Workshop 13

Nouveautés de PostgreSQL 13

13

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 13

Workshop 13

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

REVISION: 13

Table des Matières

1 Nouveautés de PostgreSQL 13 **8**

1.1 Les nouveautés . . . 9
1.2 Administration & maintenance . . . 10
1.2.1 Autovacuum : déclenchement par INSERT . . . 10
1.2.2 VACUUM : nouveaux workers . . . 11
1.2.3 vacuumdb --parallel . . . 13
1.2.4 reindexdb --jobs . . . 13
1.3 Administration : amélioration de la sauvegarde physique . . . 14
1.3.1 Fichiers manifestes . . . 14
1.3.2 Nouvel outil pg_verifybackup . . . 16
1.3.3 Suivi de l'exécution des sauvegardes . . . 18
1.3.4 Erreur fatale quand `recovery_target` ne peut être atteinte . . . 20
1.4 Administration : divers . . . 22
1.4.1 Déclencheurs BEFORE sur les partitions . . . 22
1.4.2 Nouveau paramètre maintenance_io_concurrency . . . 23
1.4.3 Nouveau paramètre ignore_invalid_pages . . . 24
1.4.4 Déconnexion des utilisateurs à la suppression d'une base de données 25
1.4.5 Extensions de confiance . . . 26
1.4.6 Prompt de psql . . . 29
1.5 Réplication physique . . . 32
1.5.1 Modification à chaud des paramètres de réplication . . . 32
1.5.2 Volume maximal de journaux conservé par les slots . . . 33
1.5.3 pg_rewind sait restaurer des journaux . . . 35
1.5.4 pg_rewind récupère automatiquement une instance . . . 36
1.5.5 pg_rewind génère la configuration de réplication . . . 37
1.6 Réplication logique . . . 38
1.6.1 Publication d'une table partitionnée . . . 38
1.6.2 Consommation mémoire du décodage logique . . . 39
1.7 Supervision . . . 40
1.7.1 Tracer le type de processus dans les journaux . . . 40
1.7.2 Échantillonner les requêtes . . . 42
1.7.3 Paramètres des requêtes préparées . . . 43
1.7.4 pg_stat_statements : temps de planification . . . 45
1.7.5 pg_stat_activity : nouveau champ leader_pid . . . 47
1.7.6 Nouvelle vue pg_stat_progress_analyze . . . 49
1.7.7 Nouvelle vue pg_shmem_allocations . . . 50
1.8 Performances . . . 52

1.8.1 Déduplication des index B-Tree . . . 52
1.8.2 Tri incrémental . . . 56
1.8.3 Paramétrage du détail pour les statistiques étendues . . . 60
1.8.4 Hash Aggregate : débord sur disque . . . 61
1.8.5 Statistiques d'utilisation des WAL . . . 63
1.8.6 EXPLAIN : Utilisation disque du planificateur . . . 67
1.9 Régressions . . . 68
1.9.1 wal_keep_segments devient wal_keep_size . . . 68
1.9.2 effective_io_concurrency : changement d'échelle . . . 69

2 Ateliers 70
2.1 TP - Partitionnement, déclencheur BEFORE . . . 70
2.1.1 Création de la table partitionnée . . . 70
2.1.2 Création d'un déclencheur BEFORE . . . 71
2.1.3 Déclencheur BEFORE qui modifie la partition cible . . . 72
2.1.4 Supprimer le déclencheur . . . 73
2.2 TP - Fichiers manifeste et vérification des sauvegardes . . . 74
2.2.1 Création de la sauvegarde . . . 74
2.2.2 Fichier manifeste . . . 75
2.2.3 Vérifier les sauvegardes . . . 75
2.3 TP - Colonne leader_pid dans pg_stat_activity . . . 78
2.4 TP - Suivi de l'exécution des sauvegardes . . . 79
2.4.1 Mise en place . . . 79
2.4.2 Sauvegarde simple . . . 79
2.4.3 Sauvegarde sans estimation de la taille . . . 81
2.5 TP - Progression de la commande ANALYZE . . . 82
2.5.1 Mise en place des tables et données . . . 82
2.5.2 ANALYZE et observations . . . 82
2.6 TP - *HashAggregate*, débord sur disque . . . 85
2.7 TP - Statistiques d'utilisation des WAL . . . 89
2.7.1 Prérequis . . . 89
2.7.2 Mise en place du test . . . 90
2.7.3 Consultation des statistiques . . . 90
2.8 TP - Réplication logique et partitionnement . . . 92
2.8.1 Prérequis . . . 92
2.8.2 Ajout d'une table partitionnée à une publication . . . 92
2.8.3 Réplication vers une table partitionnée ayant le même schéma . . . 96
2.8.4 Réplication vers une table partitionnée ayant un schéma différent . . . 97
2.9 TP - Changement à chaud des informations de réplication . . . 100

2.9.1 Mise en place du primaire 100
2.9.2 Mise en place du secondaire 101
2.9.3 Activité sur le secondaire 102
2.9.4 Modification des informations de connexion et redémarrage 103
2.9.5 Utilisateur dédié et .pgpass 104
2.9.6 Changement d'instance principale 105
2.10 TP - Volume maximal de journaux conservé par les slots 108
2.10.1 Création d'un slot 108
2.10.2 Rétention des slots 109
2.10.3 Générer de l'activité et observer le nouveau fonctionnement 109
2.11 TP - Outil pg_rewind 112
2.11.1 Création d'une instance primaire 112
2.11.2 Mettre en place la réplication 113
2.11.3 Simulation d'un failover 114
2.11.4 Utilisation de pg_rewind 114
2.12 TP - wal_keep_segments et wal_keep_size 118
2.13 TP - Déduplication des index B-Tree 120

1 NOUVEAUTÉS DE POSTGRESQL 13

Photographie de Rad Dougall, licence CC BY 3.0[1] , obtenue sur wikimedia.org[2] .

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[3]

Ce workshop sera maintenu encore plusieurs mois après la sortie de la version 13.

---

[1] https://creativecommons.org/licenses/by/3.0/deed.en
[2] https://commons.wikimedia.org/wiki/File:The_Big_Boss_Elephant_(190898861).jpeg
[3] https://github.com/dalibo/workshops/blob/master/LICENSE.md

1.1 LES NOUVEAUTÉS

- Administration :
  - maintenance
  - sauvegarde physique
  - divers
- Réplication physique & logique
- Supervision
- Performances
- Régressions
- Ateliers

PostgreSQL 13 est sorti le 24 septembre 2020.

Les points principaux sont décrits dans le « press kit[4] ».

Nous allons décrire ces nouveautés plus en détail.

---

[4] https://www.postgresql.org/about/press/presskit13/fr/

1.2 ADMINISTRATION & MAINTENANCE

- VACUUM & autovacuum
- Réindexation parallélisée

1.2.1 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.

1.2.2 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
```

1.2.3 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.

1.2.4 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 ADMINISTRATION : AMÉLIORATION DE LA SAUVEGARDE PHYSIQUE

- Fichiers manifeste
- Suivi de la sauvegarde
- Restauration & `recovery_target` non atteinte

1.3.1 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

algorithme	nombre de passes	temps moyen (s)
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 algorithmes 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.

1.3.2 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.

1.3.3 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`[5] 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.

[5] 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`.

1.3.4 ERREUR FATALE QUAND `RECOVERY_TARGET` NE PEUT ÊTRE ATTEINTE

Erreur fatale quand la cible précisée n'est pas atteinte en fin de restauration :
`FATAL: recovery ended before configured recovery target was reached`

Avant PostgreSQL 13, si une cible de restauration est configurée, mais que le rejeu des archives s'arrête avant que cette cible ne soit atteinte, l'instance finissait son démarrage normalement.

Voici un exemple de traces avec une `recovery_target_timeline` qui ne peut être atteint sur une instance PostgreSQL 12.

```
LOG:  database system was interrupted; last known up at 2020-11-24 15:36:08 CET
ERROR: could not find /backup/pitrery-pgsql-12/archived_wal/00000002.history
LOG:  starting point-in-time recovery to 2020-11-24 19:00:00+01
LOG:  restored log file "000000010000000180000005B" from archive
LOG:  redo starts at 18/5B000028
LOG:  consistent recovery state reached at 18/5B000138
LOG:  database system is ready to accept read only connections
ERROR: could not find /backup/pitrery-pgsql-12/archived_wal/000000010000000180000005C
LOG:  redo done at 18/5B000138
LOG:  restored log file "000000010000000180000005B" from archive
ERROR: could not find /backup/pitrery-pgsql-12/archived_wal/00000002.history
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
ERROR: could not find /backup/pitrery-pgsql-12/archived_wal/00000001.history
LOG:  database system is ready to accept connections
```

Dans ces messages, l'instance ne produit pas de message d'erreur en lien avec le `recovery_target_timeline`. Les erreurs remontées sont normales dans le cadre d'une restauration. Le dernier message nous informe que l'instance a démarré.

Avec la version 13 de PostgreSQL, si la cible de restauration ne peut être atteinte, une erreur fatale est émise et l'instante s'arrête :

```
LOG:  database system was interrupted; last known up at 2020-11-24 15:48:09 CET
ERROR: could not find /backup/pitrery-pgsql-13/archived_wal/00000002.history
LOG:  starting point-in-time recovery to 2020-11-24 19:00:00+01
LOG:  restored log file "000000010000000B000000EA" from archive
LOG:  redo starts at B/EA000028
LOG:  consistent recovery state reached at B/EA000100
LOG:  database system is ready to accept read only connections
ERROR: could not find /backup/pitrery-pgsql-13/archived_wal/000000010000000B000000EB
```

