PGSession 13

Nouveautés de PostgreSQL 13

DALIBO
L'expertise PostgreSQL

13

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 13

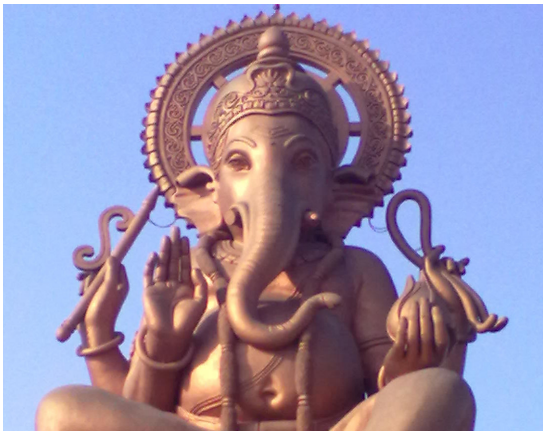
PGSession 13

TITRE : Nouveautés de PostgreSQL 13
SOUS-TITRE : PGSession 13

REVISION: 13

Table des Matières

1 NOUVEAUTÉS DE POSTGRESQL 13



Photographie de Jainswatantra[1] , licence GNU FREE Documentation Licence[2] , obtenue sur wikimedia.org[3] .

Participez à ce workshop !

Pour des précisions, compléments, liens, exemples, et autres corrections et suggestions, soumettez vos *Pull Requests* dans notre dépôt :

https://github.com/dalibo/workshops/tree/master/fr

Licence : PostgreSQL[4]

---

[1] https://commons.wikimedia.org/w/index.php?title=User:Jainswatantra&action=edit&redlink=1
[2] https://en.wikipedia.org/wiki/fr:Licence_de_documentation_libre_GNU
[3] https://commons.wikimedia.org/wiki/File:Ganesha_Bhubaneswar_Odisha.jpg
[4] https://github.com/dalibo/workshops/blob/master/LICENSE.md

1.1 LA V13

- Développement depuis le 1er juillet 2019
- Sortie le 24 septembre 2020
- version 13.1 sortie le 12 novembre 2020

1.2 LES NOUVEAUTÉS

- Administration
- Réplication physique et logique
- Supervision
- Performances
- Régressions / changements
- Ateliers

1.3 ADMINISTRATION

- Maintenance :
  - parallélisation des `vacuum` et `reindex`
  - Autovacuum : déclenchement par *INSERT*
- Création d'un fichiers manifeste par `pg_basebackup`
- Déconnexion des utilisateurs à la suppression d'une base de données

1.3.1 VACUUM : NOUVEAUX WORKERS

- *VACUUM* peut paralléliser le traitement des index
- Nouvelle option *PARALLEL*
  - si 0, non parallélisé
- La table doit avoir :
  - au minimum deux index
  - des index d'une taille supérieure à `min_parallel_index_scan_size`
- Non disponible pour le *VACUUM FULL*

Le `VACUUM` fonctionne en trois phases :

- parcours de la table pour trouver les lignes à nettoyer
- nettoyage des index de la table
- nettoyage de la table

La version 13 permet de traiter les index sur plusieurs CPU, un par index. De ce fait, ceci n'a un intérêt que si la table contient au moins deux index, et que ces index ont une taille supérieure à `min_parallel_index_scan_size` (512 ko par défaut).

Lors de l'exécution d'un `VACUUM` parallélisé, un ou plusieurs autres processus sont créés. Ces processus sont appelés des *workers*, alors que le processus principal s'appelle le *leader*. Il participe lui aussi au traitement des index. De ce fait, si une table a deux index et que la parallélisation est activée, PostgreSQL utilisera le leader pour un index et un worker pour l'autre index.

Par défaut, PostgreSQL choisit de lui-même s'il doit utiliser des workers et leur nombre. Il détermine cela automatiquement, suivant le nombre d'index éligibles, en se limitant à un maximum correspondant à la valeur du paramètre `max_parallel_maintenance_workers` (2 par défaut).

Pour forcer un certain niveau de parallélisation, il faut utiliser l'option `PARALLEL`. Cette dernière doit être suivie du niveau de parallélisation. Il est garanti qu'il n'y aura pas plus que ce nombre de processus pour traiter la table et ses index. En revanche, il peut y en avoir moins. Cela dépend une nouvelle fois du nombre d'index éligibles, de la configuration du paramètre `max_parallel_maintenance_workers`, mais aussi du nombre de workers autorisé, limité par le paramètre `max_parallel_workers` (8 par défaut).

En utilisant l'option `VERBOSE`, il est possible de voir l'impact de la parallélisation et le travail des différents *workers* :

```
CREATE TABLE t1 (c1 int, c2 int) WITH (autovacuum_enabled = off) ;
INSERT INTO t1 SELECT i,i FROM generate_series (1,1000000) i;
CREATE INDEX t1_c1_idx ON t1 (c1) ;
```

```
CREATE INDEX t1_c2_idx ON t1 (c2) ;
DELETE FROM t1 ;

VACUUM (VERBOSE, PARALLEL 3) t1 ;
INFO:  vacuuming "public.t1"
INFO:  launched 1 parallel vacuum worker for index vacuuming (planned: 1)
INFO:  scanned index "t1_c2_idx" to remove 10000000 row versions
                                                by parallel vacuum worker
DETAIL:  CPU: user: 4.14 s, system: 0.29 s, elapsed: 6.85 s
INFO:  scanned index "t1_c1_idx" to remove 10000000 row versions
DETAIL:  CPU: user: 6.31 s, system: 0.59 s, elapsed: 19.62 s
INFO:  "t1": removed 10000000 row versions in 63598 pages
DETAIL:  CPU: user: 1.16 s, system: 0.90 s, elapsed: 7.24 s
...
```

Enfin, il est à noter que cette option n'est pas disponible pour le `VACUUM FULL` :

```
# VACUUM (FULL,PARALLEL 2);
ERROR:  VACUUM FULL cannot be performed in parallel
```

1.3.2 VACUUMDB --PARALLEL

- Nouvelle option `--parallel` (`-P`)
- Utilisé pour la nouvelle clause PARALLEL de VACUUM
- À ne pas confondre avec l'option `--jobs`

L'outil `vacuumdb` dispose de l'option `-P` (ou `--parallel` en version longue). La valeur de cette option est utilisée pour la clause `PARALLEL` de la commande `VACUUM`. Il s'agit donc de paralléliser le traitement des index pour les tables disposant d'au moins deux index.

En voici un exemple :

```
$ vacuumdb --parallel 2 --table t1 -e postgres 2>&1 | grep VACUUM
VACUUM (PARALLEL 2) public.t1;
```

Ce nouvel argument n'est pas à confondre avec `--jobs` qui existait déjà. L'argument `--jobs` lance plusieurs connexions au serveur PostgreSQL pour exécuter plusieurs `VACUUM` sur plusieurs objets différents en même temps.

1.3.3 REINDEXDB --JOBS

- Nouvelle option `--jobs` (`-j`) pour reindexdb
- Lance autant de connexions sur le serveur PostgreSQL
- Exécute un REINDEX par connexion
- Incompatible avec les options SYSTEM et INDEX

L'outil `reindexdb` dispose enfin de l'option `-j` (`--jobs` en version longue).

L'outil lance un certain nombre de connexions au serveur de bases de données, ce nombre dépendant de la valeur de l'option en ligne de commande. Chaque connexion se voit dédier un index à réindexer. De ce fait, l'option `--index`, qui permet de réindexer un seul index, n'est pas compatible avec l'option `-j`.

Rappelons que la (ré)indexation est parallélisée depuis PostgreSQL 11 (paramètre `max_parallel_maintenance_workers`), et qu'un `REINDEX` peut donc déjà utiliser plusieurs processeurs.

De même, l'option `--system` permet de réindexer les index systèmes. Or ceux-ci sont toujours réindexés sur une seule connexion pour éviter un *deadlock*. L'option `--system` est donc incompatible avec l'option `-j`. Elle n'a pas de sens si on demande à ne réindexer qu'un index (`--index`).

1.3.4 AUTOVACUUM : DÉCLENCHEMENT PAR INSERT

- Avant PostgreSQL v13 :
  - des INSERTs déclenchent un ANALYZE automatique
  - mais pas de VACUUM
- VACUUM important pour les VM et FSM
- Deux nouveaux paramètres :
  - autovacuum_vacuum_insert_threshold
  - autovacuum_vacuum_insert_scale_factor

Avant la version 13, les `INSERT` n'étaient considérés par l'autovacuum que pour les opérations `ANALYZE`. Cependant, le `VACUUM` a aussi une importance pour la mise à jour des fichiers de méta-données que sont la FSM (*Free Space Map*) et la VM (*Visibility Map*). Notamment, pour cette dernière, cela permet à PostgreSQL de savoir si un bloc ne contient que des lignes vivantes, ce qui permet à l'exécuteur de passer par un `Index Only Scan` au lieu d'un `Index Scan` probablement plus lent.

Ainsi, exécuter un `VACUUM` régulièrement en fonction du nombre d'insertions réalisé est important. L'ancien comportement pouvait poser problème pour les tables uniquement en insertion.

Les développeurs de PostgreSQL ont donc ajouté cette fonctionnalité en intégrant deux nouveaux paramètres, dont le franchissement va déclencher un `VACUUM` :

- `autovacuum_vacuum_insert_threshold` indique le nombre minimum de lignes devant être insérées, par défaut à 1000 ;
- `autovacuum_vacuum_insert_scale_factor` indique le ratio minimum de lignes, par défaut à 0.2.

