclusives, et d'autre part, renommer les fonctions d'appel.

1.3.2 ARCHIVE_LIBRARY & MODULE "BASIC ARCHIVE"

- Option de remplacement pour l'`archive_command`
- Nouveau paramètre `archive_library`
- Module `basic_archive` :
  - `basic_archive.archive_directory`

Il est désormais possible d'utiliser des modules pour l'archivage plutôt que l'`archive_command`. Cela simplifie la mise en place de l'archivage, permet de rendre cette opération plus sécurisée et robuste, et aussi plus performante. Ces modules peuvent accéder à des fonctionnalités avancées de PostgreSQL comme la création des paramètres configuration ou de *background workers*.

On peut s'attendre à un gain de performance dû au fait que, plutôt de créer un processus pour l'exécution de chaque `archive_command`, PostgreSQL va utiliser le module qui a été chargé en mémoire une fois pour toutes.

On peut imaginer que des gains similaires pourront être fait pour l'établissement d'une connexion à un serveur distant. Il pourrait également être possible d'invoquer des *background workers* en réaction à une accumulation de WAL en attente d'archivage.

L'écriture d'un module d'archivage est décrite dans la documentation. Il faut pour cela écrire un programme en C, en plus de requérir des compétences particulières, les chances de planter le serveur sont grandes en cas de bug. Il semble donc plus raisonnable de s'appuyer et participer à des projets communautaires. Les outils de sauvegardes comme pgBackRest ou Barman vont sans doute également s'emparer du sujet.

Un nouveau paramètre `archive_library` a été ajouté à la configuration et permet de charger le module. Comme pour l'`archive_command`, le serveur n'effectuera la suppression ou le recyclage des WAL que lorsque le module indique que les WAL ont été archivés. Ce nouveau paramètre peut être rechargé à chaud.

```
=# \dconfig+ archive_library
                List of configuration parameters
    Parameter    | Value |  Type  | Context | Access privileges
-----------------+-------+--------+---------+-------------------
 archive_library |       | string | sighup  |
(1 row)
```

Si l'`archive_library` n'est pas remplie, PostgreSQL utilisera l'`archive_command`. En version 15, si les deux paramètres sont remplis, PostgreSQL favorisera l'`archive_library`. Ce comportement va changer en v16 ou les deux paramètres ne pourront pas être définis en même temps.

Le module d'exemple[1] `basic_archive` a également été mis à disposition pour tester la fonctionnalité et donner un exemple d'implémentation pour ce genre de module. Pour l'utiliser, il suffit d'activer l'archivage, d'ajouter le module à l'archive_library et de configurer le répertoire cible pour l'archivage. Il faut ensuite redémarrer l'instance.

Afin d'observer le fonctionnement de ce module, nous allons créer une instance neuve :

```
ARCHIVE=$HOME/archives
PGDATA=$HOME/data
PGPORT=5656
PGUSER=$USER

mkdir -p $ARCHIVE $PGDATA

initdb --data-checksum $PGDATA

cat << __EOF__ >> $PGDATA/postgresql.conf
port = $PGPORT
listen_addresses = '*'
cluster_name = 'test_archiver'
archive_mode = on
archive_library = 'basic_archive'
basic_archive.archive_directory = '$ARCHIVE'
__EOF__

pg_ctl start -D $PGDATA
```

Si on force un archivage, on voit que PostgreSQL a bien archivé dans la vue `pg_stat_archiver` :

```
psql -c "SELECT pg_create_restore_point('Forcer une ecriture dans les WAL.')"
psql -c "SELECT pg_switch_wal()"
psql -xc "SELECT * FROM pg_stat_archiver";
```

```
-[ RECORD 1 ]------+------------------------------
archived_count     | 1
last_archived_wal  | 000000010000000000000001
last_archived_time | 2022-07-06 17:39:33.632029+02
failed_count       | 0
last_failed_wal    | ¤
last_failed_time   | ¤
stats_reset        | 2022-07-06 17:39:23.287479+02
```

On peut vérifier la présence du fichier dans le répertoire :

```
SELECT file.name, stats.*
  FROM current_setting('basic_archive.archive_directory') AS archive(directory)
```

[1] https://www.postgresql.org/docs/15/basic-archive.html

```
    , LATERAL pg_ls_dir(archive.directory) AS file(name)
    , LATERAL pg_stat_file(archive.directory || '/' || file.name) AS stats

-[ RECORD 1 ]+-------------------------
name         | 000000010000000000000001
size         | 16777216
access       | 2022-07-06 17:39:33+02
modification | 2022-07-06 17:39:33+02
change       | 2022-07-06 17:39:33+02
creation     | ¤
isdir        | f
```

Le module `basic_archive` copie le WAL à archiver vers un fichier temporaire, le synchronise sur disque, puis le renomme. Si le fichier archivé existe déjà dans le répertoire cible et est identique, l'archivage est considéré comme un succès. Si le serveur plante, il est possible que des fichiers temporaires qui commencent par `archtemp` soient présents. Il est conseillé de les supprimer avant de démarrer PostgreSQL. Il est possible de les supprimer à chaud mais il faut s'assurer qu'il ne s'agisse pas d'un fichier en cours d'utilisation.

En cas d'échec de l'archivage, il est possible que l'erreur ne soit pas visible dans le titre du processus ou la vue `pg_stat_archiver`. C'est par exemple le cas si une erreur de configuration empêche le chargement du module. Les traces de PostgreSQL contiennent alors les informations nécessaires pour résoudre le problème.

Au moment de l'écriture de cet article, les paramètres des modules d'archivage ne sont pas visibles depuis `pg_settings` ou depuis la nouvelle méta-commande `\dconfig+` qui utilise cette vue.

Ce comportement s'explique par cette ligne de la documentation de la vue `pg_settings`[2] :

> This view does not display customized options until the extension module that defines them has been loaded.

C'est le processus d'archivage qui charge le module d'archivage, les paramètres qui y sont définis ne sont donc pas visibles des autres processus.

Plusieurs alternatives sont possibles pour consulter leurs valeurs :

- charger la librairie dans la session :

```
LOAD 'basic_archive';
SELECT name, setting FROM pg_settings WHERE name = 'basic_archive.archive_directory';
```

[2] https://www.postgresql.org/docs/current/view-pg-settings.html

```
                name                |                       setting
------------------------------------+-------------------------------------------------
 basic_archive.archive_directory | /home/benoit/var/lib/postgres/archives/pgsql-15b3
(1 row)
```

- consulter les valeurs des paramètres directement avec la commande `SHOW` de psql :

```
SHOW basic_archive.archive_directory;

 basic_archive.archive_directory
---------------------------------
 /home/benoit/archives
```

- voir les paramètres renseignés dans le fichier de configuration dans la vue `pg_file_settings` :

```
SELECT *
  FROM pg_file_settings
 WHERE name = 'basic_archive.archive_directory' \gx

-[ RECORD 1 ]----------------------------------------
sourcefile | /home/benoit/data/postgresql.conf
sourceline | 820
seqno      | 27
name       | basic_archive.archive_directory
setting    | /home/benoit/archives
applied    | t
error      | ¤
```

1.3.3 PERMETTRE LE PRE-FETCH DU CONTENU DES FICHIERS WAL PENDANT LE RECOVERY

- Accélération du recovery grâce au *prefetch* des blocs de données accédés dans les enregistrements de WAL
  - `recovery_prefetch` : `try, on, off`
  - `wal_decode_buffer_size` distance à laquelle on peut lire les WAL en avance de phase
- nouvelle vue : `pg_stat_recovery_prefetch`

Le nouveau paramètre `recovery_prefetch` permet d'activer le *prefetch* lors du rejeu des WAL. Il permet de lire à l'avance les WAL et d'initier la lecture asynchrone des blocs de données qui ne sont pas dans le cache de l'instance. Tous les OS ne permettent pas d'implémenter cette fonctionnalité, le paramètre a donc trois valeurs possibles `try`, `on` et `off`. La valeur par défaut est `try`.

`wal_decode_buffer_size` permet de limiter la distance à laquelle on peut lire les WAL en avance de phase. Sa valeur par défaut est de 512 ko.

Le GUC `maintenance_io_concurrency` est également utilisé pour limiter le nombre d'I/O concurrentes autorisées, ainsi que le nombre de blocs à lire en avance. Le calcul utilisé est le suivant : `maintenance_io_concurrency * 4` blocs.

Cette nouvelle fonctionnalité devrait accélérer grandement la recovery suite à un crash, une restauration ou lorsque la réplication utilise le *log shipping*.

Précédemment, pour réaliser ce genre d'optimisation, il fallait passer des outils externes comme pg_prefaulter[3] qui a servi d'inspiration à cette fonctionnalité.

Création d'un environnement de test :

```
PGDATA=/home/benoit/var/lib/postgres/testpg15
PGDATASAV=/home/benoit/var/lib/postgres/testpg15-save
PGUSER=postgres
PGPORT=5432

initdb --username "$PGUSER" "$PGDATA"
```

Démarrer PostgreSQL en forçant des checkpoints très éloignés les un des autres. Pour cela on augmente le timeout et la quantité maximale de WAL avant le déclenchement du checkpoint. On désactive aussi les *full page writes* pour éviter que les pages complètes soient dans les WAL. En effet dans ce cas, le préfetch est inutile.

```
pg_ctl -D "$PGDATA" \
       -o "-c checkpoint_timeout=60min -c max_wal_size=10GB -c full_page_writes=off" \
       -W \
       start
```

Ajouter des données avec pgbench pour générer des WAL.

```
pgbench -i -s300 postgres
psql postgres -c checkpoint
pgbench -T300 -Mprepared -c4 -j4 postgres
```

Tuer PostgreSQL pour forcer une restauration au redémarrage de PostgreSQL.

```
killall -9 postgres
```

Sauvegarder le répertoire de données.

```
cp -R "$PGDATA" "$PGDATASAV"
```

Démarrer PostgreSQL avec le prefetch désactivé :

[3] https://github.com/TritonDataCenter/pg_prefaulter

```
pg_ctl -D "$PGDATA" \
       -o "-c recovery_prefetch=off" \
       -W \
       start
```

Voici les traces du démarrage :

```
LOG:  starting PostgreSQL 15.1 on x86_64-pc-linux-gnu,
      compiled by gcc (GCC) 12.2.1 20220819 (Red Hat 12.2.1-2), 64-bit
LOG:  listening on IPv6 address "::1", port 5432
LOG:  listening on IPv4 address "127.0.0.1", port 5432
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
LOG:  database system was interrupted; last known up at 2023-02-10 23:17:43 CET
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/E7A522B8
LOG:  redo in progress, elapsed time: 10.00 s, current LSN: 0/E82D8528
LOG:  redo in progress, elapsed time: 20.00 s, current LSN: 0/E8B3A690
LOG:  redo in progress, elapsed time: 30.00 s, current LSN: 0/E93F3D98
LOG:  redo in progress, elapsed time: 40.00 s, current LSN: 0/E9C57E60
LOG:  redo in progress, elapsed time: 50.00 s, current LSN: 0/EA4EB5A8
FATAL:  the database system is not yet accepting connections
DETAIL:  Consistent recovery state has not been yet reached.
LOG:  redo in progress, elapsed time: 60.00 s, current LSN: 0/EAD8F530
FATAL:  the database system is not yet accepting connections
DETAIL:  Consistent recovery state has not been yet reached.
LOG:  invalid record length at 0/EB4CAF48: wanted 24, got 0
LOG:  redo done at 0/EB4CAF10 system usage: CPU: user: 6.75 s, system: 15.58 s, elapsed: 67.99 s
LOG:  checkpoint starting: end-of-recovery immediate wait
LOG:  checkpoint complete:
      wrote 10366 buffers (63.3%);
      0 WAL file(s) added, 4 removed, 0 recycled;
      write=0.325 s, sync=0.010 s, total=0.367 s;
      sync files=23, longest=0.005 s, average=0.001 s;
      distance=59875 kB, estimate=59875 kB
LOG:  database system is ready to accept connections
```

Arrêter PostgreSQL, copier de la sauvegarde du répertoire de données et démarrer PostgreSQL avec le prefetch activé :

```
pg_ctl -D "$PGDATA" \
       -m fast \
       stop

rm -fr "$PGDATA"
cp -r "$PGDATASAV" "$PGDATA"

pg_ctl -D "$PGDATA" \
```

```
      -o "-c recovery_prefetch=try" \
      -W \
      start

LOG:  starting PostgreSQL 15.1 on x86_64-pc-linux-gnu,
      compiled by gcc (GCC) 12.2.1 20220819 (Red Hat 12.2.1-2), 64-bit
LOG:  listening on IPv6 address "::1", port 5432
LOG:  listening on IPv4 address "127.0.0.1", port 5432
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
LOG:  database system was interrupted; last known up at 2023-02-10 23:17:43 CET
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/E7A522B8
LOG:  redo in progress, elapsed time: 10.00 s, current LSN: 0/EAD67BC0
LOG:  invalid record length at 0/EB4CAF48: wanted 24, got 0
LOG:  redo done at 0/EB4CAF10 system usage: CPU: user: 3.57 s, system: 7.26 s, elapsed: 11.46 s
LOG:  checkpoint starting: end-of-recovery immediate wait
LOG:  checkpoint complete:
      wrote 10179 buffers (62.1%);
      0 WAL file(s) added, 4 removed, 0 recycled;
      write=0.322 s, sync=0.039 s, total=0.429 s;
      sync files=23, longest=0.017 s, average=0.002 s;
      distance=59875 kB, estimate=59875 kB
LOG:  database system is ready to accept connections
```

On voit que le *redo* a duré 11.46s au lieu de 1min 8s du test lors du précédent.