```
LOG:  redo done at B/EA000100
FATAL:  recovery ended before configured recovery target was reached
LOG:  startup process (PID 229173) exited with exit code 1
LOG:  terminating any other active server processes
LOG:  database system is shut down
```

L'instance produit bien désormais un message d'erreur `FATAL` en lien avec la `recovery_target_timeline`. Le dernier message nous informe que l'instance est arrêtée.

Cette modification de comportement à deux intérêts :

1. Informer l'administrateur du fait que la restauration ne s'est pas déroulée comme prévu.
2. Permettre d'ajouter des WAL pour la restauration et d'atteindre la `recovery_target` dans le cas où on aurait oublié de tous les fournir.

1.4 ADMINISTRATION : DIVERS

- Partitionnement : déclencheurs BEFORE
- Nouveaux paramètres :
  - `ignore_invalid_pages`
  - `maintenance_io_concurrency`
- `DROP DATABASE` & déconnexion forcée
- Extensions de confiance
- Prompt de psql

1.4.1 DÉCLENCHEURS BEFORE SUR LES PARTITIONS

- possibilité de créer des déclencheurs `BEFORE` sur les tables partitionnées.

La version 11 de PostgreSQL a permis d'ajouter le support des déclencheurs sur les tables partitionnées … à l'exception des déclencheurs `BEFORE`. Ces derniers devaient donc être créés manuellement sur les partitions.

Avec la version 13, l'ajout d'un déclencheur `BEFORE` sur une table partitionnée permet de créer automatiquement les déclencheurs sur les partitions associées. Il y a cependant une limitation : ces déclencheurs ne peuvent pas changer la partition cible d'une ligne.

Voici un exemple de ce que l'on peut voir quand on affiche les détails d'une table partitionnée :

```
\d+ log

                  Partitioned table "public.log"
...
Triggers:
   log_user BEFORE INSERT ON log FOR EACH ROW EXECUTE FUNCTION log_user()
Partitions:
   log_202011 FOR VALUES FROM ('2020-11-01 00:00:00') TO ('2020-12-01 00:00:00'),
   log_202012 FOR VALUES FROM ('2020-12-01 00:00:00') TO ('2021-01-01 00:00:00')

\d log_202011
                    Table "public.log_202011"
...
Partition of: log FOR VALUES FROM ('2020-11-01 00:00:00') TO ('2020-12-01 00:00:00')
Triggers:
```

```
log_user BEFORE INSERT ON log_202011 FOR EACH ROW EXECUTE FUNCTION log_user(),
            ON TABLE log
```

1.4.2 NOUVEAU PARAMÈTRE MAINTENANCE_IO_CONCURRENCY

- Nouveau paramètre `maintenance_io_concurrency`
- Permet d'augmenter le nombre d'I/O sur les opérations de maintenance
- Équivalent à `effective_io_concurrency`

Il s'agit du « nombre de requêtes simultanées que les disques peuvent assurer efficacement pour les opérations de maintenance », comme le `VACUUM`. Ce nouveau paramètre a un rôle équivalent à `effective_io_concurrency`, qui concerne une seule session. En résumé, il permet d'estimer la capacité à faire des lectures en avance de phase sur le stockage (*prefetch*).

Le choix de sa valeur peut être délicat. `effective_io_concurrency` vaut par défaut 1, et s'estime à partir du nombre de disques (hors ceux de parité) d'une grappe de disques RAID, avec des valeurs de 500 ou plus pour des SSD (noter que les valeurs de `effective_io_concurrency` doivent être plus élevées en version 13). `maintenance_io_concurrency` vaut par défaut 10, donc une valeur nettement supérieure, car les opérations de maintenance ne sont pas censées être nombreuses en parallèle. Il faudra donc le changer aussi si `effective_io_concurrency` est modifié.

Comme `effective_io_concurrency`, ce paramètre peut être modifié au niveau de chaque tablespace avec `ALTER TABLESPACE`, si les caractéristiques physiques diffèrent.

1.4.3 NOUVEAU PARAMÈTRE IGNORE_INVALID_PAGES

- `ignore_invalid_pages` permet de continuer la récupération quand les WAL font référence à des pages invalides
- Peut (et va) causer des crashs, pertes de données, cacher et propager des corruptions
- Permet de démarrer en cas de corruption
- À utiliser sur une copie de l'instance incidentée

La détection d'enregistrements de WAL qui font références à des pages invalides pendant la récupération d'une instance cause normalement une erreur de type *PANIC*. Ce qui interrompt la récupération et le démarrage de PostgreSQL.

Activer le paramètre `ignore_invalid_pages` permet au système d'ignorer ces enregistrements et de continuer la récupération après avoir écrit un message d'erreur de type *WARNING* dans les traces de l'instance.

Il est important de travailler sur une **copie** de l'instance incidentée lorsqu'on utilise ce paramètre. En effet, son utilisation peut causer des crashs, des pertes de données, ainsi que propager ou cacher des corruptions. Elle permet cependant de continuer la récupération et démarrer le serveur dans des situations désespérées.

1.4.4 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
```

1.4.5 EXTENSIONS DE CONFIANCE

- Objet extension depuis la version 9.1
- Installation uniquement par un superutilisateur
- Apparaît la notion de *trusted extension*
  - installation par les utilisateurs ayant le droit CREATE sur la base
- Quelques extensions des *contribs* sont déclarées TRUSTED

Les extensions existent depuis la version 9.1 de PostgreSQL. Leur principal inconvénient est que seul un superutilisateur peut les installer. C'est une mesure de sécurité. Les langages sont aussi des extensions, et cette mesure de sécurité a été vue comme une régression par rapport aux versions précédentes. En effet, sur les versions antérieures à la 9.1, le propriétaire d'une base de données pouvait installer un langage sans avoir besoin d'un superutilisateur pour le faire. Cette régression était déjà gênante mais le problème a empiré avec l'arrivée des PaaS et du cloud. En effet, dans ce cas, l'acheteur d'une solution cloud avec PostgreSQL se trouve généralement propriétaire de la base, mais n'est jamais superutilisateur. Il n'a donc aucun moyen d'installer les extensions qu'il souhaite. Soit elles sont préinstallées, soit il doit demander l'installation, soit il doit s'en passer.

La version 13 améliore cela en proposant des extensions que tout utilisateur ayant le droit CREATE sur une base peut installer lui-même. Toutes les extensions ne le permettent pas. Elles doivent avoir l'attribut TRUSTED. Certaines extensions fournies dans les *contribs* ont cet attribut, d'autres non. Tout dépend du contexte d'utilisation de l'extension et des potentiels risques au niveau de la sécurité.

Voici quelques requêtes sur le catalogue pour montrer la notion d'extension de confiance :

```
-- Nombre d'extensions disponibles depuis les modules contrib

SELECT count(*) FROM pg_available_extensions;

 count
-------
    43
(1 row)

-- Nombre d'extensions TRUSTED et non TRUSTED

SELECT trusted, count(DISTINCT name)
FROM pg_available_extension_versions
GROUP BY 1;
```

```
 trusted | count
---------+-------
 f       |    22
 t       |    21
(2 rows)

-- Liste d'extensions TRUSTED

SELECT name, max(version) AS version_max
FROM pg_available_extension_versions
WHERE trusted
GROUP BY 1
ORDER BY 1;

      name       | version_max
-----------------+-------------
 btree_gin       | 1.3
 btree_gist      | 1.5
 citext          | 1.6
 cube            | 1.4
 dict_int        | 1.0
 fuzzystrmatch   | 1.1
 hstore          | 1.7
 intarray        | 1.3
 isn             | 1.2
 lo              | 1.1
 ltree           | 1.2
 pg_trgm         | 1.5
 pgcrypto        | 1.3
 plpgsql         | 1.0
 seg             | 1.3
 tablefunc       | 1.0
 tcn             | 1.0
 tsm_system_rows | 1.0
 tsm_system_time | 1.0
 unaccent        | 1.1
 uuid-ossp       | 1.1
(21 rows)
```

Et voici un exemple d'installation d'une extension à partir d'un utilisateur ayant le droit

CREATE sur la base :

```
-- création de la base b1
postgres=# create database b1;
CREATE DATABASE

-- création du rôle u1
postgres=# create role u1 login;
CREATE ROLE

-- affectation du droit CREATE sur la base b1 pour le rôle u1
postgres=# grant create on database b1 to u1;
GRANT

-- connexion à b1 en tant que u1
postgres=# \c b1 u1
You are now connected to database "b1" as user "u1".

-- installation (réussie) d'une extension TRUSTED
b1=> create extension hstore;
CREATE EXTENSION

-- installation (échouée) d'une extension NON TRUSTED
b1=> create extension pg_buffercache;
ERROR:  permission denied to create extension "pg_buffercache"
HINT:  Must be superuser to create this extension.
```

1.4.6 PROMPT DE PSQL

- Prompt psql modifié
- Intègre maintenant l'état de la transaction
- L'étoile représente une transaction valide en cours
- Le point d'exclamation représente une transaction erronée en cours