Il est à noter que nous retrouvons l'ancien comportement (pré-v13) en configurant ces deux paramètres à la valeur -1.

Le `VACUUM` exécuté fonctionne exactement de la même façon que tout autre `VACUUM`. Il va notamment nettoyer les index même si, strictement parlant, ce n'est pas indispensable dans ce cas.

1.3.5 FICHIERS MANIFESTE POUR LES SAUVEGARDES

1.3.6 FICHIERS MANIFESTES

- `pg_basebackup` crée une liste des fichiers présents dans les sauvegardes : le fichier manifeste.
- Trois nouvelles options :
  - `--no-manifest`
  - `--manifest-force-encode`
  - `--manifest-checksums=[NONE|CRC32C|SHA224|SHA256|SHA384|SHA512]`

`pg_basebackup` crée désormais par défaut un fichier manifeste. C'est un fichier `json`. Il contient pour chaque fichier inclus dans la sauvegarde :

- le chemin relatif à `$PGDATA` ;
- la taille du fichier ;
- la date de dernière modification ;
- l'algorithme de calcul de somme de contrôle utilisé ;
- la somme de contrôle.

Il contient également, un numéro de version de manifeste, sa propre somme de contrôle et la plage de journaux de transactions nécessaire à la restauration.

```
$ cat backup_manifest | jq
{
  "PostgreSQL-Backup-Manifest-Version": 1,
  "Files": [
    {
      "Path": "backup_label",
      "Size": 225,
      "Last-Modified": "2020-05-12 15:58:59 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "c7b34439"
    },
    {
      "Path": "tablespace_map",
      "Size": 0,
      "Last-Modified": "2020-05-12 15:58:59 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "00000000"
    },
    ...
  ],
  "WAL-Ranges": [
    {
```

```
      "Timeline": 1,
      "Start-LSN": "0/4000028",
      "End-LSN": "0/4000100"
    }
  ],
  "Manifest-Checksum": "1107abd51[...]732d7b24f217b5e4"
}
```

Le fichier manifeste nommé `backup_manifest` est créé quel que soit le format de sauvegarde choisi (`plain` ou `tar`).

`pg_basebackup` dispose de trois options relatives aux fichiers manifestes :

- `--no-manifest` : ne pas créer de fichier manifeste pour la sauvegarde.
- `--manifest-force-encode` : forcer l'encodage de l'intégralité des noms de fichiers de la sauvegarde en hexadécimal. Sans cette option, seuls les fichiers dont le nom n'est pas encodé en UTF-8 sont encodés en hexadécimal. Cette option est destinée aux tests des outils tiers qui manipulent des fichiers manifestes.
- `--manifest-checksums=[NONE|CRC32C|SHA224|SHA256|SHA384|SHA512]` : permet de spécifier l'algorithme de somme de contrôle appliqué à chaque fichier inclus dans la sauvegarde. L'algorithme par défaut est `CRC32C`. La valeur `NONE` a pour effet de ne pas inclure de somme de contrôle dans le fichier manifeste.

L'impact du calcul des sommes de contrôle sur la durée de la sauvegarde est variable en fonction de l'algorithme choisi.

Voici les mesures faites sur un ordinateur portable Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz avec 8 CPU et équipé d'un disque dur SSD. L'instance fait 6 Go et a été générée en utilisant `pgbench` avec un facteur d'échelle de `400`.

algorithme	nombre de passes	temps moyen (s)
pas de manifeste	50	33.6308
NONE	50	32.7352
CRC32C	50	33.6722
SHA224	50	56.3702
SHA256	50	55.664
SHA384	50	45.754
SHA512	50	46.1696

Le calcul des sommes de contrôle avec l'algorithme `CRC32C` a donc un impact peu important sur la durée de la sauvegarde. L'impact est beaucoup plus important avec les algo-

rithmes de type SHA. Les sommes de contrôle SHA avec un grand nombre de bits sont plus performantes. Ces observations sont en accord avec celles faites pendant le développement de cette fonctionnalité.

Un intérêt des sommes de contrôle SHA avec un nombre de bits élevé est de diminuer les chances de produire un faux positif. Mais surtout, dans les milieux les plus sensibles, il permet de parer à toute modification mal intentionnée d'un backup, théoriquement possible avec des algorithmes trop simples. Le manifeste doit alors être copié séparément.

1.3.7 NOUVEL OUTIL PG_VERIFYBACKUP

- Fonction : vérifier une sauvegarde au format `plain` grâce au fichier manifeste.
- 4 étapes :
  - vérification de la présence du manifeste et de sa somme de contrôle
  - vérification de la présence des fichiers écrits dans le manifeste
  - vérification des sommes de contrôle des fichiers présents dans le manifeste
  - vérification des WALs (présence, somme de contrôle des enregistrements)
- Ne dispense pas de tester les sauvegardes en les restaurant !

`pg_verifybackup` permet de vérifier que le contenu d'une sauvegarde au format `plain` correspond bien à ce que le serveur a envoyé lors de la sauvegarde.

La vérification se déroule en quatre étapes. Il est possible de demander à l'outil de s'arrêter à la première erreur avec `-e`, `--exit-on-error`. On peut également faire en sorte d'ignorer certains répertoires avec `-i`, `--ignore=RELATIVE_PATH`.

- La première étape consiste à vérifier la présence du fichier manifeste et l'exactitude de sa somme de contrôle. Par défaut, l'outil cherche le fichier de manifeste dans le répertoire de sauvegarde donné en paramètre. Il est également possible d'utiliser l'option `-m` ou `--manifest-path=PATH` pour spécifier le chemin vers le fichier de manifeste.

- La seconde étape consiste à vérifier que les fichiers présents dans le manifeste sont bien présents dans la sauvegarde. Les fichiers manquants ou supplémentaires sont signalés à l'exception de : `postgresql.auto.conf`, `standby.signal`, `recovery.signal`, le fichier manifeste lui-même ainsi que le contenu du répertoire `pg_wal`.

- La troisième étape consiste à vérifier les sommes de contrôle des fichiers présents dans le manifeste.

- La dernière étape permet de vérifier la présence et l'exactitude des journaux de transactions nécessaires à la restauration. Cette étape peut être ignorée en spécifiant le paramètre `-n, --no-parse-wal`. Le répertoire contenant les journaux de transactions peut être spécifié avec le paramètre `-w, --wal-directory=PATH`. Par défaut, l'outil cherche un répertoire `pg_wal` présent dans le répertoire de sauvegarde. Les journaux sont analysés avec `pg_waldump` pour vérifier les sommes de contrôle des enregistrements qu'ils contiennent. Seule la plage de journaux de transactions présente dans le fichier manifeste est vérifiée.

`pg_verifybackup` permet donc de vérifier que ce que contient la sauvegarde est conforme avec ce que le serveur a envoyé. Cependant, cela ne garantit pas que la sauvegarde est exempte d'autres problèmes. Il est donc toujours nécessaire de tester les sauvegardes réalisées en les restaurant.

Actuellement, seules des sauvegardes produites par `pg_basebackup` ou des outils qui l'utilisent comme `Barman` en mode *streaming-only* peuvent être vérifiées. Les autres outils de sauvegarde tels que `pgBackRest`, `pitrery` ou `Barman` (en mode rsync) ne permettent pas encore de générer des fichiers manifestes compatibles avec PostgreSQL. Cela pourrait changer dans un avenir proche.

1.3.8 DÉCONNEXION DES UTILISATEURS À LA SUPPRESSION D'UNE BASE DE DONNÉES

- Nouvelle clause `WITH FORCE` pour `DROP DATABASE`
- Force la déconnexion des utilisateurs
- Nouvel argument `--force` pour l'outil `dropdb`

Dans les versions précédentes de PostgreSQL, une erreur est levée si une connexion existe sur la base que l'on souhaite supprimer :

```
ERROR:  database "dropme" is being accessed by other users
DETAIL:  There is 1 other session using the database.
```

C'est toujours le cas par défaut ! Cependant, il est désormais possible de demander à la commande de tenter de déconnecter les personnes actives sur la base désignée afin d'en terminer la suppression :

- pour la commande SQL avec l'option `FORCE`:

  ```
  DROP DATABASE dropme WITH (FORCE);
  ```

- pour la commande `dropdb` avec l'argument `--force`:

  ```
  dropdb --force dropme
  ```

Exemple:

```
$ createdb dropme

$ psql -qc "select pg_sleep(3600)" dropme &
[1] 16426

$ dropdb dropme
dropdb: error: database removal failed: ERROR:  database "dropme" is being accessed by
DETAIL:  There is 1 other session using the database.

$ dropdb --force --echo dropme
SELECT pg_catalog.set_config('search_path', '', false);
DROP DATABASE dropme WITH (FORCE);

FATAL:  terminating connection due to administrator command
[1]+  Exit 2                  psql -qc "select pg_sleep(3600)" dropme
```

1.4 RÉPLICATION PHYSIQUE

- Modification à chaud des paramètres de réplication
- Volume maximal de journaux conservé par les slots
- Évolution dans la commande `pg_rewind` :
  - Restauration de WAL archivés via le paramètre `--restore-target-wal`
  - Génération de la configuration de la réplication
  - Récupération automatique d'une instance

1.4.1 PG_REWIND SAIT RESTAURER DES JOURNAUX

- `-c`/`--restore-target-wal` permet de restaurer les archives de journaux de transactions de l'instance cible.