Des statistiques peuvent être lues dans la nouvelle vue `pg_stat_recovery_prefetch` :

```
SELECT * FROM pg_stat_recovery_prefetch \gx

stats_reset    | 2023-02-10 23:35:55.873179+01
prefetch       | 200524
hit            | 416757
skip_init      | 2308
skip_new       | 0
skip_fpw       | 0
skip_rep       | 140012
wal_distance   | 0
block_distance | 0
io_depth       | 0
```

La signification des colonnes est la suivante :

- prefetch : Nombre de blocs récupérés avec le prefetch parce que le les blocs ne sont pas le buffer pool ;
- hit : Nombre de blocs qui n'ont pas été récupérés avec le prefetch car ils étaient déjà dans le buffer pool ;

- skip_init : Nombre de blocs qui n'ont pas été récupérés avec le prefetch car ils auraient été initialisé à zéro ;
- skip_init : Nombre de blocs qui n'ont pas été récupérés avec le prefetch car ils n'existaient pas encore ;
- skip_fpw : Nombre de blocs qui n'ont pas été récupérés avec le prefetch car une lecture de page complète était incluse dans le WAL ;
- skip_rep : Nombre de blocs qui n'ont pas été récupérés avec le prefetch car elles ont déjà été préfetchées récemment ;
- wal_distance : De combien de bytes le prefetcher est entrain de lire en avance ; block_distance : De combien de blocs le prefetcher est en train de lire en avance ;
- io_depth : Combien de prefetch ont été initialisés mais ne sont pas encore terminés.

1.3.4 PG_BASEBACKUP `--TARGET`

- Nouveau paramètre `-t/--target` pour `pg_basebackup`
  - `client`, `server` ou `blackhole`
- Sauvegarde sur le serveur seulement accessible aux membres du groupe `pg_write_server_files`
- Possibilité d'ajouter des cibles via des modules additionnels
  - module `basebackup_to_shell` fourni en exemple

Cette version introduit la notion de cible pour les sauvegardes effectuées avec `pg_basebackup`. Le nouveau paramètre `-t/--target` a été introduit à cet effet. Il peut prendre les valeurs :

- `client` : la sauvegarde est faite en local (c'est la valeur par défaut) ;
- `server` : la sauvegarde est faite sur le serveur de base de données ;
- `blackhole` : aucun fichier n'est créé, c'est utile pour les tests ;

Pour les modes `client` et `server`, il faut spécifier un chemin :

- pour `client`, il faut utiliser l'option `-D/--pgdata` ;
- pour `server`, il faut affixer `:` suivit d'un chemin à la cible.

Sauvegarder sur le serveur de base de données est une tâche plus sensible qu'effectuer une sauvegarde en local. C'est pour cette raison qu'il faut faire partie du groupe `pg_write_server_files` pour pouvoir utiliser cette cible.

Lorsque la cible est différente de `client`, l'option `-X/--wal-method` est requise et doit prendre la valeur `none` ou `fetch`. Si l'on choisit la méthode `fetch`, il peut donc être nécessaire de configurer `wal_keep_size` pour s'assurer que les WAL nécessaires pour rendre

la sauvegarde cohérente soient conservés jusqu'à la fin de l'opération. Si l'archivage est déjà configuré, l'option `none` peut être utilisée.

Voici quelques exemples de syntaxe pour la commande :

```
# Sauvegarde sur le serveur du client
pg_basebackup --checkpoint fast --progress \
              --target client --pgdata .

# Sauvegarde "à blanc"
pg_basebackup --checkpoint fast --progress \
              --target blackhole --wal-method fetch

# Sauvegarde sur le serveur de base de données
pg_basebackup --checkpoint fast --progress \
              --target server:/backup/15/main --wal-method fetch
```

Pour toutes les cibles différentes de `client` (la valeur par défaut), le format de la sauvegarde est obligatoirement `tar`. Par exemple, si `--format p` et `--target=server` sont spécifiés, l'erreur suivante est affichée.

```
pg_basebackup: error: cannot specify both format and backup target
```

Il est prévu de pouvoir étendre le fonctionnement `pg_basebackup` en ajoutant de nouveaux types de cibles. Le module de test `basebackup_to_shell` est fourni à titre d'exemple. Il permet d'exécuter une commande qui prend en entrée standard un fichier généré par la sauvegarde.

Le module ajoute à `pg_basebackup` la cible `shell`, pour laquelle il est possible d'affixer `:` et une chaîne de caractère. Cette chaîne de caractère ne peut contenir que des caractères alphanumériques.

Pour l'utiliser, il faut ajouter le module à `shared_preload_libraries` ou `local_preload_libraries`, et configurer les paramètres :

- `basebackup_to_shell.command` : une commande que le serveur va utiliser pour chaque fichier généré par `pg_basebackup`. Si `%f` est spécifié dans la commande, il sera remplacé par le nom de fichier. Si `%d` est spécifié dans la commande, il sera remplacé par la chaîne spécifiée après la cible ;
- `basebackup_to_shell.required_role` : le rôle requis pour pouvoir utiliser la cible `shell`. Il faut que l'utilisateur dispose de l'attribut `REPLICATION`.

Le module est fourni à titre d'exemple pour démontrer la création de ce genre de module et son utilisation. Son utilité est limitée. Nous allons ici l'utiliser pour chiffrer la sauvegarde :

```
BACKUP=$HOME/backup
PGDATA=$HOME/data
PGPORT=5656
PGUSER=$USER

mkdir -p $PGDATA $BACKUP

initdb --data-checksum $PGDATA

cat << __EOF__ >> $PGDATA/postgresql.conf
port = $PGPORT
listen_addresses = '*'
cluster_name = 'test_shell_module'
shared_preload_libraries = 'basebackup_to_shell'
basebackup_to_shell.command = 'gpg --encrypt --recipient %d --output $BACKUP/%f'
basebackup_to_shell.required_role = 'gpg'
__EOF__

pg_ctl start -D $PGDATA

psql -c "CREATE ROLE gpg WITH LOGIN PASSWORD 'secret'"
psql -c "CREATE ROLE non_autorise WITH LOGIN PASSWORD 'secret'"
echo "localhost:5656:postgres:gpg:secret" >> $HOME/.pgpass
echo "localhost:5656:postgres:non_autorise:secret" >> $HOME/.pgpass
chown $USER $HOME/.pgpass
chmod 600 $HOME/.pgpass
```

Effectuer une sauvegarde avec le module `shell` et l'utilisateur **gpg** :

```
pg_basebackup --checkpoint fast --progress --user gpg \
              --target shell:$PGUSER --wal-method fetch
```

On peut contrôler que le `backup_manifest` est bien chiffré en l'éditant. Pour l'afficher en clair :

```
gpg -d $BACKUP/backup_manifest
```

Avec l'utilisateur **non_autorise**, la sauvegarde échoue :

```
pg_basebackup --checkpoint fast --progress --user non_autorise \
              --target shell:$PGUSER --wal-method fetch

pg_basebackup: error: could not initiate base backup:
+++ ERROR:  permission denied to use basebackup_to_shell
```

1.3.5 AJOUT DE NOUVEAUX ALGORITHMES DE COMPRESSION

- Écritures de page complètes :
  - `pglz` (défaut utilisé pour `on`), `lz4`, `zstd`
- Sauvegardes avec `pg_basebackup` :
  - `--compression [{client|server}-]method:detail`
  - method: `gzip`, `lz4`, `zstd`
  - detail: `[level=]entier`, `workers=entier` (`zstd`)
- Récupération de WAL avec `pg_receivewal` :
  - `--compression method:detail`
  - method: `gzip`, `lz4`
  - detail: `[level=]entier`

PostgreSQL permet désormais d'utiliser les algorithmes de compressions *LZ4*, *Zstandard* en plus de *gzip* pour la compression des sauvegardes, des WAL et des écritures de page complètes.

Avantages attendus par type de compression

`gzip` est la méthode de compression historique de PostgreSQL, elle est utilisée par défaut.

`lz4` est plus rapide que `gzip` mais a généralement un taux de compression inférieur.

`zstd` présente l'avantage de permettre la parallélisation de la compression, ce qui permet plus de performances.

Type de compression et compilation

L'utilisation de l'algorithme `lz4` nécessite l'utilisation du paramètre `--with-lz4` lors de la compilation. Ce paramètre avait été ajouté en version 14 pour permettre l'utilisation de `lz4` afin de compresser les TOAST.

Le paramètre `--with-zstd` a été ajouté en version 15 pour permettre l'utilisation de l'algorithme `zstd`. Ces paramètres sont activés par défaut sur les distributions de type RockyLinux et Debian.

Écriture de pages complètes

Le paramètre `wal_compression` acceptait précédemment deux valeurs `on` et `off`. Il permettait d'activer ou non la compression des images de page complètes (*FPI: Full Page Image*) écrites dans les WAL lorsque le paramètre `full_page_writes` était activé ou pendant une sauvegarde physique.

En version 15, trois nouveaux paramètres sont ajoutés et permettent de contrôler le type d'algorithme de compression utilisé parmi : `pglz`, `lz4` et `zstd`. Le mode de compression par défaut est `pglz`, c'est l'algorithme choisi si l'on valorise `wal_compression` à `on`.

pg_basebackup

Il est désormais également possible de spécifier l'algorithme de compression utilisé par `pg_basebackup` avec l'option `--compression / -Z` dont la nouvelle spécification est :

```
-Z level
-Z [{client|server}-]method[:detail]
--compress=level
--compress=[{client|server}-]method[:detail]
```

Les valeurs possibles pour la méthode de compression sont `none`, `gzip`, `lz4` et `zstd`. Lorsqu'un algorithme de compression est spécifié, il est possible d'ajouter des options de compression en ajoutant une série de paramètre précédé de deux point et séparé par des virgules, sous la forme d'un `mot clé` ou d'un `mot clé=valeur`. Pour le moment, les mots clé suivants sont supportés :

- `[level=]entier` permet de spécifier le niveau de compression ;
- `workers=entier` permet de spécifier le nombre de processus pour la parallélisation de la compression.

Exemples :

```
$ pg_basebackup --format t \
                --compress server-lz4:1 \
                --pgdata splz41

$ pg_basebackup --format t \
                --compress server-lz4:level=1 \
                --pgdata splz411

$ pg_basebackup --format t \
                --compress server-zstd:level=9,workers=2 \
                --pgdata spzstdl9w2
```

Seule la compression `zstd` accepte le paramètre `workers` :

```
$ pg_basebackup -Ft \
                --compress=server-zstd:level=9,workers=2 \
                --pgdata stzstl9w2

$ pg_basebackup -Ft \
                --compress server-lz4:level=9,workers=2 \
                --pgdata stlzl9w2

pg_basebackup: error: could not initiate base backup:
+++ERROR:  invalid compression specification: compression algorithm "lz4" does not accept a worker
pg_basebackup: removing data directory "stlzl9w2"
```

Si aucun algorithme de compression n'est spécifié et que le niveau de compression est de

0, aucune compression n'est mise en place. Si le niveau de compression est supérieur à zéro, la compression `gzip` est utilisée avec le niveau spécifié.

Il est possible de spécifier le lieu où sera effectuée la compression en précédant le nom de l'algorithme de compression par `client-` ou `server-`. Activer la compression côté serveur permet de réduire le coût en bande passante au prix de l'augmentation de la consommation CPU. La valeur par défaut est `client` à moins que l'option `--target=server...` ne soit spécifiée, dans ce cas, la sauvegarde est réalisée sur le serveur de base de données, la compression sera donc réalisée également sur le serveur. La notion de cible est abordée dans un chapitre séparé.

Exemple d'une sauvegarde réalisée côté serveur :

```
$ pg_basebackup --wal-method fetch \
                --target server:/var/lib/postgres/sauvegarde/sstlz4 \
                --compress server-lz4
$ ls ./sstlz4/
```

```
backup_manifest base.tar.lz4
```

On peut voir que si le format ne peut être spécifié avec une sauvegarde côté serveur, il est forcé à `tar`.

```
$ pg_basebackup --wal-method fetch \
                --format p \
                --target server:/var/lib/postgres/sauvegarde/sstlz4 \
                --compress server-lz4
```

```
pg_basebackup: error: cannot specify both format and backup target
pg_basebackup: hint: Try "pg_basebackup --help" for more information
```

La compression des WAL côté serveur n'est pas possible quand `-Xstream` (ou `--wal-method stream`) est utilisé. Pour cela, il faut utiliser `-Xfetch`.

L'exemple ci-dessous montre qu'avec la compression côté serveur et l'option `-Xstream`, les WAL sont dans un fichier `tar` non compressé : `pg_wal.tar`.

```
$ pg_basebackup -Xstream \
                --format t \
                --compress server-gzip \
                --pgdata ./sctgzs
$ ls ./sctgzs/
```

```
backup_manifest base.tar.gz pg_wal.tar
```

Avec l'option `-Xfetch`, les WAL sont placés dans le répertoire `pg_wal` et compressés avec le reste de la sauvegarde.

```
$ pg_basebackup -Xfetch \
                --format t \
```

```
                --compress server-gzip \
                --pgdata ./sctgzf
$ ls ./sctgzf

backup_manifest base.tar.gz
```

Si la compression est réalisée côté client et que l'option `-Xstream` est choisie, l'algorithme de compression sélectionné doit être `gzip`. Dans ce cas, le fichier `pg_wal.tar` sera compressé en `gzip`. Si un autre algorithme est choisi, le fichier ne sera pas compressé.

Si le format `tar` est spécifié (`--format=t` / `Ft`) avec `gzip`, `lz4` et `zstd`, l'extension du fichier de sauvegarde sera respectivement `.gz`, `.lz4` ou `.zst`.