Le prompt par défaut de `psql` a été modifié pour y ajouter le joker `%x`. Ce dernier indique l'état de la transaction (valide ou erronnée, mais en cours) quand une transaction a été ouverte explicitement. En voici quelques exemples :

```
-- pas de transaction ouverte, le prompt ressemble à l'ancien
b1=> BEGIN;
BEGIN

-- maintenant que la transaction est ouverte, une étoile indique qu'on est
-- dans une transaction explicite
b1=*> CREATE TABLE t1(id integer);
CREATE TABLE

b1=*> SELECT * FROM t1;
 id
----
(0 rows)

b1=*> INSERT INTO t1 VALUES (10);
INSERT 0 1

b1=*> SELECT * FROM t1;
 id
----
 10
(1 row)

-- cette étoile reste présente jusqu'à la fin de la transaction
b1=*> COMMIT;
COMMIT

-- voilà, COMMIT ou ROLLBACK exécuté, on revient à l'ancien prompt
b1=> BEGIN;
```

```
BEGIN

-- nouvelle transaction explicite, l'étoile revient
b1=*> CREATE TABLE t1(id integer);
ERROR:  relation "t1" already exists

-- la transaction est en erreur, l'étoile est remplacée par un point
-- d'exclamation
b1=!> SELECT * FROM t1;
ERROR:  current transaction is aborted,
        commands ignored until end of transaction block

-- ce dernier restera jusqu'à l'exécution d'un ROLLBACK
b1=!> ROLLBACK;
ROLLBACK

-- cela fonctionne aussi avec un ROLLBACK TO vers un savepoint créé avant
-- l'erreur
b1=> BEGIN;
BEGIN

b1=*> SAVEPOINT sp1;
SAVEPOINT

b1=*> CREATE TABLE t1(id integer);
ERROR:  relation "t1" already exists

b1=!> SELECT * FROM t1;
ERROR:  current transaction is aborted,
        commands ignored until end of transaction block

b1=!> ROLLBACK TO sp1;
ROLLBACK

-- on se retrouve bien avec l'étoile
b1=*> SELECT * FROM t1;
 id
----
 10
```

```
(1 row)

b1=*> COMMIT;
COMMIT
```

De ce fait, l'étoile est utilisée pour indiquer une transaction valide en cours et le point d'exclamation pour une transaction en erreur en cours.

1.5 RÉPLICATION PHYSIQUE

- Modification à chaud des paramètres de réplication
- Slots : volume maximal de journaux (`max_slot_wal_keep_size`)
- pg_rewind : nouvelles fonctionnalités

1.5.1 MODIFICATION À CHAUD DES PARAMÈTRES DE RÉPLICATION

- Plus besoin de redémarrer un secondaire pour modifier les paramètres de réplication
- Notamment les paramètres `primary_conninfo` et `primary_slot_name`
- Évite la déconnexion des utilisateurs

Quand le paramétrage de la réplication d'un serveur secondaire était modifié (paramètres `primary_conninfo` et `primary_slot_name` notamment), il était auparavant nécessaire de redémarrer l'instance pour tenir compte de ces modifications, ce qui coupait les connexions en cours.

À présent, il suffit d'une simple relecture de configuration sur le secondaire après modification des paramètres. La modification peut se faire en éditant le fichier `postgresql.conf`, n'importe quel fichier inclu, ou encore à l'aide de la requête `ALTER SYSTEM SET <nom> TO <valeur>`. Le rechargement quant à lui peut être effectué à l'aide de la requête `SELECT pg_reload_conf()` ou encore avec la commande `pg_ctl -D "$PGDATA" reload`, etc.

Notez que le mot de passe associé à l'utilisateur spécifié dans le paramètre `primary_conninfo` est souvent spécifié dans un fichier `.pgpass`, qui doit être édité séparément et dont la prise en compte ne nécessite pas de rechargement de la configuration.

En revanche, cette amélioration simplifiera la suppression d'un slot, un changement d'utilisateur de réplication ou le renforcement de la configuration SSL.

Elle simplifiera surtout, dans une configuration à trois serveurs ou plus, le raccrochage d'un secondaire à un nouveau primaire suite à une bascule, ou le passage à une réplication en cascade.

1.5.2 VOLUME MAXIMAL DE JOURNAUX CONSERVÉ PAR LES SLOTS

- `max_slot_wal_keep_size` permet de spécifier le volume maximal de WAL que les slots de réplication peuvent conserver dans le répertoire `pg_wal` ;
- Ajout des colonnes `wal_status` et `safe_wal_size` à la vue `pg_replication_slots` pour permettre de suivre l'état des slots.

En cas de déconnexion d'une instance secondaire ou de retard important de la réplication, les slots de réplication permettent de conserver la quantité exacte de WAL nécessaire à l'instance secondaire pour qu'elle puisse poursuivre sa réplication.

Précédemment, la volumétrie de WAL conservée par les slots de réplication dans le répertoire de `pg_wal` n'avait pas de limite. Cette absence de limite était un problème en cas d'indisponibilité prolongée de l'instance secondaire. En effet, elle met en péril la continuité du service sur l'instance primaire en menaçant de remplir le répertoire `pg_wal`.

Le paramètre `max_slot_wal_keep_size` permet de limiter la quantité de WAL conservé par les slots. Il peut être modifié a chaud. Sa valeur par défaut est `-1`, signifiant que les slots conservent une quantité illimitée de WAL.

Afin de suivre l'état des slots de réplication, la vue `pg_replication_slots` a été enrichie avec deux nouvelles colonnes.

La première, `wal_status` permet de connaître la disponibilité des WAL réservés par le slot. Elle peut prendre quatre valeurs :

- `reserved` : le quantité de wal réservé est inférieure à `max_wal_size`.
- `extended` : le quantité de wal réservé est supérieure à `max_wal_size` mais les WAL sont conservés soit par le slot lui-même, soit par le biais du paramètre `wal_keep_size`.
- `unreserved` : le slot ne conserve plus de WAL et certains seront retirés lors du prochain `CHECKPOINT`. Cet état est réversible.
- `lost` : les WAL nécessaires au slot ne sont plus disponibles et le slot est inutilisable.

Les deux derniers statuts ne sont possibles que lorsque `max_slot_wal_keep_size` est supérieur ou égal à zéro.

La seconde colonne, `safe_wal_size` contient le volume de WAL en octet qui peut être produit sans mettre en danger le slot. Cette valeur est nulle lorsque le statut du slot est `lost` ou que `max_slot_wal_keep_size` est égal à `-1`.

Attention, le volume maximal de WAL conservés par un slot reste dans le répertoire `pg_wal` jusqu'à ce que le slot soit supprimé, même pour les slots dont le statut est `lost`. Leur volumétrie s'ajoute donc à la volumétrie normale des journaux de transaction. C'est

un facteur à prendre en compte quand on dimensionne un système de fichier dédié à `pg_wal`.

1.5.3 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`[6] 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`.

---

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

1.5.4 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.

1.5.5 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.

1.6 RÉPLICATION LOGIQUE

- Publication de table partitionnée
- Consommation mémoire des walsenders

1.6.1 PUBLICATION D'UNE TABLE PARTITIONNÉE

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[7] :

```
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[8] ou ALTER PUBLICATION[9] .

Exemple :

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

```
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.

1.6.2 CONSOMMATION MÉMOIRE DU DÉCODAGE LOGIQUE

- Nouveau paramètre `logical_decoding_work_mem`
  - Défaut : 64 Mo par session
- Contrôle la mémoire allouée au décodage logique avant de déborder sur disque
- Concerne toute session consommant un slot logique, y compris les walsenders
- Meilleur contrôle de la consommation mémoire des walsenders

Le décodage logique des journaux applicatifs peut consommer beaucoup de mémoire sur l'instance d'origine. Il n'existait jusqu'à présent aucun moyen de contrôler la quantité de mémoire réservée à cette opération. L'opération pouvait déborder sur disque une transaction seulement si le nombre de changements de cette dernière était supérieur à 4096. Or, les transactions étant entremêlées dans les journaux de transaction, le décodage peut rapidement mener à maintenir en mémoire les données de plusieurs d'entre elles en même temps, avant de pouvoir les envoyer au destinataire. Il n'était pas possible de modifier ce comportement.

Le nouveau paramètre `logical_decoding_work_mem` permet désormais de contrôler finement à quel moment le décodage d'une transaction doit déborder sur disque en définissant une limite en matière de mémoire consommée. Sa valeur par défaut est de `64MB`.

Il est donc possible de limiter la consommation mémoire des `walsenders` en abaissant ce paramètre, au prix d'une perte de performance, d'une latence supplémentaire sur la réplication logique et d'I/O supplémentaires. Au contraire, augmenter ce paramètre permet de privilégier la réplication logique et sa latence au prix d'une consommation mémoire supérieure et d'I/O supplémentaires.

Le paramètre peut être modifié pour tout le monde, ou au sein d'une session (par exemple, en utilisant l'API SQL de décodage), ou encore pour un rôle de réplication logique ou une base de donnée seulement.

1.7 SUPERVISION

- Journaux & type de processus
- Échantillonnage des requêtes
- Requêtes préparées : paramètres
- pg_stat_statements : temps de planification
- pg_stat_statements : `leader_pid`
- Vue `pg_stat_progress_analyze`
- Vue `pg_shmem_allocations`

1.7.1 TRACER LE TYPE DE PROCESSUS DANS LES JOURNAUX

- 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'`).

1.7.2 ÉCHANTILLONNER LES REQUÊTES

- `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.

1.7.3 PARAMÈTRES DES REQUÊTES PRÉPARÉES

- `log_parameter_max_length` permet de définir le volume de paramètres maximal associé aux requêtes préparées dans les traces ;
- ces valeurs sont notamment associées à leur requête préparée par les paramètres `log_min_duration_statements` ou `log_min_duration_sample` ;
- `log_parameter_max_length_on_error` permet de définir le volume de paramètres maximal affiché dans les traces des requêtes préparées à cause d'erreurs.