`pg_rewind`[5] permet de synchroniser le répertoire de données d'une instance avec un autre répertoire de données de la même instance. Il est notamment utilisé pour réactiver une ancienne instance primaire en tant qu'instance secondaire répliquée depuis la nouvelle instance primaire suite à une bascule.

Dans la terminologie de l'outil, on parle de source pour la nouvelle primaire et cible pour l'ancienne.

`pg_rewind` identifie le point de divergence entre la cible et la source. Il doit ensuite identifier tous les blocs modifiés sur la cible après le point de divergence afin de pouvoir les corriger avec les données de la source. Pour réaliser cette tâche, l'outil doit parcourir les journaux de transactions générés par la cible.

Avant PostgreSQL 13, ces journaux de transactions devaient être présents dans le répertoire `PGDATA/pg_wal` de la cible. Dans les cas où la cible n'a pas été arrêtée rapidement après la divergence, cela peut poser problème, car les WAL nécessaires ont potentiellement déjà été recyclés.

Il est désormais possible d'utiliser l'option `-c` ou `--restore-target-wal` afin que l'outil utilise la commande de restauration `restore_commande` de l'instance cible pour récupérer ces journaux de transactions à l'endroit où ils ont été archivés.

Note : certains fichiers ne sont pas protégés par les WAL et sont donc copiés entièrement. Voici quelques exemples :

- les fichiers de configuration : `postgresql.conf`, `pg_ident.conf`, `pg_hba.conf` ;
- la *visibility map* (fichiers `*_vm`), et la *free space map* (fichiers `*_fsm`) ;
- le répertoire `pg_clog`.

[5] https://www.postgresql.org/docs/13/app-pgrewind.html

1.4.2 PG_REWIND RÉCUPÈRE AUTOMATIQUEMENT UNE INSTANCE

- pg_rewind lance automatiquement la phase de récupération du serveur cible si nécessaire avant son traitement.
- il est possible de désactiver ce nouveau comportement avec `--no-ensure-shutdown`.

pg_rewind s'assure que le serveur cible a été arrêté proprement avant de lancer tout traitement. Si ce n'est pas le cas, l'instance est démarrée en mode mono-utilisateur afin d'effectuer la récupération (phase de *crash recovery*). Elle est ensuite éteinte.

L'option `--no-ensure-shutdown` permet de ne pas faire ces opérations automatiquement. Si l'instance cible n'a pas été arrêtée proprement, un message d'erreur est affiché et l'utilisateur doit faire les actions nécessaires lui-même. C'était le fonctionnement normal dans les versions précédentes de l'outil.

1.4.3 PG_REWIND GÉNÈRE LA CONFIGURATION DE RÉPLICATION

- `--write-recovery-conf` permet de générer le fichier `standby.signal` et configure la connexion à l'instance primaire dans `postgresql.auto.conf` ;
- nécessite de préciser l'argument `--source-server`

Le nouveau paramètre `-R` ou `--write-recovery-conf` permet de spécifier à pg_rewind qu'il doit :

- générer le fichier `PGDATA/standby.signal` ;
- ajouter le paramètre `primary_conninfo` au fichier `PGDATA/postgresql.auto.conf`.

Ce paramètre nécessite l'utilisation de `--source-server` pour fonctionner. La chaîne de connexion ainsi spécifiée sera celle utilisée par pg_rewind pour générer le paramètre `primary_conninfo` dans `PGDATA/postgresql.auto.conf`. Cela signifie que l'utilisateur sera le même que celui utilisé pour l'opération de resynchronisation.

1.5 RÉPLICATION LOGIQUE

Il est désormais possible de :

- ajouter une table partitionnée à une publication
- répliquer vers une table partitionnée
- répliquer depuis la racine d'une table partitionnée (option `publish_via_partition_root`)
  - en cas de partitionnement différent sur la cible

Dans les versions précédentes, il était possible d'ajouter des partitions à une publication afin de répliquer les opérations sur celles-ci. Il est désormais possible d'ajouter directement une table partitionnée à une publication, toutes les partitions seront alors automatiquement ajoutées à celle-ci. Tout ajout ou suppression de partition sera également reflété dans la liste des tables présentes dans la publication sans action supplémentaire. Il faudra cependant rafraîchir la souscription pour qu'elle prenne en compte les changements opérés avec la commande de modification de souscription[6] :

```
ALTER SUBSCRIPTION <sub> REFRESH PUBLICATION;
```

La version 13 de PostgreSQL permet de répliquer vers une table partitionnée.

Il est également possible de répliquer depuis la racine d'une table partitionnée. Cette fonctionnalité est rendue possible par l'ajout d'un paramètre de publication : `publish_via_partition_root`. Il détermine si les modifications faites sur une table partitionnée contenue dans une publication sont publiées en utilisant le schéma et le nom de la table partitionnée plutôt que ceux de ses partitions. Cela permet de répliquer depuis une table partitionnée vers une table classique ou partitionnée avec un schéma de partitionnement différent.

L'activation de ce paramètre est effectuée via la commande CREATE PUBLICATION[7] ou ALTER PUBLICATION[8] .

Exemple :

```
CREATE PUBLICATION pub_table_partitionnee
    FOR TABLE factures
    WITH ( publish_via_partition_root = true );
```

Si ce paramètre est utilisé, les ordres TRUNCATE exécutés sur les partitions ne sont pas répliqués.

[6] https://www.postgresql.org/docs/13/sql-altersubscription.html
[7] https://www.postgresql.org/docs/13/sql-createpublication.html
[8] https://www.postgresql.org/docs/13/sql-alterpublication.html

1.6 SUPERVISION

- Journaux applicatifs :
  - Tracer un échantillon des transactions suivant leur durée
  - Tracer le type de processus
- Suivi de l'avancée des `ANALYZE`
- Suivi de l'avancée des sauvegardes par `pg_basebackup`
- Statistiques d'utilisation des WAL

1.6.1 TRACER UN ÉCHANTILLON DES REQUÊTES SUIVANT LEUR DURÉE

- `log_min_duration_sample` : durée minimum requise pour qu'une requête échantillonnée puisse être tracée.
- `log_statement_sample_rate` : probabilité qu'une requête durant plus de `log_min_duration_sample` soit tracée.
- priorité de `log_min_duration_statement` sur `log_min_duration_sample`.

Activer la trace des requêtes avec un seuil trop bas via le paramètre `log_min_duration_statement` peut avoir un impact important sur les performances ou sur le remplissage des disques à cause de la quantité d'écritures réalisées.

Un nouveau mécanisme a dont été introduit pour faire de l'échantillonnage. Ce mécanisme s'appuie sur deux paramètres pour configurer un seuil de déclenchement de l'échantillonnage dans les traces : `log_min_duration_sample`, et un taux de requête échantillonné : `log_statement_sample_rate`.

Par défaut, l'échantillonnage est désactivé (`log_min_duration_sample = -1`). Le taux d'échantillonnage, dont la valeur est comprise entre `0.0` et `1.0`, a pour valeur par défaut `1.0`. La modification de ces paramètres peut être faite jusqu'au niveau de la transaction, il faut cependant se connecter avec un utilisateur bénéficiant de l'attribut `SUPERUSER` pour cela.

Si le paramètre `log_min_duration_statement` est configuré, il a la priorité. Dans ce cas, seules les requêtes dont la durée est supérieure à `log_min_duration_sample` et inférieure à `log_min_duration_statement` sont échantillonnées. Toutes les requêtes dont la durée est supérieure à `log_min_duration_statement` sont tracées normalement.

1.6.2 TRACER LE TYPE DE PROCESSUS

- Ajout d'un nouvel échappement (`%b`) à `log_line_prefix` pour tracer le type de backend.
- Le type de backend est également ajouté aux traces formatées en csv.

Le paramètre `log_line_prefix` dispose d'un nouveau caractère d'échappement : `%b`. Ce caractère permet de tracer le type de backend à l'origine d'un message. Il reprend le contenu de la colonne `pg_stat_activity.backend_type` cependant d'autres type de backend peuvent apparaître dont `postmaster`.

Exemple pour la configuration suivante de `log_line_prefix = '[%b:%p] '`.

```
[postmaster:6783] LOG:  starting PostgreSQL 13.0 on x86_64-pc-linux-gnu,
+++compiled by gcc (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2), 64-bit
[postmaster:6783] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5433"
[startup:6789] LOG:  database system was interrupted; last known up at
+++2020-11-02 12:02:32 CET
[startup:6789] LOG:  database system was not properly shut down;
+++automatic recovery in progress
[startup:6789] LOG:  redo starts at 1/593E1038
[startup:6789] LOG:  invalid record length at 1/593E1120: wanted 24, got 0
[startup:6789] LOG:  redo done at 1/593E10E8
[postmaster:6783] LOG:  database system is ready to accept connections
[checkpointer:6790] LOG:  checkpoints are occurring too frequently (9 seconds apart)
[autovacuum worker:7969] LOG:  automatic vacuum of table
+++"postgres.public.grosfic": index scans: 0
   pages: 0 removed, 267557 remain, 0 skipped due to pins, 0 skipped frozen
   tuples: 21010000 removed, 21010000 remain, 0 are dead but not yet removable,
+++oldest xmin: 6196
   buffer usage: 283734 hits, 251502 misses, 267604 dirtied
   avg read rate: 10.419 MB/s, avg write rate: 11.086 MB/s
   system usage: CPU: user: 16.78 s, system: 8.53 s, elapsed: 188.58 s
   WAL usage: 802621 records, 267606 full page images, 2318521769 bytes
[autovacuum worker:7969] LOG:  automatic analyze of table "postgres.public.grosfic"
+++system usage: CPU: user: 0.54 s, system: 0.58 s, elapsed: 5.93 s
```

Le type de backend a également été ajouté aux traces formatées en csv (`log_destination = 'csvlog'`).