Dans cet exemple d'une compression avec `gzip`, on voit que `pg_wal.tar` est compressé et que l'extension des fichiers compressé est `.gz`.

```
$ pg_basebackup -Ft --compress=gzip --pgdata tgzip
$ ls ./tgzip/

backup_manifest base.tar.gz pg_wal.tar.gz
```

Exemple d'une compression avec `lz4`, on voit que `pg_wal.tar` n'est pas compressé et que l'extension des fichiers compressé est `.lz4`.

```
$ pg_basebackup -Ft --compress=lz4 --pgdata tlz4 --progress
$ ls ./tlz4/

backup_manifest base.tar.lz4 pg_wal.tar
```

Exemple d'une compression avec `zstd`, on voit que `pg_wal.tar` n'est pas compressé et que l'extension des fichiers compressé est `.zst`.

```
$ pg_basebackup -Ft --compress=zstd --pgdata tzstd --progress
$ ls ./tzstd/

backup_manifest base.tar.zst pg_wal.tar
```

Si le format `plain` est utilisé (`--format=p` / `-Fp`), la compression côté client ne peut pas être choisie. Elle peut en revanche être spécifiée côté serveur. Dans ce cas, le serveur va compresser les données pour le transfert et le client les décompressera ensuite.

```
$ pg_basebackup -Fp --compress=lz4 --pgdata plz4

pg_basebackup: error: only tar mode backups can be compressed
```

Dans cet exemple, on voit que la sauvegarde est compressée côté serveur et décompressée sur le client :

```
$ pg_basebackup -Fp --compress=server-lz4 --pgdata scplz4
$ ls -al scplz4/

total 372
drwx------. 20 postgres postgres   4096 Dec 12 17:49 .
```

```
drwxrwxr-x.  6 postgres postgres   4096 Dec 12 17:49 ..
-rw-------.  1 postgres postgres    227 Dec 12 17:49 backup_label
-rw-------.  1 postgres postgres 235587 Dec 12 17:49 backup_manifest
drwx------.  7 postgres postgres   4096 Dec 12 17:49 base
-rw-------.  1 postgres postgres     30 Dec 12 17:49 current_logfiles
drwx------.  2 postgres postgres   4096 Dec 12 17:49 global
drwx------.  2 postgres postgres   4096 Dec 12 17:49 log
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_commit_ts
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_dynshmem
-rw-------.  1 postgres postgres   4789 Dec 12 17:49 pg_hba.conf
-rw-------.  1 postgres postgres   1636 Dec 12 17:49 pg_ident.conf
drwx------.  4 postgres postgres   4096 Dec 12 17:49 pg_logical
drwx------.  4 postgres postgres   4096 Dec 12 17:49 pg_multixact
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_notify
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_replslot
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_serial
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_snapshots
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_stat
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_stat_tmp
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_subtrans
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_tblspc
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_twophase
-rw-------.  1 postgres postgres      3 Dec 12 17:49 PG_VERSION
drwx------.  3 postgres postgres   4096 Dec 12 17:49 pg_wal
drwx------.  2 postgres postgres   4096 Dec 12 17:49 pg_xact
-rw-------.  1 postgres postgres     88 Dec 12 17:49 postgresql.auto.conf
-rw-------.  1 postgres postgres  29665 Dec 12 17:49 postgresql.conf
```

Le test suivant consiste à sauvegarder une base de 630Mo contenant principalement du *jsonb* avec les trois algorithmes de compression. Le test est réalisé sur un portable avec 8 CPU, 8 Go de RAM et un disque SSD.

Le tableau suivant montre le volume des sauvegardes (hors WAL `-Xnone`) par niveau de compression. On peut voir que l'algorithme le plus performant est `zstd`.

Niveau de compression	Vol. gzip	Vol. lz4	Vol. zstd
1	395 Mo (37%)	498 Mo (20%)	418 Mo (33%)
2	387 Mo (38%)	498 Mo (20%)	391 Mo (37%)
3	379 Mo (39%)	406 Mo (35%)	373 Mo (40%)
4	375 Mo (40%)	401 Mo (36%)	362 Mo (42%)
5	368 Mo (41%)	399 Mo (36%)	353 Mo (43%)
6	365 Mo (42%)	398 Mo (36%)	348 Mo (44%)
7	364 Mo (42%)	397 Mo (36%)	339 Mo (46%)
8	364 Mo (42%)	397 Mo (36%)	337 Mo (46%)

Niveau de compression	Vol. gzip	Vol. lz4	Vol. zstd
9	364 Mo (42%)	397 Mo (36%)	329 Mo (47%)

Le tableau suivant montre les temps de sauvegarde par niveau de compression. Pour le mode de compression `zstd`, le chiffre qui suit correspond au nombre de processus utilisés pour la compression. On voit ici que l'algorithme le plus rapide est `lz4`. `zstd` permet d'obtenir de meilleures performances si on augmente le nombre de processus dédiés à la compression.

Niveau	Vol. gzip	Vol. lz4	Vol. zstd 1	Vol. zstd 2	Vol. zstd 3
1	19.3 s	3.9 s	6.2 s	3.5 s	3.7 s
2	21.0 s	4.0 s	7.4 s	3.8 s	3.2 s
3	24.8 s	13.1 s	9.7 s	5.4 s	3.7 s
4	26.7 s	15.3 s	12.1 s	9.5 s	8.5 s
5	34.5 s	18.2 s	14.2 s	9.4 s	7.7 s
6	44.0 s	20.5 s	19.9 s	10.8 s	8.6 s
7	51.8 s	21.7 s	22.1 s	14.9 s	12.6 s
8	61.0 s	23.8 s	26.2 s	17.1 s	14.4 s
9	67.0 s	24.7 s	29.3 s	21.6 s	19.3 s

pg_receivewal

Le dernier outil qui bénéficie des nouveaux algorithmes de compression supportés par PostgreSQL est `pg_receivewal`. Là aussi, l'option `--compression / -Z` est utilisée et sa nouvelle spécification est :

```
-Z level
-Z method[:detail]
--compress=level
--compress=method[:detail]
```

Le principe est le même que pour `pg_basebackup` à quelques différences près :

- `pg_receivewal` compresse forcément les WAL côté client ;
- les algorithmes de compression disponible sont `gzip` et `lz4`. Cette évolution permettra donc d'avoir le choix entre taux de compression (`gzip`) et vitesse de compression (`lz4`).

La compression par défaut est `gzip`, les fichiers produits se terminent donc pas l'extension `.gz`. Le niveau de compression peut être ajouté après la méthode de compression sous forme d'un entier ou avec l'ensemble clé valeur `level=nombre entier`.

```
$ pg_receivewal --compress 2
$ pg_receivewal --compress gzip:2
$ pg_receivewal --compress gzip:level=2
```

Avec les commandes précédentes, on obtient :

```
000000010000000000000E6.gz 000000010000000000000E7.gz.partial
```

Les fichiers compressés avec `lz4` se terminent par `.lz4`.

```
$ pg_receivewal --compress lz4:level=1
```

Avec la commande précédente, on obtient :

```
total 80
000000010000000000000E7.lz4 000000010000000000000E8.lz4.partial
```

1.3.6 PG_DUMP

- Amélioration des performances d'export de bases avec de nombreux objets
  - désormais une seule requête pour toutes les tables à exporter
  - élimination de sous-requêtes non nécessaires
  - utilisation de `PREPARE`/`EXECUTE` pour les requêtes répétitives
- Amélioration des performances d'export parallélisé de tables TOAST
  - données TOAST désormais comptabilisées dans la planification d'un export parallélisé

Diverses optimisations ont été apportées pour améliorer les performances de l'outil `pg_dump` lorsque l'on souhaite exporter un grand nombre d'objets. Avant la version 15, `pg_dump` lançait une requête pour chaque objet dont il devait exporter les données. Désormais, il ne lance plus qu'une seule requête et c'est une clause `WHERE` qui permet de se limiter aux seuls objets voulus dans l'export.

Lorsque l'on exporte beaucoup d'objets similaires, il est probable qu'une même requête soit répétée de nombreuses fois, en changeant seulement la valeur des paramètres. C'est pourquoi `pg_dump` utilise désormais les clauses `PREPARE` et `EXECUTE`, afin de ne calculer qu'une seule fois le plan d'execution.

Afin d'éviter l'utilisation d'une sous-requête, qui peut pénaliser les performances lorsqu'un grand nombre d'objets sont exportés, `pg_dump` récupère désormais les noms de rôles via leurs OIDs. Une autre sous-requête vérifiant le lien de dépendance (`pg_depend`) entre relations et séquences a été supprimée car cette vérification était redondante avec une autre déjà en place.

L'export parallélisé des tables TOAST bénéficie d'une amélioration de ses performances car l'estimation du volume des tables a été corrigée. Cette estimation ne prenait pas en compte les champs stockés dans les tables TOAST dans le calcul du volume des tables à exporter, ceci pouvait déséquilibrer la répartition de charge des processus lancés en parallèle.

1.4 NOUVELLES VUES ET PARAMÈTRES

1.4.1 AJOUT DE LA VUE SYSTÈME PG_IDENT_FILE_MAPPINGS POUR REPORTER LES INFORMATIONS DU FICHIER PG_IDENT.CONF

- Nouvelle vue `pg_ident_file_mappings`
- Résume le contenu actuel du fichier `pg_ident.conf`
- Permet le diagnostique d'erreur et la validation de la configuration

De façon similaire à la vue `pg_hba_file_rules`, la nouvelle vue système `pg_ident_file_mappings` donne un résumé du fichier de configuration `pg_ident.conf`. En plus des informations contenues dans le fichier `pg_ident.conf`, elle va fournir une colonne `error` qui va permettre de vérifier le fonctionnement de la configuration avant application ou de diagnostiquer un éventuel problème.

Cette vue n'intervient que sur le contenu actuel du fichier, et non pas sur ce qui a pu être chargé par le serveur. Par défaut elle n'est accessible que pour les super-utilisateurs.

Voici un exemple de ce que peut retourner la vue `pg_ident_file_mappings` :

```
postgres=# select * from pg_ident_file_mappings;
 line_number | map_name |         sys_name         | pg_username |            error
-------------+----------+--------------------------+-------------+-----------------------------
          43 | workshop | dalibo                   | test        |
          44 | mymap    | /^(.*)@mydomain\.com$    | \1          |
          45 | mymap    | /^(.*)@otherdomain\.com$ | guest       |
          46 |          |                          |             | missing entry at end of line
```

On peut remarquer ci-dessus, qu'une erreur est retournée par la vue `pg_ident_file_mappings` à la ligne 46 du fichier `pg_ident.conf` : `missing entry at end of line`.

1.4.2 AJOUT DE LA VUE SYSTÈME PG_STAT_SUBSCRIPTION_STATS POUR REPORTER L'ACTIVITÉ D'UN SOUSCRIPTEUR (CF. RÉPLICATION LOGIQUE)

- Donne des informations sur les erreurs qui se sont produites durant la réplication logique
- Ajout de la fonction `pg_stat_reset_subscription_stats()`

La vue système `pg_stat_subscription_stats` permet de récupérer des informations sur les erreurs qui se sont produisent au niveau des souscriptions avec la réplication logique. Ces données sont stockées sous forme de compteur et concernent les erreurs rencontrées

lors de l'application des changements ou lors de la synchronisation initiale. Elle contient une ligne par souscription.

Voici la description des colonnes de cette vue :

- `subid` : *OID* de la souscription ;
- `subname` : nom de la souscription ;
- `apply_error_count` : nombre d'erreurs rencontrées lors de l'application des changements ;
- `sync_error_count` : nombre d'erreurs rencontrées lors de la synchronisation initiale des tables ;
- `stats_reset` : date de réinitialisation des statistiques.

La fonction `pg_stat_reset_subscription_stats` permet de réinitialiser les statistiques de la vue `pg_stat_subscription_stats`. Elle prend en paramètre soit l'*OID* d'une souscription pour ne réinitialiser que les statistiques de cette dernière, soit `NULL` pour appliquer la réinitialisation à **toutes** les souscriptions.

1.4.3 AJOUT DE NOUVELLES VARIABLES SERVEUR SHARED_MEMORY_SIZE ET SHARED_MEMORY_SIZE_IN_HUGE_PAGES

- Ajout de deux nouvelles variables serveur :
  - `shared_memory_size` : détermine la taille de la mémoire partagée
  - `shared_memory_size_in_huge_pages` : détermine le nombre de *Huge Pages* nécessaires pour stocker la mémoire partagée
- Englobe les éléments chargés avec `shared_preload_libraries`
- Uniquement accessible en lecture seule

La variable `shared_memory_size` renvoie la taille de la mémoire partagée de PostgreSQL. Le résultat est calculé après le chargement des modules complémentaires (`shared_preload_libraries`). Il tient donc compte des éventuels modules et extensions qui pourraient consommer de la mémoire partagée supplémentaire.

```
# show shared_memory_size;
 shared_memory_size
--------------------
 143MB
```

On obtient quelque chose de similaire en faisant la somme des zones de mémoire partagée allouées avec la vue `pg_shmem_allocations` :

```
# select pg_size_pretty(sum(allocated_size)) from pg_shmem_allocations;
 pg_size_pretty
```

```
---------------
 143 MB
```

La variable `shared_memory_size_in_huge_pages` va quant à elle indiquer le nombre de *Huge Pages* nécessaires pour stocker la mémoire partagée de PostgreSQL. Elle est basée sur la valeur de
`shared_memory_size` vue précédemment et sur la taille des *Huge Pages* du système. Pour récupérer cette taille, PostgreSQL va en premier lieu regarder si le paramètre `huge_page_size` apparu en version 14 est défini. Si c'est le cas, il sera utilisé pour le calcul sinon, c'est le paramétrage du système qui sera utilisé (`/proc/meminfo`).

```
# show shared_memory_size_in_huge_pages;
 shared_memory_size_in_huge_pages
----------------------------------
 72
```

Il faut également que PostgreSQL puisse utiliser les *Huge Pages*. Le paramètre `huge_pages` doit
donc être défini à `on` ou `try`. Si elles ne sont pas utilisables ou si l'on se trouve sur un autre système que linux, `shared_memory_size_in_huge_pages` retournera `-1`.

Autre particularité avec ces deux variables, ce sont des variables *calculées durant l'exécution* (`runtime-computed GUC`). Dans les versions antérieures, la consultation de ce type de paramètre avec la commande `postgres -C` renvoyait des valeurs erronées car elle nécessitait le chargement d'éléments complémentaires (ce que ne faisait pas l'ancienne implémentation). La version 15 vient corriger ce problème et permet d'obtenir des valeurs correctes pour ces paramètres. Seule restriction, les paramètres `runtime-computed GUC` ne sont consultables avec `postgres -C` que lorsque l'instance est arrêtée.

```
postgres -C shared_memory_size -D $PGDATA
postgres -C shared_memory_size_in_huge_pages -D $PGDATA
```

On peut donc dorénavant savoir combien de mémoire partagée et de *Huge Pages* le système à besoin avant de démarrer une instance PostgreSQL.