Deux nouveaux paramètres GUC ont été ajoutés pour contrôler l'affichage des paramètres associés (*bind*) aux requêtes préparées dans les traces de PostgreSQL.

`log_parameter_max_length` s'applique aux requêtes préparées tracées grâce à des paramètres comme `log_min_duration_statement` et `log_min_duration_sample`. Par défaut, ils sont affichés dans leur intégrité (valeur `-1` du paramétrage). Il est également possible de désactiver complètement leur affichage avec la valeur 0 ou de spécifier une limite en octet. Ce paramètre ne nécessite pas de redémarrage pour être modifié. En revanche, il faut employer un utilisateur bénéficiant de l'attribut `SUPERUSER` pour le faire au sein d'une session.

`log_parameter_max_length_on_error` concerne les requêtes préparées écrites dans les traces à cause d'erreurs. Par défaut, l'affichage de leurs paramètres est désactivé (valeur `0`). Cela peut être modifié à chaud par n'importe quel utilisateur, dans sa session, en spécifiant une taille en octet ou `-1` pour tout afficher.

Le comportement par défaut correspond à celui observable sur la version précédente de PostgreSQL.

Exemple :

```
$ cat ~/tmp/bench.script
SET log_parameter_max_length_on_error TO -1;
SET log_parameter_max_length TO 5;
SET log_min_duration_statement TO 1500;

\SET one 0123456789

SELECT pg_sleep(2), :one, 'logged';
SELECT pg_sleep(1), :one, 'not logged';
SELECT 1/0, :one, 'logged';

$ pgbench -c1 -t1 -M prepared -n -f ~/tmp/bench.script
```

Produit les traces :

```
LOG:  duration: 2002.175 ms  execute <unnamed>: SELECT pg_sleep(2), $1, 'logged';
DETAIL:  parameters: $1 = '12345...'
ERROR:  division by zero
CONTEXT:  unnamed portal with parameters: $1 = '123456789'
STATEMENT:  SELECT 1/0, $1, 'logged';
```

1.7.4 PG_STAT_STATEMENTS : TEMPS DE PLANIFICATION

- `pg_stat_statements` peut désormais collecter pour chaque requête le nombre de phases d'optimisation et le temps qui y est alloué ;
- `pg_stat_statements.track_planning` permet d'activer cette collecte. Sa valeur par défaut est `off`.

La collecte de statistiques sur le nombre de phases d'optimisation et leur durée peut être effectuée dans la vue `pg_stat_statements` grâce l'option `pg_stat_statements.track_planning`. Activer ce paramètre pourrait provoquer une perte visible de performance, spécialement quand peu de requêtes du même genre sont exécutées sur de nombreuses connexions concurrentes. La valeur par défaut est `off`. Seuls les superutilisateurs peuvent modifier cette configuration au sein d'une session.

Les colonnes suivantes sont alimentées lorsque ce paramètre est activé

- `plans` : Nombre d'optimisations de la requête.
- `total_plan_time` : Durée totale passée à optimiser la requête, en millisecondes.
- `min_plan_time` : Durée minimale passée à optimiser la requête, en millisecondes.
- `max_plan_time` : Durée maximale passée à optimiser la requête, en millisecondes.
- `mean_plan_time` : Durée moyenne passée à optimiser la requête, en millisecondes.
- `stddev_plan_time` : Déviation standard de la durée passée à optimiser la requête, en millisecondes.

Ces colonnes restent à zéro si le paramètre est désactivé.

Les statistiques sur le nombre de planifications et d'exécutions peuvent être différentes car enregistrées séparément. Si une requête échoue pendant son exécution, seules ses statistiques de planification sont mises à jour.

Exemple :

```
[local]:5433 postgres@postgres=# SELECT substr(query, 1, 40)||'...' AS query,
[local]:5433 postgres@postgres-#        plans,
[local]:5433 postgres@postgres-#        trunc(total_plan_time),
[local]:5433 postgres@postgres-#        trunc(min_plan_time),
[local]:5433 postgres@postgres-#        trunc(max_plan_time),
[local]:5433 postgres@postgres-#        trunc(mean_plan_time),
[local]:5433 postgres@postgres-#        trunc(stddev_plan_time)
[local]:5433 postgres@postgres-# FROM pg_stat_statements
[local]:5433 postgres@postgres-# ORDER BY plans desc
[local]:5433 postgres@postgres-# LIMIT 5;
```

```
               query                | plans | trunc | trunc | trunc | trunc | trunc
------------------------------------+-------+-------+-------+-------+-------+-------
 UPDATE pgbench_accounts SET ab... | 51797 |   957 |     0 |     0 |     0 |     0
 UPDATE pgbench_tellers SET tba... | 51797 |   884 |     0 |     0 |     0 |     0
 INSERT INTO pgbench_history (t... | 51797 |   374 |     0 |     0 |     0 |     0
 SELECT abalance FROM pgbench_a... | 51797 |   851 |     0 |     0 |     0 |     0
 UPDATE pgbench_branches SET bb... | 51797 |   861 |     0 |     0 |     0 |     0
(5 rows)
```

1.7.5 PG_STAT_ACTIVITY : NOUVEAU CHAMP LEADER_PID

- La vue `pg_stat_activity` contient une nouvelle colonne `leader_pid` pour identifier le *leader* d'un groupe de processus parallélisés ;
- Pour un *leader* ou un processus non parallélisé, la valeur de cette colonne est `NULL`.

Il est désormais possible de distinguer et faire un lien entre le processus *leader* d'une requête parallélisée et ses processus *workers* dans la vue `pg_stat_activity`.

La colonne `leader_pid` contient le pid du processus *leader* pour chaque ligne de `pg_stat_activity` correspondant à un processus *worker*. Elle contient `NULL` dans tous les autres cas :

```
PG13> SELECT leader_pid, pid, state, backend_type, query
        FROM pg_stat_activity
       WHERE backend_type IN ('client backend', 'parallel worker');

leader_pid |  pid   | state  |   backend_type   |           query
-----------+--------+--------+------------------+-----------------------------------
      NULL | 207706 | active | client backend   | SELECT avg(i) FROM test_nonparallelis
      NULL | 207949 | active | client backend   | SELECT avg(i) FROM test_parallelise;
      NULL | 207959 | active | client backend   | SELECT avg(i) FROM test_parallelise;
    207959 | 208561 | active | parallel worker  | SELECT avg(i) FROM test_parallelise;
    207959 | 208562 | active | parallel worker  | SELECT avg(i) FROM test_parallelise;
    207949 | 208564 | active | parallel worker  | SELECT avg(i) FROM test_parallelise;
    207949 | 208565 | active | parallel worker  | SELECT avg(i) FROM test_parallelise;
```

Dans les versions précédentes de PostgreSQL, il était difficile de rattacher un *parallel worker* à son processus *leader*.

Il y a eu plusieurs évolutions depuis la mise en place du parallélisme. En version 9.6, les processus dédiés au parallélisme étaient visibles dans la vue `pg_stat_activity`, mais la plupart des informations n'étaient pas renseignées. En version 10, les processus *parallel workers* étaient catégorisés comme `background workers` dans celle-ci. Depuis la version 11 de PostgreSQL, la colonne `backend_type` a une dénomination spécifique pour ces processus dédiés à la parallélisation des requêtes : `parallel worker`.

Pour comparaison, voici le résultat du précédent exemple exécuté sous PostgreSQL 12 :

```
PG12> SELECT pid, state, backend_type, query
      FROM pg_stat_activity
      WHERE backend_type IN ('client backend', 'parallel worker');
```

```
 pid    | state  |   backend_type   |                query
--------+--------+------------------+---------------------------------------
206327 | active | client backend   | SELECT avg(i) FROM test_parallelise;
206328 | active | client backend   | SELECT avg(i) FROM test_parallelise;
206329 | active | client backend   | SELECT avg(i) FROM test_nonparallelise;
207201 | active | parallel worker | SELECT avg(i) FROM test_parallelise;
207202 | active | parallel worker | SELECT avg(i) FROM test_parallelise;
207203 | active | parallel worker | SELECT avg(i) FROM test_parallelise;
207204 | active | parallel worker | SELECT avg(i) FROM test_parallelise;
```

1.7.6 NOUVELLE VUE PG_STAT_PROGRESS_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`.

1.7.7 NOUVELLE VUE PG_SHMEM_ALLOCATIONS

Vue pg_shmem_allocations :

- Voir les allocations du segment principal de mémoire partagée
- PostgreSQL et extensions

Cette vue liste l'attribution des blocs de la mémoire partagée statique de l'instance. Les segments de mémoire partagée dynamique n'y sont pas répertoriés.

Les principales lignes qui y figurent sont :

```
postgres=# SELECT * FROM pg_shmem_allocations ORDER BY size DESC LIMIT 5;

        name         |    off    |   size    | allocated_size
---------------------+-----------+-----------+----------------
 Buffer Blocks       |   6442752 | 134217728 |      134217728
 <anonymous>         |         ¤ |   4726784 |        4726784
 XLOG Ctl            |     53888 |   4208272 |        4208384
 ¤                   | 147197696 |   1913088 |        1913088
 Buffer Descriptors  |   5394176 |   1048576 |        1048576
```

La colonne `off` (ie. *offset*), indique l'emplacement. Les deux champs de taille ne diffèrent que par l'alignement de 128 octets imposé en mémoire partagée.

L'exemple ci-dessus provient d'une installation PostgreSQL 13 par défaut. La taille la plus importante correspond aux 128 Mo de *shared buffers*.