1.6.3 SUIVI DE L'EXÉCUTION DES `ANALYZE`

- Nouvelle vue `pg_stat_progress_analyze`

La vue `pg_stat_progress_analyze` vient compléter la liste des vues qui permettent de suivre l'activité des tâches de maintenance.

Cette vue contient une ligne pour chaque backend qui exécute la commande `ANALYZE`. Elle contient les informations :

- `pid` : *id* du processus qui exécute l'analyze ;
- `datid` : *oid* de la base de donnée à laquelle est connecté le backend ;
- `datname` : nom de la base de donnée à laquelle est connecté le backend ;
- `relid` : *oid* de la table analysée ;
- `phase` : phase du traitement parmi les valeurs :
  - `initializing` : préparation du scan de la table ;
  - `acquiring sample rows` : scan de la table pour collecter un échantillon de lignes ;
  - `acquiring inherited sample rows` : scan des tables filles pour collecter un échantillon de lignes ;
  - `computing statistics` : calcul des statistiques ;
  - `computing extended statistics` : calcul des statistiques étendues ;
  - `finalizing analyze` : mise à jour des statistiques dans `pg_class`.
- `sample_blks_total` : nombre total de blocs qui vont être échantillonnés ;
- `sample_blks_scanned` : nombre de blocs scannés ;
- `ext_stats_total` : nombre de statistiques étendues ;
- `ext_stats_computed` : nombre de statistiques étendues calculées ;
- `child_tables_total` : nombre de tables filles à traiter pendant la phase `acquiring inherited sample rows` ;
- `child_tables_done` : nombre de tables filles traitées pendant la phase `acquiring inherited sample rows` ;
- `current_child_table_relid` : *oid* de la table fille qui est en train d'être scannée pendant la phase `acquiring inherited sample rows`.

1.6.4 SUIVI DE L'EXÉCUTION DES SAUVEGARDES

- Nouvelle vue : `pg_stat_progress_basebackup`
- Permet de surveiller :
  - la phase de la sauvegarde ;
  - la volumétrie sauvegardée et restant à sauvegarder ;
  - le nombre de tablespaces sauvegardés et restant à sauvegarder.

La vue `pg_stat_progress_basebackup`[9] est composée des champs suivants :

- `pid` : le pid du processus *wal sender* associé à la sauvegarde ;
- `phase` : la phase de la sauvegarde ;
- `backup_total` : l'estimation de la volumétrie totale à sauvegarder ;
- `backup_streamed` : la volumétrie déjà sauvegardée ;
- `tablespaces_total` : le nombre de tablespaces à traiter ;
- `tablespaces_streamed` : le nombre de tablespaces traités.

La sauvegarde se déroule en plusieurs étapes qui peuvent être suivies grâce au champ `phase` de la vue :

- `initializing` : cette phase est très courte et correspond au moment où le *wal sender* se prépare à démarrer la sauvegarde.
- `waiting for checkpoint to finish` : cette phase correspond au moment où le processus *wal sender* réalise un `pg_start_backup` et attend que PostgreSQL fasse un `CHECKPOINT`.
- `estimating backup size` : c'est la phase où le *wal sender* estime la volumétrie à sauvegarder. Cette étape peut être longue et coûteuse en ressource si la base de données est très grosse. Elle peut être évitée en spécifiant l'option `--no-estimate-size` lors de la sauvegarde. Dans ce cas, la colonne `backup_total` est laissée vide.
- `streaming database files` : ce statut signifie que le processus *wal sender* est en train d'envoyer les fichiers de la base de données.
- `waiting for wal archiving to finish` : le *wal sender* est en train de réaliser le `pg_stop_backup` de fin de sauvegarde et attend que l'ensemble des journaux de transactions nécessaires à la restauration soient archivés. C'est la dernière étape de la sauvegarde quand les options `--wal-method=none` ou `--wal-method=stream` sont utilisées.

[9] https://www.postgresql.org/docs/13/progress-reporting.html#BASEBACKUP-PROGRESS-REPORTING

- `transferring wal files` : le *wal sender* est en train de transférer les journaux nécessaires pour restaurer la sauvegarde. Cette phase n'a lieu que si l'option `--wal-method=fetch` est utilisée dans `pg_basebackup`.

1.6.5 STATISTIQUES D'UTILISATION DES WAL

1.6.5.1 Objectifs

- Mesurer l'impact des écritures dans les WAL sur les performances ;
- Statistiques calculées :
  - nombre d'enregistrements écrits dans les WAL ;
  - quantité de données écrites dans les WAL ;
  - nombre d'écritures de pages complètes.

Afin de garantir l'intégrité des données, PostgreSQL utilise le *Write-Ahead Logging* (WAL). Le concept central du WAL est d'effectuer les changements des fichiers de données (donc les tables et les index) uniquement après que ces changements ont été écrits de façon sûre dans un journal, appelé journal des transactions.

La notion d'écriture de page complète (`full page write` ou `fpw`) est l'action d'écrire une image de la page complète (`full page image` ou `fpi`) dans les journaux de transactions. Ce comportement est régi par le paramètre `full_page_write`[10] . Quand ce paramètre est activé, le serveur écrit l'intégralité du contenu de chaque page disque dans les journaux de transactions lors de la première modification de cette page qui intervient après un point de vérification (`CHECKPOINT`). Le stockage de l'image de la page complète garantit une restauration correcte de la page en cas de redémarrage suite à une panne. Ce gain de sécurité se fait au prix d'un accroissement de la quantité de données à écrire dans les journaux de transactions. Les écritures suivantes de la page ne sont que des deltas. Il est donc préférable d'espacer les checkpoints. L'écart entre les checkpoints à un impact sur la durée de la récupération après une panne, il faut donc arriver à un équilibre entre performance et temps de récupération. Cela peut être fait en manipulant `checkpoint_timeout`[11] et `max_wal_size`[12] .

L'objectif de cette fonctionnalité est de mesurer l'impact des écritures dans les journaux de transactions sur les performances. Elle permet notamment de calculer la proportion d'écritures de pages complètes par rapport au nombre total d'enregistrements écrits dans les journaux de transactions.

Elle permet de calculer les statistiques suivantes :

- nombre d'enregistrements écrits dans les journaux de transactions ;
- quantité de données écrites dans les journaux de transactions ;
- nombre d'écritures de pages complètes.

[10] https://www.postgresql.org/docs/13/runtime-config-wal.html#GUC-FULL-PAGE-WRITES
[11] https://www.postgresql.org/docs/13/runtime-config-wal.html#GUC-CHECKPOINT-TIMEOUT
[12] https://www.postgresql.org/docs/13/runtime-config-wal.html#GUC-MAX-WAL-SIZE

À l'avenir, d'autres informations relatives à la génération d'enregistrement pourraient être ajoutées.

1.6.5.2 pg_stat_statements : informations sur les WAL

Nouvelles colonnes :

- `wal_bytes` : volume d'écriture dans les WAL en octets
- `wal_records` : nombre d'écritures dans les WAL
- `wal_fpi` : nombre d'écritures de pages complètes dans les WAL.

Trois colonnes ont été ajoutées dans la vue pg_stat_statements[13] :

- `wal_bytes` : nombre total d'octets générés par la requête dans les journaux de transactions ;
- `wal_records` : nombre total d'enregistrements générés par la requête dans les journaux de transactions ;
- `wal_fpi` : nombre de total d'écritures d'images de pages complètes généré par la requête dans les journaux de transactions.

```
=# SELECT substring(query,1,100) AS query, wal_records, wal_fpi, wal_bytes
-# FROM pg_stat_statements
-# ORDER BY wal_records DESC;
| query                                          | wal_records | wal_fpi | wal_bytes |
|:-----------------------------------------------|------------:|--------:|----------:|
| UPDATE test SET i = i + $1                     |       32000 |      16 |   2352992 |
| ANALYZE                                        |        3797 |     194 |   1691492 |
| CREATE EXTENSION pg_stat_statements            |         359 |      46 |    261878 |
| CREATE TABLE test(i int, t text)               |         113 |       9 |     35511 |
| EXPLAIN (ANALYZE, WAL) SELECT * FROM pg_class  |           0 |       0 |         0 |
| SELECT * FROM pg_stat_statements               |           0 |       0 |         0 |
| CHECKPOINT                                     |           0 |       0 |         0 |
(8 rows)
```

Note : On peut voir que la commande `CHECKPOINT` n'écrit pas d'enregistrement dans les journaux de transactions. En effet, elle se contente d'envoyer un signal au processus `checkpointer`. C'est lui qui va effectuer le travail.