1.5 PARTITIONNEMENT

1.5.1 AMÉLIORATION DU COMPORTEMENT DES CLÉS ÉTRANGÈRE LORS DE MISES À JOUR QUI DÉPLACENT DES LIGNES ENTRES LES PARTITIONS

- Correction du comportement de PostgreSQL lorsqu'un `UPDATE` sur une table partitionnée référencée par une contrainte de clé étrangère provoque la migration d'une ligne vers une autre partition.

Lorsqu'un `UPDATE` sur une table partitionnée référencée par une contrainte de clé étrangère provoque la migration d'une ligne vers une autre partition, l'opération est implémentée sous la forme d'un `DELETE` sur la partition source, suivi d'un `INSERT` sur la partition cible.

Sur les versions précédentes, cela pose un souci lorsque la contrainte de clé étrangère implémente la clause `ON DELETE`. En effet, dans ce cas, le changement de partition provoque le déclenchement de l'action associée à la commande `DELETE`, par exemple : une suppression. C'est une erreur puisqu'en réalité la ligne est juste déplacée vers une autre partition.

En version 15, le trigger posé par la contrainte de clé étrangère ne se déclenche plus sur le `DELETE` exécuté sur la partition, mais sur un `UPDATE` exécuté sur la table mère. Cela permet d'obtenir le comportement attendu.

L'implémentation choisie à une limitation : elle ne fonctionne que si la contrainte de clé étrangère concerne la table partitionnée. Cela ne devrait pas être un facteur limitant, en effet, il est rare d'avoir des clés étrangères différentes qui pointent vers les différentes partitions. On trouve généralement plutôt une clé étrangère qui pointe vers une ou plusieurs colonnes de la table partitionnée dans son ensemble.

Voici un exemple du comportement en version 14 puis 15.

Mise en place :

```
CREATE TABLE tpart (i int PRIMARY KEY, t text) PARTITION BY RANGE ( i );
CREATE TABLE tpart_1_10 PARTITION OF tpart FOR VALUES FROM (1) TO (10);
CREATE TABLE tpart_11_20 PARTITION OF tpart FOR VALUES FROM (11) TO (20);
CREATE TABLE foreignk(j int PRIMARY KEY, i int CONSTRAINT fk_tpart_i REFERENCES tpart(i) ON DELETE
INSERT INTO tpart VALUES (1, 'value 1');
INSERT INTO foreignk VALUES (1, 1, 'fk 1');
```

Voici les données présentes dans les tables :

```
SELECT *, tableoid::regclass FROM tpart;
```

```
 i |    t    |  tableoid
---+---------+------------
```

```
 1 | value 1 | tpart_1_10
(1 row)

SELECT * FROM foreignk ;

 j | i |  t
---+---+------
 1 | 1 | fk 1
(1 row)
```

Mise à jour et nouveaux contrôles en version 14 :

```
UPDATE tpart SET i = 11 WHERE i = 1;
SELECT *, tableoid::regclass FROM tpart;

 i  |    t    |  tableoid
----+---------+-------------
 11 | value 1 | tpart_11_20
(1 row)

SELECT * FROM foreignk ;

j | i | t
---+---+---
(0 rows)
```

La ligne a bien changé de partition, en revanche elle a été supprimée de la table qui référence la table partitionnée.

Avec PostgreSQL 15, on obtient désormais l'erreur suivante :

```
ERROR:  update or delete on table "tpart" violates foreign key constraint "fk_tpart_i" on table "f
DETAIL:  Key (i)=(1) is still referenced from table "foreignk".
```

1.6 TRACES

1.6.1 ACTIVATION DE LA JOURNALISATION DES CHECKPOINT ET OPÉRATIONS DE VACUUM LENTES

- Changement des valeurs par défaut des paramètres de journalisation :
  - `log_checkpoints` par défaut à `on`
  - `log_autovacuum_min_duration` par défaut à 10 minutes.

`log_checkpoints`

Le paramètre `log_checkpoints` est désormais à `on` par défaut, chaque `CHECKPOINT` sera par conséquent journalisé dans les traces de l'instance.

Les traces générées par ce paramètre contiennent des informations sur la durée des `CHECKPOINT` et sur les écritures effectuées :

```
2022-07-15 09:40:01.393 UTC [4198] LOG:  checkpoint starting: wal
2022-07-15 09:42:16.273 UTC [4198] LOG:  checkpoint complete: wrote 67 buffers (0.4%);
   0 WAL file(s) added, 0 removed, 134 recycled;
   write=134.352 s, sync=0.001 s, total=134.880 s; sync files=9, longest=0.001 s, average=0.001 s;
   distance=2192214 kB, estimate=2193764 kB
2022-07-15 09:42:29.121 UTC [4198] LOG:  checkpoint starting: wal
2022-07-15 09:43:56.646 UTC [4198] LOG:  checkpoint complete: wrote 81 buffers (0.5%);
   0 WAL file(s) added, 0 removed, 134 recycled;
   write=86.438 s, sync=0.026 s, total=87.525 s; sync files=8, longest=0.012 s, average=0.004 s;
   distance=2198655 kB, estimate=2198655 kB
2022-07-15 09:43:58.331 UTC [4198] LOG:  checkpoint starting: wal
2022-07-15 09:45:34.024 UTC [4198] LOG:  checkpoint complete: wrote 29 buffers (0.2%);
   0 WAL file(s) added, 0 removed, 134 recycled;
   write=94.874 s, sync=0.028 s, total=95.693 s; sync files=9, longest=0.016 s, average=0.004 s;
   distance=2192128 kB, estimate=2198003 kB
```

`log_autovacuum_min_duration`

Le paramètre `log_autovacuum_min_duration` est désormais configuré à 10 minutes. Cela signifie que chaque opération d'*autovacuum* qui dépasse ce délai sera tracée.

Les traces générées par ce paramètre permettent d'obtenir un rapport détaillé sur les opérations de `VACUUM` et `ANALYZE` exécutées par l'*autovacuum* :

```
2022-07-15 09:53:05.049 UTC [6563] LOG:  automatic vacuum of table "postgres.public.db_activity":
    pages: 0 removed, 108334 remain, 75001 scanned (69.23% of total)
    tuples: 0 removed, 9926694 remain, 2591536 are dead but not yet removable
    removable cutoff: 1024, which was 2 XIDs old when operation ended
    index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
```

```
    avg read rate: 62.039 MB/s, avg write rate: 16.120 MB/s
    buffer usage: 91282 hits, 58777 misses, 15272 dirtied
    WAL usage: 1 records, 1 full page images, 2693 bytes
    system usage: CPU: user: 1.34 s, system: 0.22 s, elapsed: 7.40 s
2022-07-15 09:53:08.658 UTC [6563] LOG:  automatic analyze of table "postgres.public.db_activity"
    avg read rate: 55.129 MB/s, avg write rate: 27.167 MB/s
    buffer usage: 4746 hits, 25467 misses, 12550 dirtied
    system usage: CPU: user: 0.37 s, system: 0.15 s, elapsed: 3.60 s
```

1.6.2 FORMAT DE SORTIE JSON POUR LES TRACES

- Nouveau format de sortie pour les fichiers trace : `jsonlog`

Le paramètre `log_destination` se voit enrichi d'une nouvelle option `jsonlog` qui permet d'obtenir une journalisation au format JSON.

```
postgres=# show log_destination ;
 log_destination
-----------------
 jsonlog
```

Le fichier de log produit aura alors l'extension `.json` :

```
postgres=# SELECT pg_current_logfile();
   pg_current_logfile
-------------------------
 log/postgresql-Fri.json
(1 row)
```

Voici un exemple d'une ligne de trace, la première générée au démarrage de l'instance :

```
{
  "timestamp": "2022-07-26 10:26:36.370 UTC",
  "pid": 3330,
  "session_id": "62dfc15c.d02",
  "line_num": 2,
  "session_start": "2022-07-26 10:26:36 UTC",
  "txid": 0,
  "error_severity": "LOG",
  "message": "starting PostgreSQL 15beta2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0 2021
  "backend_type": "postmaster",
  "query_id": 0
}
```

Le format JSON peut s'avérer utile pour alimenter les traces de l'instance dans un autre programme. pgBadger supporte déjà l'analyse de traces dans ce format, car il supportait

auparavant l'extension jsonlog qui ajoutait cette fonctionnalité avant qu'elle soit intégrée en standard dans PostgreSQL.

Par ailleurs, l'utilisation de l'outil jq permet de rechercher des clés spécifiques dans les traces, par exemple pour n'afficher que les erreurs :

```
[postgres@pg1 log]$ jq 'select(.error_severity == "ERROR" )' postgresql-Tue.json
{
  "timestamp": "2022-07-26 10:45:16.563 UTC",
  "user": "postgres",
  "dbname": "postgres",
  "pid": 3361,
  "remote_host": "[local]",
  "session_id": "62dfc250.d21",
  "line_num": 1,
  "ps": "INSERT",
  "session_start": "2022-07-26 10:30:40 UTC",
  "vxid": "3/20",
  "txid": 0,
  "error_severity": "ERROR",
  "state_code": "42P01",
  "message": "relation \"t2\" does not exist",
  "statement": "insert into t2 values ('missing_table_test');",
  "cursor_position": 13,
  "application_name": "psql",
  "backend_type": "client backend",
  "query_id": 0
}
```

Les données peuvent également être chargées dans une table. Il n'est pas possible d'utiliser COPY directement pour cela car les caractère d'échappement disparaissent.

```
postgres=# CREATE TABLE pglog( data jsonb);
CREATE TABLE
postgres=# COPY pglog FROM PROGRAM 'sed ''s/\\/\\\\/g'' log/postgresql-Fri.json';
COPY 52
postgres=# SELECT data->>'timestamp' AS starttime FROM pglog WHERE data ->> 'message'
          starttime
-------------------------------
 2022-08-19 16:47:48.412 CEST
(1 row)
```

1.6.3 INFORMATIONS SUPPLÉMENTAIRES DANS VACUUM VERBOSE

- Optimisations du code de la commande `VACUUM`
- Amélioration de la verbosité de la commande `VACUUM VERBOSE`

Le code de la commande `VACUUM` a été simplifié et optimisé. La nouvelle version permet de collecter plus d'informations sur l'exécution de l'opération de maintenance. Par conséquent, la sortie de la commande `VACUUM VERBOSE` est encore plus verbeuse dans PostgreSQL 15. En voici un exemple :

```
postgres=# VACUUM VERBOSE T1;
INFO:  vacuuming "postgres.public.t1"
INFO:  table "t1": truncated 1 to 0 pages
INFO:  finished vacuuming "postgres.public.t1": index scans: 0
pages: 1 removed, 0 remain, 1 scanned (100.00% of total)
tuples: 5 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 747, which was 1 XIDs old when operation ended
new relfrozenxid: 747, which is 6 XIDs ahead of previous value
index scan not needed: 1 pages from table (100.00% of total) had 5 dead item identifiers removed
avg read rate: 2.637 MB/s, avg write rate: 4.394 MB/s
buffer usage: 7 hits, 3 misses, 5 dirtied
WAL usage: 6 records, 2 full page images, 9339 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

La commande affiche un rapport détaillé de l'exécution, on y voit apparaître :

- le nouveau `relfrozenxid` après l'opération
- des informations sur l'utilisation des buffers
- des informations sur le nettoyage effectué sur les index de la table
- des métriques sur les performances du VACUUM : `avg read rate` et `avg write rate`.

La même commande en version 14 affichait un rapport moins complet:

```
postgres=# VACUUM VERBOSE T1;
INFO:  vacuuming "public.t1"
INFO:  table "t1": removed 5 dead item identifiers in 1 pages
INFO:  table "t1": found 5 removable, 0 nonremovable row versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 745
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  table "t1": truncated 1 to 0 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

1.7 DIVERS

1.7.1 POSSIBILITÉ DE DONNER/RESTREINDRE LES DROITS AUX COMMANDES SET / ALTER SYSTEM POUR LES UTILISATEURS NON PRIVILÉGIÉS

- Apparition de deux nouveaux privilèges :
  - `SET` : permet de modifier les paramètres avec le context `superuser`
  - `ALTER SYSTEM` : permet à un utilisateur non `superuser` de modifier des paramètres avec `ALTER SYSTEM SET ...`
- Donne des droits par rôle et par paramètre
- Nouvelle table système `pg_parameter_acl` qui stocke la configuration

Deux nouveaux privilèges arrivent en version 15 : `SET` et `ALTER SYSTEM`. `SET` va permettre d'autoriser la modification des paramètres de contexte `superuser` pour des rôles non privilégiés via la commande du même nom :

```
# Je dispose d'un utilisateur dalibo non superuser
postgres=> \du+ dalibo
                   Liste des rôles
 Nom du rôle | Attributs | Membre de | Description
-------------+-----------+-----------+-------------
 dalibo      |           | {}        |

# Je veux modifier le paramètre log_lock_waits qui est normalement réservé au superuser
postgres=> SET log_lock_waits = on;
ERROR:  permission denied to set parameter "log_lock_waits"

# Je donne le droit au rôle dalibo de modifier ce paramètre
postgres=# GRANT SET ON PARAMETER log_lock_waits TO dalibo;
GRANT

# L'utilisateur dalibo peut maintenant modifier uniquement ce paramètre
postgres=> SET log_lock_waits = on;
SET
postgres=> show log_lock_waits ;
 log_lock_waits
----------------
 on
```

Avec le privilège `SET`, vous pouvez donner des droits sur l'ensemble des paramètres de PostgreSQL. Cependant, cela n'a de sens que pour les paramètres de contexte `superuser` car les autres ne peuvent soit pas être défini via la commande `SET`, soit peuvent déjà être modifiés par un rôle classique.