La mémoire qui n'est pas encore utilisée apparaît également dans le résultat de la requête. Elle correspond à la colonne dont le nom est valorisée à `NULL` (ici représenté par ¤).

L'un des intérêts est de suivre la consommation d'objets de la mémoire partagée, notamment certaines extensions.

Ci-après un exemple illustré avec l'extension `pg_stat_statements`. Voici l'état original de la mémoire partagée:

```
postgres=# SELECT pg_size_pretty(sum(allocated_size))
           FROM pg_shmem_allocations ;

 total
--------
 142 MB
(1 row)
```

Nous activons l'extension pg_stat_statements et paramétrons le nombre maximum de requête qu'il peut suivre à 100.000:

```
$ cat <<EOF >> $PGDATA/postgresql.conf
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.max = 100000
EOF

$ pg_ctl restart
```

Après le redémarrage de l'instance requis par pg_stat_statements, nous observons une augmentation de la taille totale de la mémoire partagée principale :

```
postgres=# SELECT pg_size_pretty(sum(allocated_size)) FROM pg_shmem_allocations;

 pg_size_pretty
----------------
 171 MB
```

Pour 100,000 requêtes uniques suivies, pg_stat_statements consomme donc environ 29 MB de mémoire partagée.

Inspecter les détails de cette mémoire partagée principale et, pour les plus curieux, un détour dans le code nous permet de mieux identifier la consommation:

```
postgres=# SELECT name, pg_size_pretty(allocated_size)
           FROM pg_shmem_allocations
           WHERE name ~'anon|statements'
           ORDER BY allocated_size;

          name           | pg_size_pretty
-------------------------+----------------
 pg_stat_statements      | 128 bytes
 pg_stat_statements hash | 4992 bytes
 <anonymous>             | 33 MB
```

Deux nouveaux objets apparaissent en mémoire partagée principale et nous constatons l'augmentation de l'espace anonyme.

Le premier objet détient des informations globales à l'extension, le second détient la table de hashage et les statistiques référencées par celle-ci pour chaque requête sont allouées dans l'espace anonyme. Notez que ce dernier ne concerne que les statistiques, le texte des requêtes est quant à lui écrit sur disque, permettant ainsi une taille illimitée au besoin.

1.8 PERFORMANCES

- B-Tree : déduplication
- Tri incrémental
- Statistiques étendues
- Hash Aggregate : débord sur disque
- Statistiques d'utilisation des WAL
- EXPLAIN : Utilisation disque du planificateur

1.8.1 DÉDUPLICATION DES INDEX B-TREE

1.8.1.1 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.

1.8.1.2 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);
```

1.8.1.3 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`)

1.8.2 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.

1.8.3 PARAMÉTRAGE DU DÉTAIL POUR LES STATISTIQUES ÉTENDUES

- Permet de spécifier séparément la finesse de calcul pour les statistiques classiques et étendues
- Nouvelle commande :
  `ALTER STATISTICS stat_name SET STATISTICS target_value`

PostgreSQL dispose de deux types de statistiques sur les données :

- statistiques sur les colonnes **individuellement** (fraction de valeurs nulles, valeurs distinctes, valeurs les plus fréquentes, histogrammes etc..) ;
- statistiques sur **plusieurs colonnes** (dépendance entre colonnes, valeurs les plus fréquentes et nombre de valeurs distinctes).

Lors du calcul des statistiques, il est possible de modifier le nombre de lignes échantillonnées et la finesse des statistiques calculées. Cette finesse est définie globalement par le paramètre `default_statistics_target` ou spécifiquement à chaque colonne à l'aide de l'ordre SQL `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ...`.

Jusqu'à présent, ce paramétrage était valable à la fois pour les statistiques de colonnes et pour les statistiques étendues.

Il est désormais possible de les découpler et de spécifier un paramétrage spécifique aux statistiques étendues grâce à la commande suivante :

```
ALTER STATISTICS stat_name SET STATISTICS target_value;
```

Il est possible de conserver le comportement historique avec `target_value` égal à `-1`. Une valeur de `0` permet de désactiver complètement la collecte pour les statistiques étendues seulement. Enfin, une valeur supérieure à zéro définit une valeur spécifique pour la finesse des statistiques étendues.

1.8.4 HASH AGGREGATE : DÉBORD SUR DISQUE

- les nœuds *Hash Aggregate* peuvent désormais déborder sur disque
- permet une gestion plus saine de la mémoire
- nouveau paramètre `hash_mem_multiplier`
- `hash_mem_multiplier * work_mem` définit la quantité de mémoire autorisée pour chaque nœud *Hash Aggregate* ou *Hash*
- régression de performance possible si ce paramètre n'est pas adapté

Dans les versions précédentes de PostgreSQL, il était possible de consommer plus de mémoire que la valeur spécifiée dans `work_mem` si l'estimation du nombre de ligne était incorrecte dans un nœud de type *Hash Aggregate*. Dans cette nouvelle version de PostgreSQL, l'exécuteur est capable de suivre la consommation mémoire de cette opération et peut choisir de déborder le surplus de données sur disque dès que la mémoire allouée dépasse la valeur du `work_mem`.

Dans ce mode, les lignes qui correspondent à des groupes en mémoire continuent à les alimenter. Les lignes qui devraient créer de nouveaux groupes sont écrites sur disque sous la forme de partitions. Les partitions sont dimensionnées pour pouvoir être traitées en mémoire.

Lors de l'exécution, les partitions sont traitées en plusieurs passes appelée *batch*. Il est possible que le nombre de passe soit différent du nombre de partitions planifiée en cas d'erreur d'estimation. Il est également possible que le traitement des partitions donne lieu à de nouveaux débordements sur disque si la taille de la partition a été sous-estimée.

Voici un exemple de nœuds *Hash Aggregate* affiché par `EXPLAIN`.

```
HashAggregate  (actual time=773.405..889.020 rows=99999 loops=1)
  Group Key: tablea.ac1
  Planned Partitions: 32  Batches: 33  Memory Usage: 4369kB  Disk Usage: 30456kB
```

La commande permet désormais de connaître :

- le nombre de partitions estimées par le planificateur (si pertinent) ;
- le nombre de passes (*batch*) effectivement réalisées par l'exécuteur ;
- la volumétrie occupée en mémoire ;
- la volumétrie occupée sur disque (si pertinent).

Ce nouveau comportement est plus résiliant, mais peut provoquer en contrepartie des régressions de performance sur certaines requêtes qui bénéficieraient de plus de mémoire dans les versions précédentes. Un nouveau paramètre a été créé afin d'ajuster la quantité de mémoire autorisée pour la construction de table de hachage : `hash_mem_multiplier`.

La quantité de mémoire disponible pour les *Hash Aggregate* et *Hash* (et donc *Hash Join*) est désormais calculée en multipliant ce paramètre par `work_mem`.

Il faut prendre en compte le nombre maximal de connexions simultanées possible sur le serveur lorsque l'on dimensionne cette zone mémoire. En effet, comme pour `work_mem`, cette quantité de mémoire risque d'être allouée plusieurs fois simultanément.

1.8.5 STATISTIQUES D'UTILISATION DES WAL

1.8.5.1 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.

1.8.5.2 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

1.8.5.3 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
```

1.8.5.4 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

1.8.5.5 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

1.8.6 EXPLAIN : UTILISATION DISQUE DU PLANIFICATEUR

- `EXPLAIN` avec l'option `BUFFERS` affiche désormais l'utilisation des buffers lors de la phase de planification ;
- L'option `BUFFERS` ne requiert plus l'utilisation de `ANALYZE` pour être utilisée.

La commande `EXPLAIN` peut désormais afficher l'utilisation des buffers pendant la phase de planification.

L'information apparaît avec l'activation de l'option `BUFFERS`.

```
postgres@bench=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM pgbench_accounts ;

                    QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..80029.47 rows=3032147 width=97)
                               (actual time=0.056..1250.572 rows=3000000 loops=1)
   Buffers: shared hit=2253 read=47455 dirtied=32194 written=16228
 Planning:
   Buffers: shared hit=32 read=1 dirtied=1
 Planning Time: 0.624 ms
 Execution Time: 1357.520 ms
(6 rows)
```

L'option `BUFFERS` peut désormais être utilisée sans `ANALYZE`. Il n'affiche alors que le volume de buffers manipulé lors de la phase de planification :

```
postgres@bench=# EXPLAIN (BUFFERS) SELECT * FROM pgbench_accounts ;
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..80029.47 rows=3032147 width=97)
 Planning:
   Buffers: shared hit=47 read=8
(3 rows)
```

1.9 RÉGRESSIONS

- Réplication : `wal_keep_segments` devient `wal_keep_size`
- `effective_io_concurrency` : changement d'échelle

1.9.1 WAL_KEEP_SEGMENTS DEVIENT WAL_KEEP_SIZE

- `wal_keep_segments` devient `wal_keep_size`
- La quantité de WAL à conserver est maintenant spécifiée en taille et plus en nombre de fichiers

Avant PostgreSQL 13, le paramètre `wal_keep_segments` avait pour fonction de spécifier le nombre de WAL à conserver pour les instances secondaires. Ce mécanisme est intéressant pour permettre à une instance secondaire de se raccrocher à l'instance primaire suite à une déconnexion. Il permet également de garantir que si cet état dure longtemps, le système de fichiers qui contient les journaux de transactions ne se remplira pas, contrairement au mécanisme de slot de réplication.

Concernant les slots, le nouveau paramètre `max_slot_wal_keep_size`, a été créé pour appliquer une limite similaire aux WAL conservés par des slots de réplication. Ce paramètre permet de spécifier la valeur de cette limite en mégaoctets. Les paramètres `min_wal_size` et `max_wal_size` spécifient également les limites avant déclenchement d'un checkpoint en mégaoctets.