---

[13] https://www.postgresql.org/docs/13/pgstatstatements.html

1.6.5.3 EXPLAIN : affichage des WAL

- EXPLAIN :
  - ANALYZE : prérequis
  - WAL : affiche les statistiques d'utilisation des WAL

```
Insert on test (actual time=3.231..3.231 rows=0 loops=1)
 WAL: records=1000 bytes=65893
```

Une option WAL a été ajoutée à la commande EXPLAIN[14] . Cette option doit être utilisée conjointement avec ANALYZE.

```
=# EXPLAIN (ANALYZE, WAL, BUFFERS, COSTS OFF)
-# INSERT INTO test (i,t)
-#   SELECT x, 'x: '|| x FROM generate_series(1,1000) AS F(x);

                          QUERY PLAN
-------------------------------------------------------------------------------
Insert on test (actual time=3.410..3.410 rows=0 loops=1)
  Buffers: shared hit=1012 read=6 dirtied=6
  I/O Timings: read=0.149
  WAL: records=1000 fpi=6 bytes=70646
  ->  Function Scan on generate_series f (actual time=0.196..0.819 rows=1000 loops=1)
Planning Time: 0.154 ms
Execution Time: 3.473 ms
```

1.6.5.4 auto_explain : affichage des WAL

- auto_explain.log_analyze : prérequis
- auto_explain.log_wal : affiche les statistiques d'utilisation des journaux de transactions dans les plans
- équivalent de l'option WAL de EXPLAIN

L'extension auto_explain[15] a également été mise à jour. La nouvelle option auto_explain.log_wal contrôle si les statistiques d'utilisation des journaux de transactions sont ajoutées dans le plan d'exécution lors de son écriture dans les traces. C'est l'équivalent de l'option WAL d'EXPLAIN. Cette option n'a d'effet que si auto_explain.log_analyze est activé. auto_explain.log_wal est désactivé par défaut. Seuls les utilisateurs ayant l'attribut SUPERUSER peuvent le modifier.

---

[14] https://www.postgresql.org/docs/13/sql-explain.html
[15] https://www.postgresql.org/docs/13/auto-explain.html

1.6.5.5 autovacuum : affichage des WAL

- statistiques d'utilisation des WAL ajoutées dans les traces de l'autovacuum.

```
WAL usage: 120 records, 3 full page images, 27935 bytes
```

Lorsque l'exécution de l'autovacuum déclenche une écriture dans les traces (paramètre log_autovacuum_min_duration[16]), les informations concernant l'utilisation des journaux de transactions sont également affichées.

```
LOG:  automatic vacuum of table "postgres.pg_catalog.pg_statistic": index scans: 1
  pages: 0 removed, 42 remain, 0 skipped due to pins, 0 skipped frozen
  tuples: 214 removed, 404 remain, 0 are dead but not yet removable, oldest xmin: 613
  buffer usage: 154 hits, 1 misses, 3 dirtied
  avg read rate: 4.360 MB/s, avg write rate: 13.079 MB/s
  system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
  WAL usage: 120 records, 3 full page images, 27935 bytes
```

Les commandes `ANALYZE` et `VACUUM` ne disposent pas d'options permettant de tracer leurs statistiques d'utilisation des journaux de transactions. Cependant, il est possible de récupérer ces informations dans la vue `pg_stat_statements`.

---

[16] https://www.postgresql.org/docs/13/runtime-config-autovacuum.html#GUC-LOG-AUTOVACUUM-MIN-DURATION

1.7 PERFORMANCES

- Optimisation du stockage des B-Tree
- Tri incrémental

1.7.1 OPTIMISATION DU STOCKAGE DES B-TREE

1.7.1.1 Objectifs

- Réduction du volume d'un index en ne stockant qu'une seule fois chaque valeur
- Gain en espace disque et en performance en lecture
- Implémentation paresseuse : pas de perte de performance en écriture

Un index est une structure de données permettant de retrouver rapidement les données. L'utilisation d'un index simplifie et accélère les opérations de recherche, de tri, de jointure ou d'agrégation. La structure par défaut pour les index dans PostgreSQL est le *btree*, pour *balanced tree*.

Lorsque la colonne d'une table est indexée, pour chaque ligne de la table, un élément sera inséré dans la structure *btree*. Cette structure, dans PostgreSQL, est stockée physiquement dans des pages de 8 Ko par défaut.

La version 13 vient modifier ce comportement. Il est en effet possible pour l'index de ne stocker qu'une seule fois la valeur pour de multiples lignes.

Cette opération de déduplication fonctionne de façon paresseuse. La vérification d'une valeur déjà stockée dans l'index et identique ne sera pas effectuée à chaque insertion. Lorsqu'une page d'un index est totalement remplie, l'ajout d'un nouvel élément déclenchera une opération de fusion.

1.7.1.2 Nouveaux éléments

- Nouvelles colonnes visibles avec l'extension *pageinspect*
- Champ *allequalimage* dans `bt_metap()`
  - si *true* : possibilité de déduplication
- Champs *htid* et *tids* dans `bt_page_items()`
  - utilisés pour stocker tous les tuples indexés pour une valeur donnée

Prenons par exemple la table et l'index suivant :

```
CREATE TABLE t_dedup (i int);
CREATE INDEX t_dedup_i_idx ON t_dedup (i);
INSERT INTO t_dedup (i) SELECT g % 2 FROM generate_series(1, 4) g;
CREATE EXTENSION pageinspect;
```

Nous allons vérifier la structure interne de l'objet :

```
pg13=# SELECT itemoffset,ctid,itemlen,data,htid,tids
       FROM bt_page_items('t_dedup_i_idx', 1);
```

```
 itemoffset | ctid  | itemlen |          data           | htid  | tids
------------+-------+---------+-------------------------+-------+------
          1 | (0,2) |      16 | 00 00 00 00 00 00 00 00 | (0,2) |
          2 | (0,4) |      16 | 00 00 00 00 00 00 00 00 | (0,4) |
          3 | (0,1) |      16 | 01 00 00 00 00 00 00 00 | (0,1) |
          4 | (0,3) |      16 | 01 00 00 00 00 00 00 00 | (0,3) |
```

Pour les 4 lignes les valeurs 0 et 1, visible dans le champ *data* sont dupliquées.

Continuons d'insérer des données jusqu'à remplir la page d'index :

```
pg13=# INSERT INTO t_dedup (i) SELECT g % 2 FROM generate_series(1, 403) g;
INSERT 0 403
pg13=# SELECT count(*) FROM bt_page_items ('t_dedup_i_idx', 1);

 count
-------
   407
(1 ligne)
```

Insérons un nouvel élément dans la table :

```
thibaut=# INSERT INTO t_dedup (i) SELECT 0;
INSERT 0 1
thibaut=# SELECT count(*) FROM bt_page_items('t_dedup_i_idx', 1);
 count
-------
     3
(1 ligne)
```

Le remplissage de la page d'index a déclenché une opération de fusion en dédupliquant les lignes :

```
pg13=# SELECT itemoffset,ctid,itemlen,data,htid,tids
       FROM bt_page_items('t_dedup_i_idx', 1);
-[ RECORD 1 ]-----------------------------------------------------------------
itemoffset | 1
ctid       | (16,8395)
itemlen    | 1240
data       | 00 00 00 00 00 00 00 00
htid       | (0,2)
tids       | {"(0,2)","(0,4)","(0,6)","(0,8)","(0,10)","(0,12)","(0,14)",
           | (...)
```

```
           | "(1,170)","(1,172)","(1,174)","(1,176)","(1,178)","(1,180)"}
-[ RECORD 2 ]-----------------------------------------------------------------
itemoffset | 2
ctid       | (1,182)
itemlen    | 16
data       | 00 00 00 00 00 00 00 00
htid       | (1,182)
tids       |
-[ RECORD 3 ]-----------------------------------------------------------------
itemoffset | 3
ctid       | (16,8396)
itemlen    | 1240
data       | 01 00 00 00 00 00 00 00
htid       | (0,1)
tids       | {"(0,1)","(0,3)","(0,5)","(0,7)","(0,9)","(0,11)","(0,13)",
           |  (...)
           | "(1,171)","(1,173)","(1,175)","(1,177)","(1,179)","(1,181)"}
```

L'opération de déduplication est également déclenchée lors d'un REINDEX ainsi qu'à la création de l'index.

Cette fonctionnalité permet tout d'abord, suivant la redondance des données indexées, un gain de place non négligeable. Cette moindre volumétrie permet des gains de performance en lecture et en écriture.
D'autre part, la déduplication peut diminuer la fragmentation des index, y compris pour des index uniques, du fait de l'indexation des anciennes versions des lignes.

Il peut exister des cas très rares pour lesquels cette nouvelle fonctionnalité entraînera des baisses de performances. Par exemple, pour certaines données quasi uniques, le moteur va passer du temps à essayer de dédupliquer les lignes pour un gain d'espace négligeable. En cas de souci de performance, on pourra choisir de désactiver la déduplication :

```
CREATE INDEX t_i_no_dedup_idx ON t (i) WITH (deduplicate_items = off);
```

1.7.1.3 Limitation

- Déduplication non disponible pour plusieurs types de colonnes :
  - les types `text`, `varchar` et `char` si une collation non-déterministe est utilisée
  - le type `numeric` et par extension les types `float4`, `float8` et `jsonb`
  - les types composites, tableau et intervalle
  - les index couvrants (mot clé `INCLUDE`)

1.7.2 TRI INCRÉMENTAL

- Nouveau nœud Incremental Sorting
- Profiter des index déjà présents
- Trier plus rapidement
  - notamment en présence d'un LIMIT

PostgreSQL est capable d'utiliser un index pour trier les données. Cependant, dans certains cas, il ne sait pas utiliser l'index alors qu'il pourrait le faire. Prenons un exemple.