Il est possible de déterminer les paramètres réservés aux `superuser` avec la requête suivante :

```
SELECT * FROM pg_settings WHERE context = 'superuser';
```

Pour le privilège `ALTER SYSTEM`, il va permettre de donner le droit à un rôle non `superuser` de réaliser des commandes `ALTER SYSTEM SET ...` sur des paramètres spécifiques. Contrairement à `SET`, il peut s'appliquer à tous les paramètres du fichier `postgresql.conf` :

```
# Toujours avec le rôle dalibo, je veux modifier le shared_buffers de mon instance
postgres=> ALTER SYSTEM SET shared_buffers = '500MB';
ERROR:  permission denied to set parameter "shared_buffers"

# Je donne le droit au rôle dalibo de modifier ce paramètre
postgres=# GRANT ALTER SYSTEM ON PARAMETER shared_buffers TO dalibo;
GRANT

# L'utilisateur dalibo peut maintenant modifier uniquement ce paramètre
postgres=> ALTER SYSTEM SET shared_buffers = '500MB';
ALTER SYSTEM

# La modification a bien été répercutée dans le fichier postgresql.auto.conf
postgres=> \! cat 15/main/postgresql.auto.conf
shared_buffers = '500MB'
```

De façon classique, on utilisera la commande `REVOKE` pour retirer ces droits :

```
REVOKE SET ON PARAMETER log_lock_waits FROM dalibo;
REVOKE ALTER SYSTEM ON PARAMETER shared_buffers FROM dalibo;
```

Même si un utilisateur dispose des droits pour modifier tous les paramètres présents dans le fichier `postgresql.auto.conf`, ce privilège ne donne pas le droit de faire un `ALTER SYSTEM RESET ALL`. Il faudra passer par un super utilisateur ou les annuler un par un.

Afin d'enregistrer la configuration de ces nouveaux privilèges, une nouvelle table système est disponible : `pg_parameter_acl`.

```
postgres=# select * from pg_parameter_acl ;
  oid  |    parname     |                  paracl
-------+----------------+-------------------------------------------
 16394 | shared_buffers | {postgres=sA/postgres,dalibo=A/postgres}
 16390 | log_lock_waits | {postgres=sA/postgres,dalibo=s/postgres}
```

On y retrouve un *OID*, le nom du paramètre et les privilèges par rôle. Concernant les privilèges, deux nouvelles abréviations apparaissent : `s` pour le privilège `SET` et `A` pour `ALTER SYSTEM`.

1.7.2 RÉVOCATION DU DROIT PAR DÉFAUT CREATE SUR LE SCHÉMA PUBLIC POUR LE GROUPE PUBLIC

- USAGE par défaut pour le rôle PUBLIC
- CREATE et USAGE par défaut pour le rôle pg_database_owner
- adaptation de pg_dump pour extraire ces changements
- Attention lors des montées de version !

Cette nouvelle version supprime le privilège par défaut CREATE sur le schéma public pour le rôle PUBLIC. Pour rappel, PUBLIC peut être vu comme un rôle implicitement défini qui inclut en permanence tous les rôles. Le propriétaire par défaut du schéma public n'est plus postgres mais le rôle pg_database_owner. Ce mécanisme permet au propriétaire de la base de données d'obtenir implicitement le droit CREATE sur le schéma public.

Voici ce que ça donne en comparant avec une instance en version 14 :

```
# En version 14
postgres=# \dn+
                            Liste des schémas
  Nom   | Propriétaire |    Droits d'accès    |      Description
--------+--------------+----------------------+------------------------
 public | postgres     | postgres=UC/postgres+| standard public schema
        |              | =UC/postgres         |

# En version 15
postgres=# \dn+
                                      Liste des schémas
  Nom   |   Propriétaire    |            Droits d'accès             |      Description
--------+-------------------+---------------------------------------+------------------------
 public | pg_database_owner | pg_database_owner=UC/pg_database_owner+| standard public schema
        |                   | =U/pg_database_owner                  |
```

On constate bien le changement de propriétaire et la perte de l'abréviation C sur la ligne =U/pg_database_owner qui correspond aux privilèges par défaut du rôle PUBLIC.

Même si la configuration des privilèges est reprise lors d'une montée de version, il convient de réaliser une étape préalable de vérification afin de déterminer d'éventuel impact que pourrait avoir ce changement. Notamment, si un rôle doit créer des objets dans le schéma public, qu'il n'est pas propriétaire de la base de données et, qu'aucun privilège CREATE spécifique n'a été donné car on se basait sur le privilège CREATE qui était implicitement donné au rôle PUBLIC.

Ces changements sur le schéma public ont donné lieu à des ajustements dans l'outil pg_dump. Il extrait désormais correctement le propriétaire du schéma, les privilèges et [*security labels*] sur le schéma public même s'il a été supprimé puis recréé.

1.7.3 AJOUT DE LA POSSIBILITÉ DE CRÉER DES SÉQUENCES UNLOGGED

- Évite de répliquer une séquence d'une table `unlogged`
- Pas dans un but de performance
- Une séquence d'identité hérite automatiquement de la persistance de la table de référence

Il est maintenant possible de définir une séquence comme non journalisée (`unlogged`). Contrairement aux tables, cette option n'est pas destinée à améliorer les performances mais principalement à éviter de répliquer des objets inutilement.

Dorénavant, une séquence identitaire hérite automatiquement de la persistance de la table dont elle dépend.

```
# En version 14
postgres=# create unlogged table journal (id integer GENERATED ALWAYS AS IDENTITY);
CREATE TABLE

# Vérifions la persistance de la séquence associée
postgres=# \ds+
                                    Liste des relations
 Schéma |      Nom       |   Type   | Propriétaire | Persistence |   Taille   | Description
--------+----------------+----------+--------------+-------------+------------+-------------
 public | journal_id_seq | séquence | postgres     | permanent   | 8192 bytes |

# En version 15
postgres=# create unlogged table journal (id integer GENERATED ALWAYS AS IDENTITY);
CREATE TABLE

# Vérification de la persistance
postgres=# \ds+
                                    Liste des relations
 Schéma |      Nom       |   Type   | Propriétaire |  Persistence   | Taille | Description
--------+----------------+----------+--------------+----------------+--------+-------------
 public | journal_id_seq | séquence | postgres     | non journalisé | 16 kB  |
```

Il est également possible de définir manuellement une séquence comme `unlogged` avec les commandes suivantes :

```
CREATE UNLOGGED SEQUENCE ma_seq;
ALTER SEQUENCE ma_seq SET LOGGED|UNLOGGED;
```

Enfin, la persistance des séquences est conservée lors des opérations d'export / import avec des outils comme `pg_dump` et `pg_restore`.

1.7.4 NOUVELLE VARIABLE D'ENVIRONNEMENT PSQL_WATCH_PAGER

- Permet de définir un *pager* pour la commande `\watch`
- Privilégier le pager *pspg*
- Fonctionne uniquement sous Unix

La méta-commande `\watch [durée]` de psql peut être placée juste après un ordre SQL pour le réexécuter à intervalle régulier.

```
[local]:5445 postgres@postgres=# SELECT 'hello world' \watch 1
Thu 18 Aug 2022 02:11:59 PM CEST (every 1s)

  ?column?
-------------
 hello world
(1 row)

Thu 18 Aug 2022 02:12:00 PM CEST (every 1s)

  ?column?
-------------
 hello world
(1 row)
```

Afin de faciliter la lecture du résultat des requêtes exécutées de cette manière, il est maintenant possible de définir un *pager* via la variable d'environnement `PSQL_WATCH_PAGER`.

N'importe quel *pager* peut-être utilisé. Cependant, seul *pspg* semble pour le moment réussir à interpréter correctement le flux renvoyé par la commande `\watch`. Des *pager* traditionnels peuvent être utilisés (`less` par exemple), mais le résultat n'est pas particulièrement pratique à analyser et il finit généralement par être inutilisable.

Pour que *pspg* puisse interpréter correctement le flux envoyé par la commande `\watch`, il faudra utiliser l'option `--stream`. Voici comment le définir :

```
export PSQL_WATCH_PAGER="pspg --stream"
```

1.7.5 COLLATION ICU DÉCLARÉES GLOBALEMENT

- Définition des collations ICU pour toute l'instance ou une base de données
- `initdb` et `createdb`
  - `--locale-provider={icu|libc}`
  - `--icu-locale=LOCALE`
- `CREATE DATABASE .. LOCALE_PROVIDER [icu,libc] ICU_LOCALE "LOCALE"`
- Contrôle des versions de collation par base de données
  - colonne `pg_database.datcollversion`
  - fonction `pg_database_collation_actual_version`
  - mise à jour : `ALTER DATABASE .. REFRESH COLLATION VERSION`

Le support pour l'utilisation des collations ICU a été ajouté en version 10 de PostgreSQL pour éviter d'être dépendant de la bonne gestion des mises à jour de la bibliothèque glibc. En effet, un changement de sa version peut modifier l'ordre de certains tris, et donc peut changer le résultat d'un `SELECT … ORDER BY …` ou l'ordre des clés dans les index. En raison de ce risque de corruption, une réindexation est nécessaire en cas de restauration ou de promotion d'une instance secondaire sur un serveur avec une version de glibc différente.

ICU[4] est une bibliothèque qui permet une gestion standardisée des collations. Cela permet d'éviter les problèmes décrits précédemment en versionnant les collations et en permettant aux administrateurs de choisir quand/si ils changent de collation. Elles permettent aussi d'ajouter des fonctionnalités comme la possibilité d'ordonner les résultats en respectant ou non la casse et les accents (disponible avec PostgreSQL 12). Pour finir, les collations ICU permettent à PostgreSQL d'utiliser les *abbreviate keys* dans les index, ce qui permet notamment d'accélérer la création des index[5] .

La version 10 de PostgreSQL permet d'ajouter les collations ICU aux collations disponibles sur une instance.

```
SELECT CASE collprovider
         WHEN 'i' THEN 'icu'
         WHEN 'd' THEN 'default'
         WHEN 'c' THEN 'libc'
         ELSE 'N/A'
       END as provider, collversion, count(*)
  FROM pg_collation
 GROUP BY 1,2
 ORDER BY 1,2 NULLS FIRST;

 provider | collversion | count
```

[4] https://icu.unicode.org/home
[5] https://blog.anayrat.info/2017/11/19/postgresql-10-icu-abbreviated-keys/

```
----------+-------------+-------
 default  | ¤           |     1
 icu      | 153.14      |   461
 icu      | 153.14.39   |   324
 libc     | ¤           |  1009
(4 rows)
```

L'ajout des collations au catalogue se fait soit lors de la création de l'instance, soit grâce à la commande CREATE COLLATION.

```
CREATE COLLATION capitalfirst (PROVIDER=icu, LOCALE='en-u-kf-upper');
```

Si la collation est modifiée, le message d'erreur suivant est visible dans les traces lorsque la collation est utilisée :

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but
         the operating system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION
       pg_catalog."xx-x-icu" REFRESH VERSION, or build PostgreSQL with the
       right library version.
```

Cela permet d'introduire une vérification de la version de la collation utilisée pour les objets dépendants d'une collation spécifique. La requête suivante permet de remonter les objets qui dépendent d'une collation dont la version a été mise à jour.

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

Les objets concernés doivent être reconstruits avant de mettre à jour la version de la collation avec la commande :

```
ALTER COLLATION .. REFRESH COLLATION VERSION;
```

Cette commande ne fait que mettre à jour la collation, elle n'effectue aucun contrôle sur les objets pour vérifier qu'ils ont bien été reconstruits.

En version 10, Le choix de la collation ICU ne peut être spécifié que dans la clause COLLATE, ce qui rend cette fonctionnalité difficile à utiliser.

Exemple :

```
CREATE TABLE t1 (t text COLLATE "en-US-x-icu");
ALTER TABLE t1 ALTER t TYPE text COLLATE "fr-FR-x-icu";
CREATE INDEX ON t1(t COLLATE "fr-BE-x-icu");
```

La version 13 ajoute la possibilité de versionner les collations fournies par la glibc avec la version de cette librairie. La gestion des versions des collations sous Windows est également supporté.

```
 provider | collversion | count
----------+-------------+-------
 default  | ¤           |     1
 icu      | 153.14      |   461
 icu      | 153.14.39   |   324
 libc     | ¤           |     4
 libc     | 2.34        |  1005
(5 rows)
```

La version 14 permet de supporter la gestion des versions sur FREEBSD.

La version 15 de PostgreSQL rend possible l'utilisation d'ICU pour gérer les collations pour l'ensemble de l'instance ou d'une base de données.

Les commandes `initdb` et `createdb` disposent désormais de deux nouvelles option `--locale-provider={icu|libc}` et `--icu-locale=LOCALE` pour spécifier la collation ICU utilisée pour les nouvelles instances.

```
initdb --locale-provider=icu --icu-locale=fr-FR-x-icu --locale=fr_FR.UTF-8 \
       --data-checksum /var/lib/postgresql/15/data

The files belonging to this database system will be owned by user "benoit".
This user must also own the server process.

The database cluster will be initialized with this locale configuration:
  provider:    icu
  ICU locale:  fr-FR-x-icu
  LC_COLLATE:  fr_FR.UTF-8
  LC_CTYPE:    fr_FR.UTF-8
  LC_MESSAGES: fr_FR.UTF-8
  LC_MONETARY: fr_FR.UTF-8
  LC_NUMERIC:  fr_FR.UTF-8
  LC_TIME:     fr_FR.UTF-8
The default text search configuration will be set to "french".
...
```

Lorsque le fournisseur de collation de l'instance est `libc`, il faut bien penser à préciser la base modèle `template0` pour la création d'une base avec le fournisseur `icu` :

```
# createdb --locale-provider=icu --icu-locale=fr-FR-x-icu \
           --template template0 --echo dbtest

SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE dbtest TEMPLATE template0 LOCALE_PROVIDER icu ICU_LOCALE 'fr-FR-x-icu';
```

On peut voir que la commande `CREATE DATABASE` s'est vu ajouter deux nouveaux paramètres : `LOCALE_PROVIDER` et `ICU_LOCALE`.

Pour finir, il est désormais possible de voir la version de la collation utilisée pour une base de données grâce la nouvelle colonne `datcollversion` de la vue `pg_database` ainsi que la version de la collation présente au niveau du système grâce à la fonction `pg_database_collation_actual_version()`.