Afin d'harmoniser les noms et unités des paramètres GUC qui permettent de spécifier des quantités de WAL, il a été décidé de :

- renommer `wal_keep_segments` en `wal_keep_size` ;
- utiliser l'unité `MB` pour spécifier la quantité de WAL à conserver grâce à ce paramètre.

Par exemple, sachant qu'un journal vaut généralement 16 Mo, on remplacera :

```
wal_keep_segments = 100
```

par :

```
wal_keep_size = 1600MB
```

Toutefois, une minorité des installations sera concernée par ce changement, car la valeur par défaut est 0. De nos jours, une réplication est plutôt sécurisée par *log shipping* ou par slot de réplication.

1.9.2 EFFECTIVE_IO_CONCURRENCY : CHANGEMENT D'ÉCHELLE

Paramètre `effective_io_concurrency` :

- Nombre d'I/O en parallèle pour une session
- L'échelle change :
  - multiplier les valeurs par 1 à 5

`effective_io_concurrency` sert à contrôler le nombre d'accès simultanés qu'un **client** peut demander aux disques. En pratique, il ne sert qu'à l'optimiseur pour juger de l'intérêt d'un accès par *Bitmap Heap Scan*.

Pour des raisons de cohérence, suite à l'introduction du paramètre `maintenance_io_concurrency`, la valeur de ce paramètre est modifiée.

La valeur reste entre 0 et 1000. Le défaut était et reste à 1. Pour des disques mécaniques, compter le nombre de disques dans une grappe RAID (hors parité). Pour des SSD, on peut monter à plusieurs centaines. Le paramètre est au niveau d'un client : on le baissera s'il y a beaucoup de requêtes simultanées.

Si une valeur a été calculée, on peut convertir avec la formule suivante :

```
SELECT  round(sum(ANCIENNE_VALEUR / n::float))
FROM    generate_series(1, ANCIENNE_VALEUR) s(n);
```

Ce qui donne les conversions suivantes :

Ancienne valeur	Nouvelle valeur
1	1
2	3
3	6
10	29
100	519

2 ATELIERS

2.1 TP - PARTITIONNEMENT, DÉCLENCHEUR `BEFORE`

- Création / suppression d'un déclencheur `BEFORE` sur une table partitionnée ;
- Cas d'un déclencheur `BEFORE` qui modifie la partition cible.

2.1.1 CRÉATION DE LA TABLE PARTITIONNÉE

Créer d'une table partitionnée :

```
psql << EOF
DROP TABLE IF EXISTS log;

CREATE TABLE log (
   id serial,
   details text,
   creation_ts timestamp with time zone,
   created_by text
) PARTITION BY RANGE (creation_ts);

CREATE TABLE log_202011
   PARTITION OF log
   FOR VALUES FROM ('2020-11-01 00:00:00+01'::timestamp with time zone)
                TO ('2020-12-01 00:00:00+01'::timestamp with time zone);
CREATE TABLE log_202012
   PARTITION OF log
   FOR VALUES FROM ('2020-12-01 00:00:00+01'::timestamp with time zone)
                TO ('2021-01-01 00:00:00+01'::timestamp with time zone);
EOF
```

2.1.2 CRÉATION D'UN DÉCLENCHEUR BEFORE

Créer un déclencheur BEFORE pour mettre le nom de l'utilisateur :

```
psql << 'EOF'
CREATE OR REPLACE FUNCTION log_user() RETURNS trigger AS $log_user$
    BEGIN
        NEW.created_by := current_user;
        RETURN NEW;
    END;
$log_user$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS log_user ON log;

CREATE TRIGGER log_user BEFORE INSERT ON log
    FOR EACH ROW EXECUTE FUNCTION log_user();
EOF
```

Afficher la description de la table partitionnée et d'une partition :

```
psql << EOF
\d+ log
\d log_202011
EOF
```

On obtient :

```
                          Partitioned table "public.log"
...
Triggers:
   log_user BEFORE INSERT ON log FOR EACH ROW EXECUTE FUNCTION log_user()
Partitions:
   log_202011 FOR VALUES FROM ('2020-11-01 00:00:00+01')
                           TO ('2020-12-01 00:00:00+01'),
   log_202012 FOR VALUES FROM ('2020-12-01 00:00:00+01')
                           TO ('2021-01-01 00:00:00+01')

                             Table "public.log_202011"
...
Partition of: log
   log FOR VALUES FROM ('2020-11-01 00:00:00+01')
                    TO ('2020-12-01 00:00:00+01')
```

```
Triggers:
   log_user BEFORE INSERT ON log_202011 FOR EACH ROW EXECUTE FUNCTION log_user(),
                ON TABLE log
```

Test d'insertion d'une ligne dans la partition :

```
psql << EOF
INSERT INTO log(details, creation_ts)
       VALUES ('Message', '2020-12-07 10:07:54'::timestamp);
SELECT tableoid::regclass, * FROM log;
EOF
```

On observe que le déclencheur a fonctionné comme attendu :

```
  tableoid   | id | details |      creation_ts       | created_by
-------------+----+---------+------------------------+------------
 log_202012 |  1 | Message | 2020-12-07 10:07:54+01 | postgres
(1 row)
```

2.1.3 DÉCLENCHEUR BEFORE QUI MODIFIE LA PARTITION CIBLE

Créer un déclencheur BEFORE qui modifie l'horodatage de création :

```
psql << 'EOF'
CREATE OR REPLACE FUNCTION log_antidate_stamp() RETURNS trigger AS $log_stamp$
    BEGIN
        NEW.creation_ts := NEW.creation_ts - INTERVAL '1 MONTH';
        RETURN NEW;
    END;
$log_stamp$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS log_antidate_stamp ON log;

CREATE TRIGGER log_antidate_stamp BEFORE INSERT ON log
    FOR EACH ROW EXECUTE FUNCTION log_antidate_stamp();
EOF
```

Test d'insertion d'une ligne dans la partition :

```
psql -v ON_ERROR_STOP=1 << EOF
INSERT INTO log(details, creation_ts)
       VALUES ('Message', '2020-12-07 10:07:54'::timestamp with time zone);
SELECT * FROM log;
```

```
EOF
```

Un message nous informe qu'il est impossible de modifier la partition cible d'une ligne avec un déclencheur `BEFORE`.

```
ERROR:  moving row to another partition during a BEFORE FOR EACH ROW trigger
+++ is not supported
DETAIL:  Before executing trigger "log_antidate_stamp", the row was to be in
+++ partition "public.log_202012".
```

2.1.4 SUPPRIMER LE DÉCLENCHEUR

Supprimer le déclencheur `log_antidate_stamp` :

```
psql << EOF
DROP TRIGGER log_antidate_stamp ON log;
DROP FUNCTION log_antidate_stamp;
EOF
```

Afficher les informations détaillées sur la table partitionnée `log` et sa partition `log_202011`. Confirmer que le déclencheur a été supprimé de la partition :

```
                          Partitioned table "public.log"
...
Triggers:
   log_user BEFORE INSERT ON log FOR EACH ROW EXECUTE FUNCTION log_user()
Partitions:
   log_202011 FOR VALUES FROM ('2020-11-01 00:00:00+01')
                           TO ('2020-12-01 00:00:00+01'),
   log_202012 FOR VALUES FROM ('2020-12-01 00:00:00+01')
                           TO ('2021-01-01 00:00:00+01')

                            Table "public.log_202011"
...
Partition of:
   log FOR VALUES FROM ('2020-11-01 00:00:00+01')
                    TO ('2020-12-01 00:00:00+01')
Triggers:
   log_user BEFORE INSERT ON log_202011 FOR EACH ROW EXECUTE FUNCTION log_user(),
               ON TABLE log
```

2.2 TP - FICHIERS MANIFESTE ET VÉRIFICATION DES SAUVEGARDES

- Création d'une sauvegarde ;
- Vérification du fichier manifeste ;
- Test de corruptions.

2.2.1 CRÉATION DE LA SAUVEGARDE

Réaliser une sauvegarde au format `plain` et observer le résultat :

```
$ export BKP_DIR=/bkp
$ pg_basebackup --format=p --pgdata=$BKP_DIR/bkp_plain
$ ls -al $BPK_DIR/bkp_plain
total 260
drwx------. 19 postgres postgres   4096 May 13 11:20 .
drwxrwxr-x.  5 postgres postgres   4096 May 13 11:20 ..
-rw-------.  1 postgres postgres    225 May 13 11:20 backup_label
-rw-------.  1 postgres postgres 135117 May 13 11:20 backup_manifest
drwx------.  5 postgres postgres   4096 May 13 11:20 base
drwx------.  2 postgres postgres   4096 May 13 11:20 global
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_commit_ts
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_dynshmem
-rw-------.  1 postgres postgres   4513 May 13 11:20 pg_hba.conf
-rw-------.  1 postgres postgres   1636 May 13 11:20 pg_ident.conf
drwx------.  4 postgres postgres   4096 May 13 11:20 pg_logical
drwx------.  4 postgres postgres   4096 May 13 11:20 pg_multixact
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_notify
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_replslot
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_serial
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_snapshots
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_stat
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_stat_tmp
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_subtrans
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_tblspc
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_twophase
-rw-------.  1 postgres postgres      3 May 13 11:20 PG_VERSION
drwx------.  3 postgres postgres   4096 May 13 11:20 pg_wal
drwx------.  2 postgres postgres   4096 May 13 11:20 pg_xact
```

```
-rw-------.  1 postgres postgres     88 May 13 11:20 postgresql.auto.conf
-rw-------.  1 postgres postgres  27902 May 13 11:20 postgresql.conf
```