Voici un jeu de données contenant une table à trois colonnes, et un index sur une colonne :

```
DROP TABLE IF exists t1;
CREATE TABLE t1 (c1 integer, c2 integer, c3 integer);
INSERT INTO t1 SELECT i, i+1, i+2 FROM generate_series(1, 10000000) AS i;
CREATE INDEX ON t1(c2);
ANALYZE t1;
```

PostgreSQL sait utiliser l'index pour trier les données. Par exemple, voici le plan d'exécution pour un tri sur la colonne `c2` (colonne indexée au niveau de l'index `t1_c2_idx`) :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t1_c2_idx on t1  (cost=0.43..313749.06 rows=10000175 width=12)
                             (actual time=0.016..1271.115 rows=10000000 loops=1)
   Buffers: shared hit=81380
 Planning Time: 0.173 ms
 Execution Time: 1611.868 ms
(4 rows)
```

En revanche, si le tri concerne les colonnes c2 et c3, les versions 12 et antérieures ne savent pas utiliser l'index, comme le montre ce plan d'exécution :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather Merge  (cost=697287.64..1669594.86 rows=8333480 width=12)
          (actual time=1331.307..3262.511 rows=10000000 loops=1)
  Workers Planned: 2
```

```
  Workers Launched: 2
  Buffers: shared hit=54149, temp read=55068 written=55246
  ->  Sort  (cost=696287.62..706704.47 rows=4166740 width=12)
       (actual time=1326.112..1766.809 rows=3333333 loops=3)
       Sort Key: c2, c3
       Sort Method: external merge  Disk: 61888kB
       Worker 0:  Sort Method: external merge  Disk: 61392kB
       Worker 1:  Sort Method: external merge  Disk: 92168kB
       Buffers: shared hit=54149, temp read=55068 written=55246
       ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                              (actual time=0.015..337.901 rows=3333333 loops=3)
             Buffers: shared hit=54055
 Planning Time: 0.068 ms
 Execution Time: 3716.541 ms
(14 rows)
```

Comme PostgreSQL ne sait pas utiliser un index pour réaliser ce tri, il passe par un parcours de table (parallélisé dans le cas présent), puis effectue le tri, ce qui prend beaucoup de temps. La requête a plus que doublé en durée d'exécution.

La version 13 est beaucoup plus maligne à cet égard. Elle est capable d'utiliser l'index pour faire un premier tri des données (sur la colonne c2 d'après notre exemple), puis elle complète le tri par rapport à la colonne c3 :

```
                                     QUERY PLAN
-------------------------------------------------------------------------------
Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
           (actual time=0.082..2427.099 rows=10000000 loops=1)
 Sort Key: c2, c3
 Presorted Key: c2
 Full-sort Groups: 312500  Sort Method: quicksort
     Average Memory: 26kB  Peak Memory: 26kB
 Buffers: shared hit=81387
 -> Index Scan using t1_c2_idx on t1 (cost=0.43..313746.43 rows=10000000 width=12)
                              (actual time=0.007..1263.517 rows=10000000 loops=1)
       Buffers: shared hit=81380
Planning Time: 0.059 ms
Execution Time: 2766.530 ms
(9 rows)
```

La requête en version 12 prenait 3,7 secondes en parallélisant sur trois processus. La

version 13 n'en prend que 2,7 secondes, sans parallélisation. On remarque un nouveau type de nœud, le « Incremental Sort », qui s'occupe de re-trier les données après un renvoi de données partiellement triées, grâce au parcours d'index.

L'apport en performance est déjà très intéressant, d'autant qu'il réduit à la fois le temps d'exécution de la requête, mais aussi la charge induite sur l'ensemble du système. Cet apport en performance devient remarquable si on utilise une clause `LIMIT`. Voici le résultat en version 12 :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3 LIMIT 10;

                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=186764.17..186765.34 rows=10 width=12)
      (actual time=718.576..724.791 rows=10 loops=1)
   Buffers: shared hit=54149
   ->  Gather Merge  (cost=186764.17..1159071.39 rows=8333480 width=12)
                        (actual time=718.575..724.788 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=54149
         ->  Sort  (cost=185764.15..196181.00 rows=4166740 width=12)
                      (actual time=716.606..716.608 rows=10 loops=3)
               Sort Key: c2, c3
               Sort Method: top-N heapsort  Memory: 25kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
               Buffers: shared hit=54149
               ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12
                                    (actual time=0.010..347.085 rows=3333333 loops=3
                     Buffers: shared hit=54055
 Planning Time: 0.044 ms
 Execution Time: 724.818 ms
(16 rows)
```

Et celui en version 13 :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=0.48..1.24 rows=10 width=12) (actual time=0.027..0.029 rows=10 loops=1)
   Buffers: shared hit=4
```

```
   ->  Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
                            (actual time=0.027..0.027 rows=10 loops=1)
         Sort Key: c2, c3
         Presorted Key: c2
         Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak Memor
         Buffers: shared hit=4
         ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000 wid
                                                 (actual time=0.012..0.014 rows=11 lo
               Buffers: shared hit=4
 Planning Time: 0.052 ms
 Execution Time: 0.038 ms
(11 rows)
```

La requête passe donc de 724 ms avec parallélisation, à 0,029 ms sans parallélisation.

1.8 RÉGRESSIONS / CHANGEMENTS

- `wal_keep_segments` devient `wal_keep_size`
- Changement d'échelle du paramètre `effective_io_concurrency`

1.9 FUTUR (VERSION 14)

- Amélioration des performances avec plusieurs milliers de connexions
- `scram-sha-256` pourrait devenir l'encodage de mot de passe par défaut

1.10 ATELIERS

- Installation de PostgreSQL 13
- Monitoring : nouvelle colonne dans pg_stat_activity
- Nouveauté dans les index `b-tree`
- Nouveautés au niveau du backup
- Nouveautés dans `pg_rewind`
- Nouveauté dans la réplication logique

1.10.1 INSTALLATION DE POSTGRESQL 13

Pour une installation simple, suivre la procédure du site officiel : https://www.postgresql.org/download/linux/redhat/

Les commandes d'installation sont à effectuer avec l'utilisateur *root*.

1.10.1.1 Installation d'outils et des dépendances

```
yum install -y vim nano less rsync
yum -y install https://dl.fedoraproject.org/pub/epel/\
epel-release-latest-7.noarch.rpm
yum install jq -y
```

1.10.1.2 Installation de PostgreSQL et de la première instance

```
# Install the repository RPM

yum install -y https://download.postgresql.org/pub/repos/yum/reporpms/\
EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# Install PostgreSQL:
yum install -y postgresql13-server

# Initialisation de la base de données et démarrage automatique
/usr/pgsql-13/bin/postgresql-13-setup initdb
systemctl enable postgresql-13
systemctl start postgresql-13
```

1.10.1.3 Initialisation de la seconde instance

- Copie du fichier de configuration systemd des instance PostgreSQL :

```
cp /usr/lib/systemd/system/postgresql-13.service \
   /usr/lib/systemd/system/postgresql-13-i2.service
```

- Edition du fichier de configuration systemd de l'instance

Modification de la variable : `Environment=PGDATA=`

```
Environment=PGDATA=/var/lib/pgsql/13/instance2/
```

Ceci permet de changer le *PG_DATA* de l'instance

- Initialisation de l'instance

```
/usr/pgsql-13/bin/postgresql-13-setup initdb postgresql-13-i2
```

- Modification du port d'écoute de la nouvelle instance et initialisation :

```
sed -ie 's/^#port = 5432/port = 5433/' /var/lib/pgsql/13/instance2/postgresql.conf
systemctl enable postgresql-13-i2
systemctl start postgresql-13-i2
```

Dans le fichier de configuration nous remarquons que « `wal_keep_segments` » devient « `wal_keep_size` ». Ce paramètre est exprimé en Mo.

```
grep wal_keep_ /var/lib/pgsql/13/data/postgresql.conf
```

Génération des données pour le TP (avec l'utilisateur *postgres*) :

```
createdb pgbench
/usr/pgsql-13/bin/pgbench -i -s 466 --partitions=4 pgbench
```

Le scale factor (l'option -s) permet de fixer le nombre d'enregistrements dans la base de données. Un article intéressant[17] permet de faire le lien entre cette valeur et la taille finale de la base de données.

---

[17] https://www.cybertec-postgresql.com/en/a-formula-to-calculate-pgbench-scaling-factor-for-target-db-size/

1.10.2 MONITORING : NOUVELLE COLONNE DANS PG_STAT_ACTIVITY

Une nouvelle colonne fait son apparition dans le catalogue **pg_stat_activity** elle permet de connaitre le PID du leader pour l'interrogation parallèle.

Note : Les commandes sont à effectuer avec l'utilisateur *postgres*.

- Dans la console *psql* exécutez :

```
SELECT
  COUNT(DISTINCT leader_pid) AS nb_requetes_parallelisees,
  COUNT(leader_pid) AS parallel_workers
FROM pg_stat_activity;
```

ou pour une version plus complète :

```
SELECT
  query, leader_pid, array_agg(pid) FILTER (WHERE leader_pid != pid) AS members
FROM pg_stat_activity
WHERE leader_pid IS NOT NULL
GROUP BY query, leader_pid;
```

- Puis utiliser la métacommande `\watch 1` afin de rafraichir l'affichage

```
postgres=# \watch 1

Mon Nov 16 13:20:30 2020 (every 1s)

 pid | query | leader_pid | members
-----+-------+------------+---------
(0 rows)

(...)
```

- Se connecter à la base de données pgbench à partir d'un deuxième terminal :

```
psql -d pgbench
```

- Exécuter dans la seconde console psql la séquence suivante :

```
BEGIN;

SELECT count(*) FROM pgbench_accounts;

-- vous devez apercevoir dans votre première console quelque chose similaire à
--
--                query                  | leader_pid |    members
-- SELECT count(*) from pgbench_accounts; |      14401 | {16286,16287}
--(1 row)

-- Maintenant nous allons provoquer une erreur dans la transaction
```

```
-- afin de vérifier le nouveau comportement de psql

SELECT * FROM error;

ROLLBACK,
```

Lors de cette transaction nous observons que le prompt de *psql* a changé :

```
pgbench=*#
pgbench=!#
```

- Sur la première console :

```
                query                   | leader_pid |    members
----------------------------------------+------------+---------------
 select count(*) from pgbench_accounts; |      14401 | {14620,14621}
(1 row)
```

`leader_pid` correspond au PID de la session ayant démarré les workers dans l'exécution de la requête.