```
SELECT datname, datlocprovider, datcollate, datctype,
       datcollversion,
       pg_database_collation_actual_version(oid) AS actualcolversion
  FROM pg_database
 WHERE datname = 'iso88591' \gx

-[ RECORD 1 ]----+---------------
datname          | iso88591
datlocprovider   | c
datcollate       | fr_FR.iso88591
datctype         | fr_FR.iso88591
datcollversion   | 2.34
actualcolversion | 2.34
```

Si les versions sont différentes, les index doivent être reconstruits avant de rafraîchir la version de la collation pour la base concernée avec la commande suivante :

```
ALTER DATABASE .. REFRESH COLLATION VERSION;
```

1.7.6 AJOUT DE L'OPTION --CONFIG-FILE À PG_REWIND

- Nouvelle option `-C/--config-file`
- Permet l'utilisation de l'option `-c/--restore-target-wal` quand la configuration de PostgreSQL n'est pas dans `$PGDATA`.

L'option `-c/--restore-target-wal` ajoutée en version 13 permet d'utiliser la commande de restauration des archives (`restore_command`) stockées dans le fichier de configuration de l'instance pour récupérer les WAL nécessaires à l'opération de *rewind*, s'ils ne sont plus dans le répertoire `pg_wal`.

Ce mode de fonctionnement pose problème pour les installations ou les fichiers de configuration de PostgreSQL ne sont pas stockés dans le répertoire de données de l'instance. C'est par exemple le cas par défaut sur les installations DEBIAN. Sur ce genre d'installation, le fichier de configuration doit être copié dans le répertoire de données manuellement avant de lancer l'opération de *rewind*. Cela peut également complexifier l'implémentation d'outils de haute disponibilité avec reconstruction automatique comme Patroni[6] .

[6] https://github.com/zalando/patroni/pull/2225

L'option `-C/--config-file` permet de donner à pg_rewind le chemin du fichier de configuration. Il sera ensuite utilisé par pg_rewind lors du démarrage de PostgreSQL (option `-C` du postmaster) :

- afin d'obtenir la configuration de la commande de restauration ;
- afin d'arrêter PostgreSQL proprement avant la réalisation du *rewind*.

---

2 PERFORMANCES

2.1 EXÉCUTION EN PARALLÈLE DES REQUÊTES SELECT DISTINCT

- Parallélisation des clauses `DISTINCT` en deux phases :
  - première phase de déduplication (parallélisée)
  - seconde phase d'agrégation et de déduplication des résultats de la première phase

Depuis la version 9.6, les agrégations peuvent être parallélisées par PostgreSQL. Cette fonctionnalité ne pouvait cependant pas être utilisée pour la clause `DISTINCT`.

Voici un jeu d'essais qui permet de mettre en évidence le comportement de PostgreSQL :

```
CREATE TABLE test_distinct(i int);
INSERT INTO test_distinct SELECT (random()*10000)::int FROM generate_series(1, 1000000);
ANALYZE test_distinct;
```

Le plan de la requête suivante en version 14 est alors :

```
EXPLAIN (ANALYZE) SELECT DISTINCT i FROM test_distinct;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=16925.00..17024.68 rows=9968 width=4)
                (actual time=1001.069..1006.089 rows=10001 loops=1)
   Group Key: i
   Batches: 1  Memory Usage: 913kB
   ->  Seq Scan on test_distinct  (cost=0.00..14425.00 rows=1000000 width=4)
                                  (actual time=0.049..278.769 rows=1000000 loops=1)
 Planning Time: 0.260 ms
 Execution Time: 1007.694 ms
(6 rows)
```

La version 15 découpe la déduplication en deux phases. La première phase, qui peut être parallélisée, permet de rendre les lignes distinctes avec un `sort unique` ou un `hashaggregate`. Le résultat produit par les processus parallélisés sont combinés et rendu distinct a nouveau dans une seconde phase.

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=12783.76..12883.78 rows=10002 width=4)
                (actual time=346.312..350.351 rows=10001 loops=1)
   Group Key: i
   Batches: 1  Memory Usage: 913kB
   ->  Gather  (cost=10633.33..12733.75 rows=20004 width=4)
```

```
              (actual time=316.791..328.705 rows=30003 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  HashAggregate  (cost=9633.33..9733.35 rows=10002 width=4)
                           (actual time=308.740..312.559 rows=10001 loops=3)
              Group Key: i
              Batches: 1  Memory Usage: 913kB
              Worker 0:  Batches: 1  Memory Usage: 913kB
              Worker 1:  Batches: 1  Memory Usage: 913kB
              ->  Parallel Seq Scan on test_distinct  (cost=0.00..8591.67 rows=416667 width=4)
                                                      (actual time=0.033..79.344 rows=333333 loop
 Planning Time: 0.222 ms
 Execution Time: 351.886 ms
(14 rows)
```

La méthode employée par PostgreSQL pour rendre les lignes distinctes dépend de la répartition des données.

Par exemple, en régénérant les données avec 1000 valeurs différentes au lieu de 10000 :

```
TRUNCATE test_distinct;
INSERT INTO test_distinct SELECT (random()*1000)::int FROM generate_series(1, 1000000);
ANALYSE test_distinct;
```

On voit que PostgreSQL utilise un distinct pour la seconde phase :

```
[local]:5437 postgres@postgres=# EXPLAIN (ANALYZE) SELECT DISTINCT i FROM test_distinct ;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Unique  (cost=10953.33..10963.34 rows=1001 width=4)
         (actual time=280.039..284.113 rows=1001 loops=1)
   ->  Sort  (cost=10953.33..10958.33 rows=2002 width=4)
             (actual time=280.035..283.177 rows=3003 loops=1)
         Sort Key: i
         Sort Method: quicksort  Memory: 97kB
         ->  Gather  (cost=10633.33..10843.54 rows=2002 width=4)
                     (actual time=277.377..281.096 rows=3003 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  HashAggregate  (cost=9633.33..9643.34 rows=1001 width=4)
                                  (actual time=270.683..271.037 rows=1001 loops=3)
                     Group Key: i
                     Batches: 1  Memory Usage: 129kB
                     Worker 0:  Batches: 1  Memory Usage: 129kB
                     Worker 1:  Batches: 1  Memory Usage: 129kB
                     ->  Parallel Seq Scan on test_distinct  (cost=0.00..8591.67 rows=416667 width
                                                             (actual time=0.036..89.688 rows=33333
 Planning Time: 0.201 ms
```

```
 Execution Time: 284.301 ms
(15 rows)
```

2.2 PG_STAT_STATEMENTS

- Nouvelles statistiques ajoutées dans `pg_stat_statements` pour tracer :
  - l'activité de JIT
  - les temps d'accès aux fichiers temporaires
- L'extension passe en 1.10

La vue [`pg_stat_statement`], fournie avec l'extension éponyme, s'est vue ajouter une série de compteurs permettant de suivre l'activité du compilateur à la volée (JIT) sur un serveur. Ces informations sont très intéressantes car jusqu'à maintenant, il n'y avait aucun moyen de superviser l'utilisation de JIT. Bien souvent, les seules fois où l'on entendait parler du JIT étaient quand les temps de planifications pénalisaient le temps d'exécution de la requête.

Voici la liste des compteurs qui sont cumulés, comme les autres informations de cette vue :

- `jit_functions` : nombre total de fonctions compilées par JIT pour cette requête ;
- `jit_generation_time` : temps total consacré à générer du code JIT pour cette requête, il est exprimé en millisecondes ;
- `jit_inlining_count` : nombre de fois où les fonctions ont été incluses ;
- `jit_inlining_time` : temps total consacré à l'inclusion des fonctions pour cette requête, il est exprimé en millisecondes ;
- `jit_optimization_count` : nombre de requêtes qui ont été optimisées ;
- `jit_optimization_time` : temps total consacré à l'optimisation pour cette requête, il est exprimé en millisecondes ;
- `jit_emission_count` : nombre de fois où du code a été émis ;
- `jit_emission_time` : temps total consacré à émettre du code, il est exprimé en millisecondes.

Des informations concernant les temps d'accès aux fichiers temporaires ont également été ajoutées :

- `temp_blk_read_time` : temps total consacré à la lecture de blocs de fichiers temporaires, il est exprimé en millisecondes. Ce paramètre est valorisé à zéro si `track_io_timing` est désactivé.
- `temp_blk_write_time` : temps total consacré à écrire des blocs de fichiers temporaires, il est exprimé en millisecondes. Ce paramètre est valorisé à zéro si

`track_io_timing` est désactivé.

Suite à l'ajout de ces fonctionnalités, l'extension passe en version 1.10.

---

3 RÉPLICATION LOGIQUE

3.1 NOUVELLE OPTION `TABLES IN SCHEMA`

- Permet de publier toutes les tables d'un schéma
- Possibilité de mixer tables et schéma
- Nouvelle table système `pg_publication_namespace` qui référence les schémas à publier

Cette nouvelle version introduit la possiblité de publier la totalité des tables d'un schéma. Elle permet via les ordres `CREATE / ALTER PUBLICATION` de définir un ou plusieurs schémas pour lesquels toutes les tables seront incluses dans la publication (également les tables créées dans le futur).

```
CREATE PUBLICATION p1 FOR TABLES IN SCHEMA s1,s2;

ALTER PUBLICATION p1 ADD TABLES IN SCHEMA s3;
```

Comme pour la clause `ALL TABLES`, la clause `TABLES IN SCHEMA` nécessite d'utiliser un role ayant l'attribut `superuser`.

Il est également possible de mixer tables et schémas dans une publication :

```
CREATE PUBLICATION p2 FOR TABLE t1,t2,t3, TABLES IN SCHEMA s1;

ALTER PUBLICATION p2 ADD TABLE t4, TABLES IN SCHEMA s2;
```

Les schémas ajoutés dans une publication sont stockés dans une nouvelle vue système `pg_publication_namespace`, qui va contenir les `oid` des publications et des schémas publiés.

```
SELECT pubname, pnnspid::regnamespace
FROM pg_publication_namespace pn, pg_publication p
WHERE pn.pnpubid = p.oid;

 pubname | pnnspid
---------+---------
 p1      | s1
 p1      | s2
```

Le plugin `pgoutput` a également été modifié pour prendre en compte ce changement. Maintenant, en cas d'utilisation de la clause `TABLES IN SCHEMA`, il va vérifier si la relation fait bien partie des schémas inclus dans la publication avant l'envoi des changements au souscripteur.

La commande \dRp+ a été mise à jour pour prendre en compte cette nouvelle fonctionnalité. Elle affiche donc la liste des schémas associés à une publication :

```
postgres=# \dRp+ p1

Publication pub1
Owner     | All tables | Inserts | Updates | Deletes | Truncates | Via root
----------+------------+---------+---------+---------+-----------+----------
postgres  | f          | t       | t       | t       | f         | f

Tables from schemas:
"s1"
"s2"
```

Pour finir, pg_dump a également été mis à jour pour identifier si une publication inclut la clause TABLES IN SCHEMA. Voici un exemple de ce que génère pg_dump :

```
--
-- Name: p1; Type: PUBLICATION; Schema: -; Owner: postgres
--
CREATE PUBLICATION p1 WITH (publish = 'insert, update, delete');
ALTER PUBLICATION p1 OWNER TO postgres;
--
-- Name: p1 s1; Type: PUBLICATION TABLES IN SCHEMA; Schema: s1; Owner: postgres
--
ALTER PUBLICATION p1 ADD TABLES IN SCHEMA s1;
```

3.2 LES DONNÉES PUBLIÉES PEUVENT ÊTRE FILTRÉES

- Ajout de la clause WHERE pour filtrer les données d'une table à publier
- Filtre uniquement par table
- Pas de restriction de colonne pour l'opération INSERT
- Restrictions pour les opérations UPDATE et DELETE (colonnes couvertes par REPLICA IDENTITY)
- Ne fonctionne qu'avec des expressions simples (y compris fonctions de base et opérateurs logiques)

Autre grosse nouveauté pour la réplication logique, il est maintenant possible d'appliquer un filtre pour ne répliquer que partiellement une table. Ce filtre est géré par l'option WHERE au niveau de la publication et est spécifique à une table.

```
CREATE PUBLICATION p1 FOR TABLE t1 WHERE (ville = 'Reims');
```

Cette clause `WHERE` n'autorise que des expressions simples (y compris les fonctions de base et les opérateurs logiques) et ne peut faire référence qu'aux colonnes de la table sur laquelle la publication est mise en place. Il n'est pour le moment pas possible d'utiliser de fonctions, d'opérateurs, de types et de collations qui sont définis par un utilisateur. Les fonctions non immutables et les colonnes système sont également inutilisables.

Si une publication ne publie que des ordres INSERT, il n'y a pas de limitation sur les colonnes utilisées dans le filtre. En revanche, si la publication concerne les ordres UPDATE et DELETE, il faut que les colonnes du filtre fassent partie de l'identité de réplica. Cela signifie que ces colonnes doivent faire partie de la clé primaire si [`REPLICA IDENTITY`](https://www.postgresql.org/docs/current/logical-replication-publication.htm est laissé à sa valeur par défaut. Si un index unique est créé et utilisé pour la clause `REPLICA IDENTITY USING INDEX`, ces colonnes doivent en faire partie. Enfin, si `REPLICA IDENTITY` est valorisé à `FULL`, n'importe quelle colonne peut faire partie du filtre. Cette dernière configuration est cependant peu performante.