Réaliser une sauvegarde au format `tar` compressé et observer le résultat :

```
$ export BKP_DIR=/bkp
$ pg_basebackup --format=t --gzip --pgdata= $BKP_DIR/bkp_compresse
$ ls -al $BKP_DIR/bkp_compresse
total 3132
drwx------. 2 postgres postgres    4096 May 13 11:15 .
drwxrwxr-x. 3 postgres postgres    4096 May 13 11:15 ..
-rw-------. 1 postgres postgres  135258 May 13 11:15 backup_manifest
-rw-------. 1 postgres postgres 3037500 May 13 11:15 base.tar.gz
-rw-------. 1 postgres postgres   17073 May 13 11:15 pg_wal.tar.gz
```

2.2.2 FICHIER MANIFESTE

Lister les fichiers présents dans le manifeste :

```
$cat $BKP_DIR/bkp_plain/backup_manifest | jq '.Files[] .Path'
```

Lister les informations sur les journaux de transactions nécessaire à la restauration :

```
$ cat $BKP_DIR/bkp_plain/backup_manifest | jq -r '."WAL-Ranges"[]'
{
  "Timeline": 1,
  "Start-LSN": "0/8000028",
  "End-LSN": "0/8000100"
}
```

2.2.3 VÉRIFIER LES SAUVEGARDES

2.2.3.1 Vérification d'une sauvegarde au format `plain`

Vérifier la sauvegarde faite au format `plain` :

```
$ pg_verifybackup $BKP_DIR/bkp_plain
backup successfully verified
```

2.2.3.2 Ajout de fichier à la sauvegarde

Ajouter des fichiers a la sauvegarde :

```
$ mkdir $BKP_DIR/bkp_plain/conf
$ touch $BKP_DIR/bkp_plain/conf/postgresql.conf
```

Vérifier la sauvegarde :

```
$ pg_verifybackup $BKP_DIR/bkp_plain
pg_verifybackup: error: "conf/postgresql.conf" is present on disk but not in \
  the manifest
```

Ajouter des fichiers peut être utile si vous souhaitez sauvegarder votre configuration avec les données sur des installations de PostgreSQL du type DEBIAN/UBUNTU.

Refaire la vérification en ignorant le répertoire de configuration :

```
$ pg_verifybackup --ignore=conf /$BKP_DIR/bkp_plain
backup successfully verified
```

2.2.3.3 Corruption d'un journal de transactions

Corrompre un journal de transactions dans la sauvegarde :

```
$ cp $BKP_DIR/bkp_plain/pg_wal/000000010000000000000008 .
$ printf 'X' \
  | dd conv=notrunc of=$BKP_DIR/bkp_plain/pg_wal/000000010000000000000008 \
       bs=1 seek=10
```

Vérifier la sauvegarde :

```
$ pg_verifybackup --ignore=conf $BKP_DIR/bkp_plain
pg_waldump: fatal: could not find a valid record after 0/8000028
pg_verifybackup: error: WAL parsing failed for timeline 1
```

Effectuer la même vérification en ignorant les journaux de transactions :

```
$ pg_verifybackup --ignore=conf --no-parse-wal $BKP_DIR/bkp_plain
backup successfully verified
```

2.2.3.4 Plus de corruptions des fichiers dans la sauvegarde

Modifier le fichier PG_VERSION :

```
$ cp $BKP_DIR/bkp_plain/PG_VERSION .
$ echo "##" >> $BKP_DIR/bkp_plain/PG_VERSION
```

Retirer le fichier backup_label et le wal modifié précédemment :

```
$ mv $BKP_DIR/bkp_plain/backup_label .
$ rm $BKP_DIR/bkp_plain/pg_wal/000000010000000000000008
```

Compter les fichiers dans le répertoire pg_twophase puis déplacer le répertoire :

```
$ find $BKP_DIR/bkp_plain/pg_twophase/ -type f | wc -l
0
$ mv $BKP_DIR/bkp_plain/pg_twophase .
```

Vérifier la sauvegarde :

```
$ pg_verifybackup --ignore=conf $BKP_DIR/bkp_plain
pg_verifybackup: error: "PG_VERSION" has size 6 on disk but size 3 in the manifest
pg_verifybackup: error: "backup_label" is present in the manifest but not on disk
pg_waldump: fatal: could not find any WAL file
pg_verifybackup: error: WAL parsing failed for timeline 1
```

pg_verifybackup ne vérifie que les fichiers, c'est pour cette raison qu'il n'y a pas d'erreurs associées à la suppression du répertoire pg_twophase.

La base de données a beaucoup grandi et la sauvegarde est volumineuse. Vérifier la sauvegarde en sortant dès la première erreur :

```
$ pg_verifybackup --ignore=conf --exit-on-error $BKP_DIR/bkp_plain
pg_verifybackup: error: "PG_VERSION" has size 6 on disk but size 3 in the manifest
```

2.2.3.5 Vérifier une sauvegarde au format tar

Tenter de vérifier la sauvegarde compressée :

```
$ pg_verifybackup $BKP_DIR/bkp_compresse/
```

On peut observer que la vérification échoue, car il faut que la sauvegarde soit décompressée et décompactée pour que les fichiers soient accessibles.

2.3 TP - COLONNE LEADER_PID DANS PG_STAT_ACTIVITY

- Compter les requêtes parallélisées et le nombre processus *workers* invoqués ;
- Identifier les processus *workers* d'une requête parallélisée.

Créer une table et y insérer un nombre conséquent de lignes :

```
CREATE TABLE test_parallelise(i int, t text);

INSERT INTO test_parallelise(i, t)
       SELECT i, 't :' || i
       FROM generate_series(1,30000000) AS F(i);
```

Ouvrir plusieurs sessions `psql` et y exécuter la requête suivante :

```
SELECT count(*) FROM test_parallelise ;
```

Dans une session supplémentaire, compter le nombre de requêtes actuellement parallélisées et le nombre de processus *worker* invoqués :

```
SELECT COUNT(DISTINCT leader_pid) AS nb_requetes_parallelisees,
       COUNT(leader_pid) AS parallel_workers
  FROM pg_stat_activity
 WHERE leader_pid IS NOT NULL
   AND backend_type = 'parallel worker' \watch 1
```

Exemple de résultat :

```
 nb_requetes_parallelisees | parallel_workers
---------------------------+------------------
                         2 |                4
```

Identifier les processus *worker* d'une requête parallélisée :

```
SELECT pid,
       ARRAY(SELECT pid
             FROM pg_stat_activity
             WHERE leader_pid = psa.pid) AS workers,
       state, query
  FROM pg_stat_activity AS psa
 WHERE backend_type = 'client backend';
```

Exemple de retour :

```
  pid   |     workers     | state  | query
--------+-----------------+--------+--------------------------------------
 684097 | {684113,684114} | active | SELECT count(*) FROM test_parallelise ;
```

2.4 TP - SUIVI DE L'EXÉCUTION DES SAUVEGARDES

- Suivi d'une sauvegarde simple ;
- Suivi sans estimation de la taille.

2.4.1 MISE EN PLACE

Mettre en place les bases de données pour le test :

```
psql <<EOF
CREATE DATABASE bench1;
CREATE TABLESPACE bench2
       LOCATION 'CHEMIN_VERS_LE_TABLESPACE'
CREATE DATABASE bench2
       WITH TABLESPACE 'bench2';
EOF
```

Générer des données avec `pg_bench` :

```
pgbench --initialize --scale 100 bench1
pgbench --initialize --scale 100 bench2
```

Remarque : un facteur d'échelle de 100 génère une base de données d'approximativement 1,5 Go.

2.4.2 SAUVEGARDE SIMPLE

Dans une session, lancer une commande `watch` pour observer le contenu de la vue système `pg_stat_progress_basebackup` :

```
watch "psql  \
  --expanded \
  --command=\"SELECT * FROM pg_stat_progress_basebackup\""
```

Dans une autre session, lancer une sauvegarde :

```
pg_basebackup \
  --format=t --gzip \
  --pgdata=CHEMIN_VERS_LE_BACKUP
```

Observer le déroulement de la sauvegarde dans la vue système.

Exemple :

```
-[ RECORD 1 ]--------+-------------------------
pid                  | 321128
phase                | streaming database files
backup_total         | 3177546240
backup_streamed      | 2496398336
tablespaces_total    | 2
tablespaces_streamed | 1
```

On constate que :

- `pid` correspond bien au *walsender* qui sert le `pg_base_backup` :

  ```
  $ psql \
      --expanded \
      --command="SELECT pid, backend_type, application_name \
                 FROM pg_stat_activity \
                 WHERE pid = 321313"
  -[ RECORD 1 ]----+--------------
  pid              | 321313
  backend_type     | walsender
  application_name | pg_basebackup
  ```

- Le champ `phase` passe sucessivement par les états : `initializing`, `waiting for checkpoint to finish`, `estimating backup size`, `streaming database files`, `waiting for wal archiving to finish` et `transferring wal files`. Certaines phases sont très courtes et peuvent ne pas être observables.

- L'estimation de la taille de la base de données correspond bien à la taille de PGDATA, plus la taille du tablespace, moins celle des journaux de transactions.

  ```
  $ du -sh $PGDATA
  2.5G  $PGDATA
  $ du -sh $PGDATA/pg_tblspc/16385/
  1.5G  $PGDATA/pg_tblspc/16385/
  $ du -sh $PGDATA/pg_wal
  1.1G  $PGDATA/pg_wal
  ```

- Le compteur de tablespace est bien incrémenté.