1.10.3 NOUVEAUTÉ DANS LES INDEX B-TREE

Créer les index suivants dans la base de données pgbench :

```
CREATE INDEX index_dup ON pgbench_accounts_1 (abalance) WITH (deduplicate_items = OFF);
CREATE INDEX index_dedup ON pgbench_accounts_1 (abalance) WITH (deduplicate_items = ON);
```

- Comparez la taille des 2 index.

```
SELECT pg_size_pretty(pg_relation_size('<nom_index>'));
```

- Créer un nouvel index non dédupliqué sur la clé primaire de la partition *pgbench_accounts_1*. Comparer la taille à l'index *pgbench_accounts_1_pkey*. Que constatez-vous ? Pourquoi ?

1.10.4 NOUVEAUTÉS AU NIVEAU DU BACKUP

Nous allons nous intéresser aux deux nouvelles fonctionnalités suivantes :

- Suivi de l'avancement pg_basebackup
- Fichier de manifest dans pg_basebackup

1.10.4.1 Prérequis pour la suite des exercices

Ce prérequis est indispensable pour la suite de l'atelier :

- Créer un slot de réplication. Il sera utilisé pour synchroniser une future instance secondaire dans la suite de l'atelier :

```
psql -p 5432 -c "SELECT pg_create_physical_replication_slot('secondaire')"
```

1.10.4.2 Suivi de l'avancement d'une sauvegarde par pg_basebackup

- Réaliser une sauvegarde physique avec l'outil pg_basebackup depuis l'utilisateur PostgreSQL :

```
/usr/pgsql-13/bin/pg_basebackup --format=t --gzip --pgdata=/tmp/bkp2
```

Astuce : Par default l'algorithme de checksum utilisé est CRC32C. Celui-ci est le plus performant en termes de vitesse. Vous pouvez en définir d'autre qui sont plus sûr grace au paramètre `--manifest-checksums=algorithm`. Les algorithmes disponibles sont : NONE, CRC32C, SHA224, SHA256, SHA384, ou SHA512.

- Depuis une seconde console, utiliser la console *psql* pour observer la table système pg_stat_progress_basebackup :

Afin d'obtenir un pourcentage de progression vous nous pouvons utiliser la requête suivante.

```
SELECT *, (backup_streamed / backup_total::float) * 100 AS pct
FROM pg_stat_progress_basebackup ;
\watch 2
```

La vue système pg_stat_progress_basebackup permet de connaitre la progression du backup. Attention, étant donné que la base de données est en ligne, ces données sont une estimation. Des modifications ou des insertions peuvent survenir pendant la sauvegarde et augmenter la taille totale à sauvegarder.

1.10.4.3 Fichier de manifest dans pg_basebackup

- Une fois la sauvegarde terminée, lister les fichiers présents dans */tmp/bkp2*.

On observe un fichier backup_manifest,

Pour plus de lisibilité, vous pouvez utiliser l'outil jq permettant d'afficher et parser facilement un fichier de type JSON

```
jq -C . /tmp/bkp2/backup_manifest|less -R
```

- Visualiser le fichier. Qu'observe-t-on ?

On observe un document de type JSON contenant plusieurs clefs :

- la version du fichier manifeste ;
- une liste de fichiers contenant pour chacun d'eux son nom et son emplacement, sa taille, la date de modification, l'algorithme de somme de contrôle utilisé, ainsi que la somme de contrôle ;
- en fin de fichier, une clef `WAL-Ranges` qui permet de savoir la *timeline* courante et la portion de fichiers WAL indispensables à la sauvegarde (position LSN) ;
- et enfin la somme de contrôle du fichier lui-même.
- Vérifier la sauvegarde avec la commande « `pg_verifybackup` »

```
/usr/pgsql-13/bin/pg_verifybackup -e /tmp/bkp2/
```

Nous remarquons que l'outil ne permet pas de contrôler les sauvegardes de type tar. Il faut donc extraire les fichiers pour pouvoir les contrôler.

Il y a 3 fichiers dans cette sauvegarde : * `backup_manifest` * `pg_wal.tar.gz` * `base.tar.gz`

Il nous faut extraire la sauvegarde. Nous allons le faire dans le répertoire `pg_data` de l'instance n° 2. Cela nous permettra de gagner du temps pour la création de l'instance secondaire en réplication.

- Arrêter l'instance n°2 avec l'utilisateur *root* :

```
systemctl stop postgresql-13-i2
```

- Contrôler que l'ensemble des fichiers de l'instance n°2 soit bien supprimé :

```
rm -rf /var/lib/pgsql/13/instance2/*
ls -alh /var/lib/pgsql/13/instance2/
```

- Décompresser la sauvegarde :

Le fichier `base.tar.gz` sera décompressé dans */var/lib/pgsql/13/instance2/* et `pg_wal.tar.gz` dans */var/lib/pgsql/13/instance2/pg_wal/* :

```
tar -xzvf /tmp/bkp2/base.tar.gz -C /var/lib/pgsql/13/instance2/
tar -xzvf /tmp/bkp2/pg_wal.tar.gz -C /var/lib/pgsql/13/instance2/pg_wal/
```

- Vérifier

Une fois la décompression effectuée, nous allons contrôler que l'ensemble des fichiers soit bon grâce au manifest de la sauvegarde et grâce à la commande `pg_verifybackup`.

```
/usr/pgsql-13/bin/pg_verifybackup -m /tmp/bkp2/backup_manifest \
     /var/lib/pgsql/13/instance2/
```

Astuce : vous pouvez préfixer cette commande avec la commande `time` pour connaitre le temps d'exécution.

1.10.5 MISE EN RÉPLICATION

Nous allons maintenant modifier la configuration de l'instance 2 pour la rattacher en réplication à l'instance 1.

- Créer un fichier qui sera utilisé pour l'instance 2 contenant les informations de connexion pour accéder au primaire :

```
touch /var/lib/pgsql/.pgpassinstance2
chmod 600 /var/lib/pgsql/.pgpassinstance2
```

- Créer, sur l'instance n°1, l'utilisateur utilisé pour la réplication :

```
createuser --replication -P replication
```

Lors du prompt du mot de passe, nous utiliserons le mot de passe « **dalibo** ».

- Préciser le mot de passe pour une connexion sans mot de passe :

```
echo '#hostname:port:database:username:password' >> /var/lib/pgsql/.pgpassinstance2
echo '127.0.0.1:5432:replication:replication:dalibo' >> /var/lib/pgsql/.pgpassinstance2
```

- Mettre à jour le fichier *postgresql.conf* :

Nous allons modifier les paramètres `port`, `primary_conninfo` et `primary_slot_name` afin de paramétrer la réplication.

```
sed -ie 's/^#port = 5432/port = 5433/' /var/lib/pgsql/13/instance2/postgresql.conf

# on renseigne les informations permettant au secondaire d'atteindre le primaire
sed -ie "s%^#primary_conninfo = ''%primary_conninfo = 'user=replication\
 passfile=''/var/lib/pgsql/.pgpassinstance2'' host=127.0.0.1 port=5432\
 sslmode=prefer sslcompression=0'%" /var/lib/pgsql/13/instance2/postgresql.conf

# Nous utiliserons le slot de réplication `secondaire` créé précédemment.
sed -ie "s/^#primary_slot_name = ''/primary_slot_name = 'secondaire'/" \
 /var/lib/pgsql/13/instance2/postgresql.conf
```

- Créer le fichier `standby.signal` pour indiquer à PostgreSQL que l'instance est un serveur secondaire :

```
touch /var/lib/pgsql/13/instance2/standby.signal
```

- Démarrer l'instance secondaire avec l'utilisateur *root* :

```
systemctl start postgresql-13-i2
```

- Contrôler les traces de l'instance dans /var/lib/pgsql/13/instance2/log/postgresql-Tue.log
  Contrôler rapidement que la réplication soit bien fonctionnelle. Nous pouvons utiliser la requête suivante sur le primaire :

```
SELECT client_addr, state, sent_lsn, write_lsn, flush_lsn, replay_lsn
  FROM pg_stat_replication;

SELECT slot_name, slot_type, active_pid, restart_lsn, wal_status
  FROM pg_replication_slots;
```

Nous avons à disposition une grappe PostgreSQL composée d'une instance primaire et d'une instance secondaire.

1.10.6 NOUVEAUTÉS DE PG_REWIND

1.10.6.1 Mise en place des pré-requis

Afin de pouvoir utiliser pg_rewind certains prérequis doivent être mis en place :

- **archivage des WAL**

pg_rewind a besoin de l'archivage des WAL pour pouvoir fonctionner correctement. Nous allons donc, dans un premier temps, configurer l'archivage sur les 2 instances primaire et secondaire.