Si on déclare un filtre sur une colonne qui ne fait pas partie de l'identité de réplica, PostgreSQL refuse les mises à jour et renvoie le message suivant. Il faut donc être très vigilant lors de la mise en place d'un filtre sur une publication.

```
ERROR:  cannot update table "rep"
DETAIL:  Column used in the publication WHERE expression is not part of the replica identity.

ERROR:  cannot delete from table "rep"
DETAIL:  Column used in the publication WHERE expression is not part of the replica identity.
```

Voici comment est mis en place le filtre :

- il est appliqué **avant** de décider de la publication d'une modification ;
- si la validation du filtre renvoie `NULL` ou `false`, la modification ne sera pas publiée ;
- les `TRUNCATE` sont ignorés puisqu'ils modifient l'ensemble de la table et sont répliqués depuis PostgreSQL 11 ;
- les `INSERT` et les `DELETE` sont répliqués normalement du moment que le filtre est validé ;
- les UPDATE sont plus compliqué. Les exemples suivant décrivent les trois cas de figure et la façon dont PostgreSQL les gère.

```
# Sur le publieur

# Création de la table rep
pub=# CREATE TABLE rep (i INT PRIMARY KEY);
CREATE TABLE
```

```
# Insertion de données
pub=# INSERT INTO rep SELECT generate_series(1,10);
INSERT 0 10

# Création de la publication avec filtre
pub=# CREATE PUBLICATION p1 FOR TABLE rep WHERE (i > 5);
CREATE PUBLICATION

# Sur le souscripteur

# Création de la table rep
sub=# CREATE TABLE rep (i INT PRIMARY KEY);
CREATE TABLE

# Mise en place de la souscription
sub=# CREATE SUBSCRIPTION s1
sub-# CONNECTION 'host=/var/run/postgresql port=5449 user=postgres dbname=pub'
sub-# PUBLICATION p1
sub-# WITH (copy_data = true);
CREATE SUBSCRIPTION

# Vérification des données
sub=# SELECT * FROM rep;
 i
----
 6
 7
 8
 9
 10
```

Réalisation d'un UPDATE où l'ancienne et la nouvelle version de ligne valident le filtre.

```
# Sur le publieur
pub=# UPDATE rep SET i = 20 WHERE i = 10;

# Sur le souscripteur
sub=# SELECT * FROM rep;
 i
----
 6
 7
 8
 9
 20

# Dans la log du souscripteur au niveau DEBUG5
CONTEXT: processing remote data for replication origin "pg_16395" during message
```

```
          type "UPDATE" in transaction 766, finished at 2/D602FC30
```

Dans le cas contraire, voici ce qui se passe :

- si l'ancienne version de la ligne ne valide pas le filtre (donc que la ligne n'existe pas sur le souscripteur), un `INSERT` sera envoyé au souscripteur plutôt qu'un `UPDATE` ;

```
# Sur le publieur
pub=# UPDATE rep SET i = 15 WHERE i = 1;

# Sur le souscripteur
sub=# SELECT * FROM rep;
  i
----
  6
  7
  8
  9
  20
  15

# Dans la log du souscripteur au niveau DEBUG5
CONTEXT:  processing remote data for replication origin "pg_16395" during message
          type "INSERT" in transaction 767, finished at 2/D602FD28
```

- dans le cas contraire, si la nouvelle ligne ne valide pas le filtre (donc que la ligne ne doit plus exister sur le souscripteur), un `DELETE` remplacera l'`UPDATE`.

```
# Sur le publieur
pub=# UPDATE rep SET i = 0 WHERE i = 6;

# Sur le souscripteur
sub=# SELECT * FROM rep;
  i
----
  7
  8
  9
  20
  15

# Dans la log du souscripteur au niveau DEBUG5
CONTEXT:  processing remote data for replication origin "pg_16395" during message
          type "DELETE" in transaction 768, finished at 2/D602FE20
```

Si l'option `copy_data = true` est utilisée lors du `CREATE SUBSCRIPTION`, seules les données préexistantes satisfaisant le filtre seront répliquées durant la copie initiale des don-

nées. Si le souscripteur est dans une version inférieure à la 15, l'initialisation des données se fera sans utilisation du filtre. Les lignes publiées par la suite seront correctement filtrées.

Il est également possible d'avoir pour une même souscription, plusieurs publications pour une même table avec des filtres différents. Dans un tel cas, ces filtres seront combinés avec un `OR` de sorte que les modifications qui répondent à n'importe quel filtre seront répliquées.

```
# Sur le publieur

pub=# SELECT * FROM rep;
 i
----
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

# Création première publication
pub=# CREATE PUBLICATION pub1 FOR TABLE rep WHERE (i < 4);
CREATE PUBLICATION

# Création deuxième publication
pub=# CREATE PUBLICATION pub2 FOR TABLE rep WHERE (i > 7);
CREATE PUBLICATION

# Sur le souscripteur

# Création de la première souscription
sub=# CREATE SUBSCRIPTION sub1
sub-# CONNECTION 'host=/var/run/postgresql port=5449 user=postgres dbname=pub'
sub-# PUBLICATION pub1
sub-# WITH (copy_data = true);
CREATE SUBSCRIPTION

# Création de la deuxième souscription
sub=# CREATE SUBSCRIPTION sub2
sub-# CONNECTION 'host=/var/run/postgresql port=5449 user=postgres dbname=pub'
sub-# PUBLICATION pub2
```

```
sub-# WITH (copy_data = true);
CREATE SUBSCRIPTION

sub=# SELECT * FROM rep;
 i
---
 1
 2
 3
 8
 9
 10
```

Afin de faciliter l'administration, les méta-commandes `\d` et `\dRp+` ont été modifiées pour prendre en compte la mise en place d'un filtre pour une publication :

```
pub=# \dRp+
                                    Publication pub1
 Propriétaire | Toutes les tables | Insertions | Mises à jour | Suppressions | Tronque | Via la racine
--------------+-------------------+------------+--------------+--------------+---------+---------------
 postgres     | f                 | t          | t            | t            | t       | f
Tables :
    "public.rep" WHERE (i < 4)

                                    Publication pub2
 Propriétaire | Toutes les tables | Insertions | Mises à jour | Suppressions | Tronque | Via la racine
--------------+-------------------+------------+--------------+--------------+---------+---------------
 postgres     | f                 | t          | t            | t            | t       | f
Tables :
    "public.rep" WHERE (i > 7)

pub=# \d rep
                  Table « public.rep »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 i       | integer |                 | not null  |
Index :
    "rep1_pkey" PRIMARY KEY, btree (i)
Publications :
    "pub1" WHERE (i < 4)
    "pub2" WHERE (i > 7)
```

Concernant les tables partitionnées publiées, l'application du filtre dépendra du paramètre `publish_via_partition_root`. Si celui-ci est à `true`, le filtre sera appliqué pour chaque partition. Par contre, s'il est à `false` (qui est la valeur par défaut), il ne sera pas appliqué sur la table partitionnée.

3.3 AJOUT DE LA VUE SYSTÈME PG_STAT_SUBSCRIPTION_STATS POUR REPORTER L'ACTIVITÉ D'UN SOUSCRIPTEUR

- Donne des informations sur les erreurs qui se sont produites durant la réplication logique
- Ajout de la fonction `pg_stat_reset_subscription_stats()`

La vue système `pg_stat_subscription_stats` permet de récupérer des informations sur les erreurs qui se sont produisent au niveau des souscriptions avec la réplication logique. Ces données sont stockées sous forme de compteur et concernent les erreurs rencontrées lors de l'application des changements ou lors de la synchronisation initiale. Elle contient une ligne par souscription.

Voici la description des colonnes de cette vue :

- `subid` : *OID* de la souscription ;
- `subname` : nom de la souscription ;
- `apply_error_count` : nombre d'erreurs rencontrées lors de l'application des changements ;
- `sync_error_count` : nombre d'erreurs rencontrées lors de la synchronisation initiale des tables ;
- `stats_reset` : date de réinitialisation des statistiques.

La fonction `pg_stat_reset_subscription_stats` permet de réinitialiser les statistiques de la vue `pg_stat_subscription_stats`. Elle prend en paramètre soit l'*OID* d'une souscription pour ne réinitialiser que les statistiques de cette dernière, soit `NULL` pour appliquer la réinitialisation à **toutes** les souscriptions.

---

4 DÉVELOPPEMENT + CHANGEMENT SYNTAXE SQL

4.1 AJOUT DE LA COMMANDE SQL MERGE

- Insérer, mettre à jour ou supprimer des lignes conditionnellement en un seul ordre SQL.

Fonctionnement de la commande MERGE

La commande MERGE permet d'insérer, mettre à jour ou supprimer des lignes conditionnellement en un seul ordre SQL.

Voici un exemple d'utilisation de cette commande :

```
CREATE TABLE mesures_capteurs (
  id INT PRIMARY KEY,
  top_mesure INT,
  derniere_mesure INT,
  derniere_maj TIMESTAMP WITH TIME ZONE
);

INSERT
  INTO mesures_capteurs
  VALUES (1, 10, 10, current_timestamp - INTERVAL  '1 day'),
         (2,  5,  5, current_timestamp - INTERVAL '11 day'),
         (3, 20, 20, current_timestamp - INTERVAL  '1 day'),
         (4, 15, 15, current_timestamp - INTERVAL  '1 day');

CREATE TABLE import_mesures_capteurs (
  id INT,
  mesure INT
);

INSERT
  INTO import_mesures_capteurs(id, mesure)
  VALUES (2, 15), -- supprimer la ligne dans la table mesures_capteurs (trop ancienne)
         (3, 10), -- ne pas mettre a jour le top
         (4, 16), -- mettre a jour le top
         (5, 19); -- insérer la ligne

BEGIN;

MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
```

```
  -- insérer les nouvelles lignes
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <= current_timestamp ) THEN
  -- supprimer les mesures de capteurs si l'ancienne mesure date de plus de 10 jours
  DELETE
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  -- mettre à jour seulement la mesure
  UPDATE
  SET derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
WHEN MATCHED THEN
  -- mettre à jour le top et la mesure
  UPDATE
  SET top_mesure = i.mesure,
      derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
;

TABLE mesures_capteurs;

ROLLBACK;
```

Le résultat correspond à l'attendu :

- la ligne 1 n'a pas changée ;
- la ligne 2 a été supprimée car elle est trop ancienne ;
- la ligne 3 a été mise à jour (colonnes `derniere_mesure` et `derniere_maj`) ;
- la ligne 4 a été mise à jour (colonnes `top_mesures`, `derniere_mesure` et `derniere_maj`) ;
- la ligne 5 a été insérée.

```
TABLE mesures_capteurs ;
```

```
 id | top_mesure | derniere_mesure |         derniere_maj
----+------------+-----------------+------------------------------
  1 |         10 |              10 | 2022-12-13 16:29:55.671426+01
  3 |         20 |              10 | 2022-12-14 16:30:01.658568+01
  4 |         16 |              16 | 2022-12-14 16:30:01.658568+01
  5 |         19 |              19 | 2022-12-14 16:30:01.658568+01
(4 rows)
```

Le prototype de la commande est le suivant :

```
[ WITH with_query [, ...] ]
MERGE INTO target_table_name [ [ AS ] target_alias ]
USING data_source ON join_condition
when_clause [...]
```

Ou *data_source* est :

```
{ source_table_name | ( source_query ) } [ [ AS ] source_alias ]
```

Lors de son exécution, la commande commence par réaliser une jointure entre la source de donnée et la cible.

- la source de donnée peut être une table, une requête ou une CTE ;
- la table cible ne peut pas être une vue matérialisée, une table étrangère ou la cible de la définition d'une règle[7] ;
- la condition de jointure ne doit contenir que des colonnes des tables source et cible qui participent à la jointure ;
- la jointure ne doit produire qu'une ligne pour chaque ligne candidate. C'est à dire qu'à chaque ligne de la cible, il ne doit correspondre qu'une ligne dans la source. Si ce n'est pas le cas, la première ligne sera utilisée pour modifier la cible et la suivante provoquera une erreur. Ce genre de situation peut également se produire si un TRIGGER insère une ligne qui est ensuite modifiée par la commande MERGE.

Voici un exemple où l'action exécutée sur la seconde ligne est un INSERT :

```
BEGIN;

-- mesure qui déclenche l'insertion d'une nouvelle ligne dans la cible
-- car l'id 5 n'existe pas dans la table mesures_capteurs mais figure déjà
-- dans import_mesures_capteurs
INSERT INTO import_mesures_capteurs VALUES (5,1);

MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
  -- clause WHEN qui insére les nouvelles lignes
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <= current_timestamp ) THEN
  DELETE
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  UPDATE
  SET derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
WHEN MATCHED THEN
  UPDATE
  SET top_mesure = i.mesure,
      derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
```

[7] https://www.postgresql.org/docs/15/sql-createrule.html

```
;

ROLLBACK;

ERROR:  duplicate key value violates unique constraint "mesures_capteurs_pkey"
DETAIL:  Key (id)=(5) already exists.
```

Ce second exemple illustre un cas où c'est l'action `UPDATE` qui est déclenchée deux fois :

```
BEGIN;

-- mesures pour le capteur numéro 3 qui déclenche une mise à jour des
-- colonnes derniere_mesure et derniere_maj
INSERT INTO import_mesures_capteurs VALUES (3,1);

MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <= current_timestamp ) THEN
  DELETE
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  -- clause WHEN déclenchée
  UPDATE
  SET derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
WHEN MATCHED THEN
  UPDATE
  SET top_mesure = i.mesure,
      derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
;
ROLLABCK;

ERROR:  MERGE command cannot affect row a second time
HINT:  Ensure that not more than one source row matches any one target row.
```

La clause *when_clause* de la commande `MERGE` correspond à :

```
{ WHEN MATCHED [ AND condition ] THEN { merge_update | merge_delete | DO NOTHING } |
  WHEN NOT MATCHED [ AND condition ] THEN { merge_insert | DO NOTHING } }
```

Chaque ligne candidate se voit assigner le statut `[NOT] MATCHED` suivant que la jointure a été un succès ou non. Ensuite, les clauses `WHEN` sont évaluées dans l'ordre où elles sont spécifiées. Seule l'action associée à la première clause `WHEN` qui renvoie `vrai` est exécutée.

Si une clause WHEN [NOT] MATCHED est spécifiée sans clause AND, elle sera la dernière clause [NOT] MATCHED de ce type pour la requête. Si une autre clause de ce type est présente après, une erreur est remontée.

Voici un exemple :

```
BEGIN;

MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <= current_timestamp ) THEN
  DELETE
WHEN MATCHED THEN
  UPDATE
  SET top_mesure = i.mesure,
      derniere_mesure = i.mesure,
      derniere_maj = current_timestamp
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  -- clause WHEN qui provoque l'erreur
  UPDATE
  SET derniere_mesure = i.mesure,
      derniere_maj = current_timestamp;

ROLLBACK;
```

```
ERROR:  unreachable WHEN clause specified after unconditional WHEN clause
```

Les clauses merge_insert, merge_update et merge_delete correspondent respectivement à :

```
INSERT [( column_name [, ...] )]
[ OVERRIDING { SYSTEM | USER } VALUE ]
{ VALUES ( { expression | DEFAULT } [, ...] ) | DEFAULT VALUES }
```

```
UPDATE SET { column_name = { expression | DEFAULT } |
 ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
```

‘DELETE

Lorsqu’elles sont exécutées, ces actions ont les mêmes effets que des ordres INSERT, UPDATE ou DELETE classiques. La syntaxe est similaire, à la différence prêt qu’il n’a ni clause FROM ni clause WHERE. Les actions agissent sur la cible, utilisent les lignes courantes de la jointure et agissent sur la cible. Il est possible de spécifier DO NOTHING si on souhaite

ignorer la ligne en cours. Ce résultat peut également être obtenu si aucune clause n'est évaluée à `vrai`.