- Créer les répertoires de destination des archives :

```
mkdir -p /var/lib/pgsql/archives/instance1
mkdir -p /var/lib/pgsql/archives/instance2
```

- Modifier la configuration de l'instance primaire :

```
sed -ie 's/^#archive_mode = off/archive_mode = on/'\
 /var/lib/pgsql/13/data/postgresql.conf
sed -ie "s;^#archive_command = '';archive_command =\
 '/usr/bin/rsync -a %p /var/lib/pgsql/archives/instance1/%f';"\
 /var/lib/pgsql/13/data/postgresql.conf
```

- Modifier la configuration de l'instance secondaire :

```
sed -ie 's/^#archive_mode = off/archive_mode = on/'\
 /var/lib/pgsql/13/instance2/postgresql.conf
sed -ie "s;^#archive_command = '';archive_command =\
 '/usr/bin/rsync -a %p /var/lib/pgsql/archives/instance2/%f';"\
 /var/lib/pgsql/13/instance2/postgresql.conf
sed -ie "s;^#restore_command = '';restore_command =\
 '/usr/bin/rsync -a /var/lib/pgsql/archives/instance1/%f %p';"\
 /var/lib/pgsql/13/instance2/postgresql.conf
```

- Redémarrer les deux instances avec l'utilisateur *root* :

```
systemctl restart postgresql-13
systemctl restart postgresql-13-i2
```

- Contrôler que l'archivage est bien opérationnel :

```
psql -c 'select pg_switch_wal();'
ls /var/lib/pgsql/archives/
```

Si aucun wal n'est archivé, chercher dans les traces PostgreSQL une éventuelle erreur.

- **Activation des `wal_log_hints`**

`pg_rewind` a besoin de l'activation du paramètre `wal_log_hints`. Ce paramètre fait que, après un checkpoint, le serveur PostgreSQL écrit le contenu entier de chaque page disque dans les journaux de transactions lors de la première modification de cette page. Il réalise cette écriture même pour des modifications non critiques comme les *hint bits* ou les sommes de contrôle.

- Activer les `wal_log_hints` sur les deux instances :

```
sed -ie 's/^#wal_log_hints = off/wal_log_hints = on/'\
 /var/lib/pgsql/13/data/postgresql.conf
sed -ie 's/^#wal_log_hints = off/wal_log_hints = on/'\
 /var/lib/pgsql/13/instance2/postgresql.conf
```

- Redémarrer les deux instances avec l'utilisateur *root* :

```
systemctl restart postgresql-13
systemctl restart postgresql-13-i2
```

- Créer un fichier `.pgpass` :

```
cp -a /var/lib/pgsql/.pgpassinstance2 /var/lib/pgsql/.pgpass
echo '127.0.0.1:5433:*:replication:dalibo' >> /var/lib/pgsql/.pgpass
```

1.10.6.2 Notation

Pour la suite de l'atelier :

- l'instance n°1 est l'ancienne instance primaire sur `/var/lib/pgsql/13/data`
- l'instance n°2 est l'ancienne instance secondaire sur `/var/lib/pgsql/13/instance2`

1.10.6.3 Promotion de l'instance n°2

- Promouvoir l'instance secondaire :

```
/usr/pgsql-13/bin/pg_ctl promote -D /var/lib/pgsql/13/instance2/ -w
```

Le résultat de la commande doit être semblable à :

```
waiting for server to promote.... done
server promoted
```

- Contrôler que l'instance est bien promu :

```
psql -p 5433 -c 'SELECT pg_is_in_recovery()'
```

La sortie doit être **f** :

```
pg_is_in_recovery
-------------------
 f
(1 row)
```

À partir de cet instant, il n'est plus possible de raccrocher cette instance secondaire fraichement promue.

1.10.6.4 Divergence des deux instances

Nous allons en plus ajouter des nouvelles données pour faire diverger encore plus les deux instances.

- Sur l'instance n°1 :

```
psql -p 5432 << EOF
  CREATE TABLE test1 (a int);
  INSERT INTO test1(a) SELECT y FROM generate_series(1, 100) a(y);
EOF
```

- Sur l'instance n°2 :

```
psql -p 5433 << EOF
  CREATE TABLE test2 (a int);
  INSERT INTO test2(a) SELECT y FROM generate_series(101, 200) a(y);
EOF
```

Vous pouvez utiliser la fonction `generate_series` pour ajouter encore plus de données.

1.10.6.5 Raccrochage de l'instance n°1 avec pg_rewind

Nous allons stopper l'instance n°1 de manière brutale dans le but de la raccrocher à l'instance n°2 en tant que secondaire.

- Arrêt de l'instance n°1 :

```
/usr/pgsql-13/bin/pg_ctl stop -D /var/lib/pgsql/13/data -m immediate -w
```

Note : la méthode d'arrêt recommandée est `-m fast`. L'objectif ici est de mettre en évidence les nouvelles fonctionnalités de pg_rewind.

- Donner les autorisations nécessaires à l'utilisateur de réplication, afin qu'il puisse utiliser pg_rewind :

```
psql -p 5433 <<_EOF_
GRANT EXECUTE
  ON function pg_catalog.pg_ls_dir(text, boolean, boolean)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_stat_file(text, boolean)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_read_binary_file(text)
  TO replication;
GRANT EXECUTE
  ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean)
  TO replication;
_EOF_
```

- Sauvegarder la configuration de l'instance n°1 :

```
cp /var/lib/pgsql/13/data/postgresql.conf\
 /var/lib/pgsql/postgresql.conf.backup_instance1
```

Les fichiers de configuration présents dans PGDATA seront écrasés par l'outil.

- Utiliser pg_rewind :

Afin d'observer l'ancien fonctionnement par défaut de pg_rewind, utiliser le paramètre `--no-ensure-shutdown`.

```
/usr/pgsql-13/bin/pg_rewind \
 --target-pgdata /var/lib/pgsql/13/data/ \
 --source-server "host=127.0.0.1 port=5433 user=replication dbname=postgres" \
 --write-recovery-conf --no-ensure-shutdown \
 --progress --dry-run
```

Un message d'erreur nous informe que l'instance n'a pas été arrêtée proprement :

```
pg_rewind: connecté au serveur
```

```
pg_rewind: fatal : le serveur cible doit être arrêté proprement
```

Relancer `pg_rewind`, sans le paramètre `--no-ensure-shutdown` ni `--dry-run` (qui empêche de réellement rétablir l'instance), afin d'observer le nouveau fonctionnement par défaut :

```
/usr/pgsql-13/bin/pg_rewind \
 --target-pgdata /var/lib/pgsql/13/data/ \
 --source-server "host=127.0.0.1 port=5433 user=replication dbname=postgres" \
 --write-recovery-conf \
 --progress
```

- Configurer l'instance n°1 restaurée :

Une fois l'opération réussie, restaurer le fichier de configuration d'origine sur l'ancienne primaire et y ajouter la configuration de la réplication.

```
# récupération de la configuration initiale
cp /var/lib/pgsql/postgresql.conf.backup_instance1 \
 /var/lib/pgsql/13/data/postgresql.conf

# on renseigne les informations permettant au secondaire d'atteindre le primaire
sed -ie "s%^#primary_conninfo = ''%primary_conninfo = 'user=replication\
 passfile=''/var/lib/pgsql/.pgpassinstance1'' host=127.0.0.1 port=5433\
 sslmode=prefer sslcompression=0'%" /var/lib/pgsql/13/data/postgresql.conf

# Nous utiliserons le slot de réplication `secondaire`
sed -ie "s/^#primary_slot_name = ''/primary_slot_name = 'secondaire'/" \
 /var/lib/pgsql/13/data/postgresql.conf
```

- Créer le slot de réplication sur l'instance n°2 :

```
psql -p 5433 -c "SELECT pg_create_physical_replication_slot('secondaire')"
```

- Relancer l'instance n°1 avec l'utilisateur *root* :

```
systemctl restart postgresql-13
```

- Contrôler dans les journaux applicatifs le déroulé des opérations.
- Révoquer les droits de l'utilisateur *replication* :

Une fois l'opération terminée, n'oubliez pas de révoquer les droits ajoutés à l'utilisateur *réplication*.

```
psql -p 5433 <<_EOF_
REVOKE EXECUTE
  ON function pg_catalog.pg_ls_dir(text, boolean, boolean)
  FROM  replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_stat_file(text, boolean)
```

```
  FROM replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_read_binary_file(text)
  FROM replication;
REVOKE EXECUTE
  ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean)
  FROM replication;
_EOF_
```

1.10.6.6 Possible erreur

L'utilisation de pg_rewind peut être compliquée à cause d'un problème d'accès au WAL.

pg_rewind redémarre PostgreSQL en mode mono-utilisateur afin de réaliser une récupération de l'instance. L'opération échoue, car PostgreSQL n'arrive pas à trouver les fichiers WAL dans le répertoire *PGDATA/pg_wal* de l'instance principale.

pg_rewind a besoin des WAL archivés. Il faut contrôler la restore_command de l'instance :

```
grep restore_command /var/lib/pgsql/13/data/postgresql.conf
```

Si la restore_command n'est pas correctement positionnée, modifiez-la puis relancer la commande pg_rewind avec avec l'option --restore-target-wal :

```
/usr/pgsql-13/bin/pg_rewind \
 --target-pgdata /var/lib/pgsql/13/data/ \
 --source-server "host=127.0.0.1 port=5433 user=replication dbname=postgres" \
 --write-recovery-conf \
 --restore-target-wal
 --progress
```

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegardes et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **DEVSQLPG : SQL pour PostgreSQL**
  https://dali.bo/devsqlpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancé**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle vers PostgreSQL**
  https://dali.bo/migorpg

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.