De la même manière que pour un ordre `INSERT` classique, il est possible de forcer des valeurs pour les colonnes auto-générées en plaçant la clause `OVERRIDING {SYSTEM | USER} VALUE` juste avant la clause VALUES de l'`INSERT`.

Les actions `INSERT`, `UPDATE` et `DELETE` ne contiennent pas de clause `RETURNING`, la commande `MERGE` n'en dispose donc pas non plus.

Privilèges

Les privilèges nécessaires pour exécuter la commande `MERGE` sont les mêmes que pour exécuter les commandes `INSERT`, `UPDATE` et `DELETE` implémentées dans le `MERGE` sur la table cible (ou ses colonnes). Il est également nécessaire d'avoir le droit en lecture sur la table source et toutes les colonnes de la table cible présentes dans les prédicats.

MERGE et triggers

La commande `MERGE` fonctionne également avec les *triggers* :

- `BEFORE STATEMENT`, qui se déclenchent pour toutes les actions spécifiées dans l'ordre `MERGE` qu'elles soient exécutées ou non ;
- `BEFORE ROW`, qui se déclenchent après qu'une clause `WHEN` soit validée mais avant que l'action ne soit exécutée ;
- `AFTER ROW`, qui se déclenchent après qu'une action ait été exécutée ;
- `AFTER STATEMENT`, qui sont exécutés après l'évaluation des clauses `WHEN` pour toutes les actions spécifiées qu'elles aient été exécutées ou non.

INSERT ON CONFLICT vs MERGE

La version 9.5 a vu l'arrivée de la commande `INSERT ON CONFLICT` qui permet d'exécuter une action lorsque une erreur de violation de contrainte d'unicité ou d'exécution est détectée. Le cas d'utilisation le plus fréquent est la réalisation d'un `UPSERT` (`INSERT` ou `UPDATE` atomique). On remarque ici que les fonctionnalités couvertes par la commande `MERGE` se recoupe en partie mais pas totalement.

Les actions disponibles pour `INSERT ON CONFLICT` sont de deux types : `UPDATE` ou `DO NOTHING`. Là encore, il y a une différence avec la commande `MERGE` qui permet en plus de faire des suppressions.

`ON CONFLICT UPDATE` garantit l'exécution atomique d'un `INSERT` ou d'un `UPDATE` même en cas de forte concurrence d'accès. La commande `MERGE` n'a pas ce genre de garantie. Si une commande `INSERT` est exécutée en même temps que le `MERGE`, il est possible que le `MERGE` ne la voit pas et choisisse d'utiliser l'action `INSERT`, ce qui aboutira à une erreur de

violation de contrainte d'unicité. C'est la raison pour laquelle la commande `MERGE` avait été initialement refusée et remplacée par `INSERT ON CONFLICT`.

Pour finir, la façon dont les lignes sont sélectionnées pour réaliser une action est différente. `INSERT ON CONFLICT` permet de spécifier la ou les colonnes d'un index avec la clause `UNIQUE` ou une contrainte. Si un conflit est détecté, l'action spécifiée est exécutée. La commande `MERGE` joint les lignes de deux tables afin de déterminer s'il y a une correspondance et permet ensuite de réaliser des filtres sur d'autres colonnes des tables afin de décider quelle action exécuter. Le mécanisme est donc très différent.

Les deux commandes peuvent donc être utilisées pour réaliser des opérations similaires mais ne sont pas interchangeables.

4.2 PERMETTRE L'USAGE D'INDEX POUR LES CONDITION BASÉES SUR ^@ ET STARTS_WITH()

- `starts_with()` et son opérateur `^@` sont indexables directement avec un btree avec une collation C
  - fonctionnement similaire à `LIKE 'chaine%'`
  - conversion en deux prédicats `>=`, `<`
- Si un index SPGist utilisable existe, `LIKE 'chaine%'` est transformé en `^@`

Le planificateur est désormais capable de traiter la fonction `stars_with()` et l'opérateur équivalent `^@` de la même manière que l'expression `chaine LIKE 'foo%'`. Le prédicat est transformé en deux conditions `>=` et `<` qui sont indexables si la collation est `C`.

Voici un exemple :

```
CREATE TABLE startswith(t text);
INSERT INTO startswith SELECT 'commence par ' || i FROM generate_series(1,10000) AS  F(i);
CREATE INDEX ON startswith USING btree (t COLLATE "C");
ANALYZE startswith;

EXPLAIN (COSTS OFF)
  SELECT *
    FROM startswith
   WHERE starts_with(t, 'commence par 1');

                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on startswith
   Filter: starts_with(t, 'commence par 1'::text)
   ->  Bitmap Index Scan on startswith_t_idx
```

```
         Index Cond: ((t >= 'commence par 1'::text) AND (t < 'commence par 2'::text))
(4 rows)
```

On voit dans l'exemple suivant que si la collation est différente de `C`, l'usage de l'index est impossible.

```
DROP INDEX startswith_t_idx ;
CREATE INDEX ON startswith USING btree (t COLLATE "fr_FR.utf8");

EXPLAIN (COSTS OFF)
  SELECT *
    FROM startswith
   WHERE starts_with(t, 'commence par 1');
```

```
                    QUERY PLAN
--------------------------------------------------
 Seq Scan on startswith
   Filter: starts_with(t, 'commence par 1'::text)
(2 rows)
```

Si un index SP-Gist existe, l'index peut être utilisé, comme le montre cet exemple.

```
DROP INDEX startswith_t_idx;
CREATE INDEX ON startswith USING spgist (t);
EXPLAIN (COSTS OFF)
  SELECT *
    FROM startswith
   WHERE starts_with(t, 'commence par 1');
```

```
                      QUERY PLAN
-------------------------------------------------------
 Index Only Scan using startswith_t_idx on startswith
   Index Cond: (t ^@ 'commence par 1'::text)
   Filter: starts_with(t, 'commence par 1'::text)
(3 rows)
```

Dans ce cas, les prédicats du type `chaine LIKE 'foo%'` sont transformés avec l'opérateur `^@` pour tirer parti de l'index.

```
EXPLAIN (COSTS OFF)
  SELECT *
    FROM startswith
   WHERE t LIKE 'commence par 1%';
```

```
                      QUERY PLAN
-------------------------------------------------------
 Index Only Scan using startswith_t_idx on startswith
   Index Cond: (t ^@ 'commence par 1'::text)
   Filter: (t ~~ 'commence par 1%'::text)
(3 rows)
```

4.3 AJOUT DE FONCTIONS D'EXPRESSION RÉGULIÈRES POUR LA COMPATIBILITÉ AVEC D'AUTRES SGBD

- nouvelles fonctions :
  - `regexp_count()`
  - `regexp_instr()`
  - `regexp_like()`
  - `regexp_substr()`
- fonction améliorée :
  - `regexp_replace()`

Les fonctions[8] `regexp_count()`, `regexp_instr()`, `regexp_like()` et `regexp_substr()` ont été ajouté à PostgreSQL afin d'augmenter la compatibilité avec les autres SGBD et de faciliter la réalisation de certaines tâches. La fonction `regexp_replace()` a également été étendue.

La **fonction** `regexp_count()` permet de compter le nombre de fois qu'une expression régulière trouve une correspondance dans la chaine placée en entrée.

```
regexp_count(
    string text
    , pattern text
    [, start integer[, flags text]]
) → integer`
```

Note: la liste des flags[9] se trouve dans la documentation, certaines fonctions ont des flags supplémentaires comme `g` qui permet de sélectionner toutes les occurrences.

Il y a une occurrence d'une chaîne contenant une lettre en minuscule suivie de 3 nombres dans la chaine d'entrée si l'on commence à la position 5.

```
SELECT regexp_count('a125 a5 a661 B12 lmlm', '[a-z][\d]{3}', 5);

 regexp_count
--------------
            1
(1 row)
```

La **fonction** `regexp_instr()` permet de renvoyer la position de la première occurrence qui correspond à l'expression régulière dans la chaine placée en entrée ou zéro si elle n'est pas trouvée.

[8] https://www.postgresql.org/docs/15/functions-string.html#FUNCTIONS-STRING-OTHER
[9] https://www.postgresql.org/docs/15/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE

```
regexp_instr(
    string text
    , pattern text
    [, start integer[, N integer[, endoption integer[, flags text[, subexpr integer]]]]]
) → integer
```

Si on reprend l'exemple précédent, la position de la première occurrence de l'expression régulière dans la chaîne placée en entrée en commençant la recherche à la position 5 est 9.

```
SELECT regexp_instr('a125 a5 a661 B12 lmlm', '[a-z][\d]{3}', 5);
```

```
 regexp_instr
--------------
            9
(1 row)
```

La **fonction** `regexp_like()` permet de renvoyer `true` s'il y a une occurrence qui correspond à l'expression régulière dans la chaine placée en entrée, `false` sinon.

```
regexp_like(
    string text
    , pattern text[, flags text]
) → boolean
```

Voici un exemple qui illustre le fonctionnement de cette fonction.

```
SELECT regexp_like('a125 a5 a661 B12 lmlm', '[a-z][\d]{3}') AS "regex 1",
       regexp_like('a125 a5 a661 B12 lmlm', '[a-z]{2}[\d]{3}') AS "regex 2";
```

```
regex 1 | regex 2
---------+---------
 t       | f
(1 row)
```

La **fonction** `regexp_substr()` permet de renvoyer la chaine qui correspond à la Nème occurrence de l'expression régulière ou NULL s'il n'y a pas d'occurrence.

```
regexp_substr(
    string text
    , pattern text
    [, start integer[, N integer[, flags text[, subexpr integer]]]]
) → text
```

```
SELECT regexp_substr('a125 a5 a661 B12 lmlm', '[a-z][\d]{3}') AS "regex 1",
       regexp_substr('a125 a5 a661 B12 lmlm', '[a-z][\d]{3}', 1, 2) AS "regex 2",
       regexp_substr('a125 a5 a661 B12 lmlm', '([a-z])([\d]{3})', 1, 2, 'i', 2) AS "regex 3" \gx
```

```
-[ RECORD 1 ]-
regex 1 | a125  -- première occurrence
```

```
regex 2 | a661  -- seconde occurrence
regex 3 | 661   -- seconde occurrence seconde sous expression (parenthèses)
```

La **fonction** `regexp_replace()` permet de remplacer la Nème occurrence d'une chaîne de caractère qui correspond à l'expression régulière par une autre chaine de caractère dans la chaine fournie en entrée. Son fonctionnement a été étendu en version 15 pour permettre de spécifier une position de départ et un nombre d'expressions à remplacer.

```
regexp_replace(
    string text
    , pattern text
    , replacement text
    [, start integer[, flags text]]

regexp_replace(
    string text
    , pattern text
    , replacement text
    , start integer
    , N integer
    [, flags text]
) → text
```

Cet exemple remplace la seconde occurrence de l'expression régulière par 'PostgreSQL'.

```
SELECT regexp_replace(e'Un Oracle a prédit le succès de Oracle', 'Oracle', 'PostgreSQL', 1, 2);

              regexp_replace
-------------------------------------------
 Un Oracle a prédit le succès de PostgreSQL
(1 row)
```

Avec le flag `g`, il est facile de remplacer toutes les occurrences repérées par 'PostgreSQL'.

```
SELECT regexp_replace('oracle SQLServer Mysql', '(Oracle|\w*SQL\w*)', 'PostgreSQL', 'g

          regexp_replace
-----------------------------------
 PostgreSQL PostgreSQL PostgreSQL
(1 row)
```

5 RÉGRESSIONS

5.1 RETRAIT DU SUPPORT DES INSTANCES DE VERSIONS 9.1 ET ANTÉRIEURES

- Changement sur la compatibilité des outils en version 15 :
  - `psql`, `pg_dump` et `pg_dumpall` ne supportent plus l'accès à des serveurs 9.1 ou antérieur
  - `pg_upgrade` ne supporte plus la mise à niveau depuis une instance 9.1 ou antérieur.

Cette version contient des modifications de catalogue qui impactent la compatibilité avec d'anciennes version. Le client `psql` ne supporte ainsi plus d'accéder à des serveurs de versions 9.1 ou antérieures.

Les outils `pg_dump` et `pg_dumpall` ne supportent plus d'effectuer des exports de données depuis une instance de version 9.1 ou antérieure. La restauration d'anciennes archives n'est par ailleurs pas garantie.

De plus, `pg_ugrade` ne supporte plus la mise à niveau depuis une instance de version 9.1 ou antérieure.

Ces régressions peuvent être particulièrement impactantes pour les migrations. Une version intermédiaire devra dans certains cas être utilisée pour la mise à niveau en 15 d'une très ancienne version.

5.2 PYTHON2 DÉPRÉCIÉ : RETRAIT DES LANGAGES PLPYTHON2U ET PLPYTHONU

- Fin du support de Python 2.x
  - Retrait des langages procéduraux `plpython2u` et `plpythonu`

Cette version marque la fin du support de Python 2 comme langage procédurale pour les fonctions PostgreSQL.

Le langage procédural `plpython2u`, qui implémente *PL/Python* avec Python 2, est ainsi retiré. Seul le langage `plpython3u`, qui implémente *PL/Python* avec Python 3, est désormais utilisable.

Le langage procédural `plpythonu`, qui pouvait pointer sur la version 2 ou 3 en fonction du paramètre par défaut configuré dans chaque version de PostgreSQL, a également été retiré puisqu'il n'a plus d'utilité.

5.3 QUESTIONS ?

Merci de votre écoute !
Nouveautés de la version 15 :
https://dali.bo/workshop15_html
https://dali.bo/workshop15_pdf

NOTES

NOTES

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**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.