2.4.3 SAUVEGARDE SANS ESTIMATION DE LA TAILLE

Lancer la commande suivante en utilisant un chemin différent pour la sauvegarde :

```
$ pg_basebackup \
  --format=t --gzip \
  --no-estimate-size \
  --pgdata=CHEMIN_VERS_LE_BACKUP
```

Cette fois-ci, on observe que le champ `backup_total` est vide. En effet, la phase `estimating backup size` n'a pas été exécutée. Pour des instances particulièrement grosses, cela peut permettre de diminuer le temps de sauvegarde.

2.5 TP - PROGRESSION DE LA COMMANDE ANALYZE

- mise en place des tables et données ;
- calcul des statistiques et observations.

2.5.1 MISE EN PLACE DES TABLES ET DONNÉES

Créer une table partitionnée avec 10 partitions :

```
psql << _EOF_
CREATE TABLE test(i int, j float, k float, l float, m float)
       PARTITION BY RANGE (i);
CREATE TABLE test_1 PARTITION OF test FOR VALUES FROM(0) TO(10);
CREATE TABLE test_2 PARTITION OF test FOR VALUES FROM(10) TO(20);
CREATE TABLE test_3 PARTITION OF test FOR VALUES FROM(20) TO(30);
CREATE TABLE test_4 PARTITION OF test FOR VALUES FROM(30) TO(40);
CREATE TABLE test_5 PARTITION OF test FOR VALUES FROM(40) TO(50);
CREATE TABLE test_6 PARTITION OF test FOR VALUES FROM(50) TO(60);
CREATE TABLE test_7 PARTITION OF test FOR VALUES FROM(60) TO(70);
CREATE TABLE test_8 PARTITION OF test FOR VALUES FROM(70) TO(80);
CREATE TABLE test_9 PARTITION OF test FOR VALUES FROM(80) TO(90);
CREATE TABLE test_10 PARTITION OF test FOR VALUES FROM(90) TO(100);

INSERT INTO test
      SELECT random()*99, random(), random(), random(), random()
      FROM generate_series(1, 20000000);
_EOF_
```

2.5.2 ANALYZE ET OBSERVATIONS

Dans une session psql, lancer la commande :

```
psql -x << _EOF_
SELECT pid,
       datname,
       relid::regclass,
       phase,
       (100 * sample_blks_scanned::float / sample_blks_total)::int pct_sampled,
       ext_stats_computed || '/' || ext_stats_total AS extended_stat_no,
       child_tables_done || '/' || child_tables_total AS child_table_no,
```

```
       current_child_table_relid::regclass
FROM pg_stat_progress_analyze \watch 0.1
_EOF_
```

Dans une autre session, lancer la commande analyze :

```
psql -c "ANALYZE test";
```

On observe que pendant la phase d'acquisition des échantillions les partitions de la table défilent les une après les autres afin de calculer les statistiques de la table partitionnée.

```
-[ RECORD 1 ]-------------+--------------------------------
pid                       | 23887
datname                   | postgres
relid                     | test
phase                     | acquiring inherited sample rows
pct_sampled               | 6
extended_stat_no          | 0/0
child_table_no            | 0/5
current_child_table_relid | test_1

-[ RECORD 1 ]-------------+--------------------------------
pid                       | 23887
datname                   | postgres
relid                     | test
phase                     | acquiring inherited sample rows
pct_sampled               | 19
extended_stat_no          | 0/0
child_table_no            | 3/5
current_child_table_relid | test_4
```

Ensuite, les partitions sont analysée individuellement :

```
-[ RECORD 1 ]-------------+---------------------
pid                       | 23887
datname                   | postgres
relid                     | test_2
phase                     | acquiring sample rows
pct_sampled               | 6
extended_stat_no          | 0/0
child_table_no            | 0/0
current_child_table_relid | -
```

```
-[ RECORD 1 ]-------------+----------------------
pid                       | 23887
datname                   | postgres
relid                     | test_3
phase                     | acquiring sample rows
pct_sampled               | 28
extended_stat_no          | 0/0
child_table_no            | 0/0
current_child_table_relid | -

-[ RECORD 1 ]-------------+----------------------
pid                       | 23887
datname                   | postgres
relid                     | test_5
phase                     | acquiring sample rows
pct_sampled               | 42
extended_stat_no          | 0/0
child_table_no            | 0/0
current_child_table_relid | -
```

Et les statistiques calaculées :

```
-[ RECORD 1 ]-------------+----------------------
pid                       | 11836
datname                   | postgres
relid                     | test_3
phase                     | computing statistics
pct_sampled               | 100
extended_stat_no          | 0/0
child_table_no            | 0/0
current_child_table_relid | -
```

2.6 TP - HASHAGGREGATE, DÉBORD SUR DISQUE

Dimensionnement de `hash_mem_multiplier`.

Créer deux tables, les alimenter et calculer les statistiques dessus :

```
CREATE TABLE tableA(ac1 int, ac2 int);
CREATE TABLE tableB(bc1 int, bc2 int);
INSERT INTO tableA
   SELECT x, random()*100
   FROM generate_series(1,1000000) AS F(x);
INSERT INTO tableB
   SELECT mod(x,100000), random()*100
   FROM generate_series(1,1000000) AS F(x)
   ORDER BY 1;
ANALYZE tableA, tableB;
```

Afficher la valeur des paramètres `hash_mem_multiplier` et `work_mem` :

```
=> SELECT name, setting, unit, context
   FROM pg_settings
   WHERE name IN('work_mem', 'hash_mem_multiplier');

name                 | setting | unit | context
---------------------+---------+------+---------
 hash_mem_multiplier | 1       | ¤    | user
 work_mem            | 4096    | kB   | user
(2 rows)
```

Comme le montre la colonne `context`, ces deux paramètres peuvent être modifiés dans une session.

Exécuter la requête suivante et observer les informations du nœud *HashAggregate* :

```
=> EXPLAIN (ANALYZE, SETTINGS)
   SELECT ac1, count(ac2), sum(bc2)
   FROM tableA INNER JOIN TABLEB ON ac1 = bc1
   GROUP BY Ac1;

                                    QUERY PLAN
-------------------------------------------------------------------------------------
HashAggregate (actual time=781.895..898.119 rows=99999 loops=1)
 Group Key: tablea.ac1
```

```
 Planned Partitions: 32  Batches: 33  Memory Usage: 4369kB  Disk Usage: 30456kB
 -> Hash Join (actual time=170.824..587.731 rows=999990 loops=1)
     Hash Cond: (tableb.bc1 = tablea.ac1)
     ->  Seq Scan on tableb (actual time=0.037..81.744 rows=1000000 loops=1)
     ->  Hash (actual time=169.215..169.216 rows=1000000 loops=1)
          Buckets: 131072  Batches: 16  Memory Usage: 3471kB
          ->  Seq Scan on tablea (actual time=0.041..62.197 rows=1000000 loops=1)
Planning Time: 0.244 ms
Execution Time: 905.274 ms
```

Le nœud *HashAggregate* permet de connaître les informations suivantes :

- le planificateur avait prévu de créer 32 partitions sur disque ;
- l'exécuteur a eu besoin de réaliser 33 passes (*batch*) ;
- la quantité de mémoire utilisée est 4 Mo ;
- le quantité de disque utilisé est de 30 Mo.

Modifier la configuration de `hash_mem_multiplier` à 5:

```
SET hash_mem_multiplier TO 5;
```

Exécuter à nouveau la commande `EXPLAIN`. Voici le plan obtenu :

```
HashAggregate (actual time=693.633..711.925 rows=99999 loops=1)
 Group Key: tablea.ac1
 Planned Partitions: 8  Batches: 1  Memory Usage: 15633kB
 -> Hash Join (actual time=172.467..558.101 rows=999990 loops=1)
     Hash Cond: (tableb.bc1 = tablea.ac1)
     -> Seq Scan on tableb (actual time=0.017..81.738 rows=1000000 loops=1)
     -> Hash (actual time=171.096..171.097 rows=1000000 loops=1)
         Buckets: 524288  Batches: 4  Memory Usage: 13854kB
         -> Seq Scan on tablea (actual time=0.008..59.526 rows=1000000 loops=1)
Settings: hash_mem_multiplier = '20'
Planning Time: 0.336 ms
Execution Time: 723.450 ms
```

Nous constatons que :

- concernant le nœud *HashAggregate* :
  - la quantité de mémoire utilisée est désormais de 15 Mo ;
  - le planificateur avait prévu d'écrire 8 partitions sur disque ;
  - l'exécuteur n'a eu besoin de réaliser qu'une seule passe, ce qui est mieux que prévu. Cette différence provient probablement d'une erreur d'estimation.
  - l'exécution est deux fois plus rapide : `310.388` < `153.824`

- concernant les nœuds *Hash* et *Hash Join* :
  - utilisent maintenant plus de mémoire avec 13 Mo :
  - l'algorithme utilise donc moins de batch pour traiter la table de hachage ;
  - le temps de création de la table de hachage est stable ;
  - la jointure est marginalement plus rapide.

L'augmentation de la mémoire disponible pour les tables de hachage a ici permit d'accélérer l'agrégation des données.

Modifier la configuration de `hash_mem_multiplier` à 20 :

```
SET hash_mem_multiplier TO 20;
```

Exécuter une nouvelle fois la commande `EXPLAIN` :

```
HashAggregate (actual time=603.446..624.357 rows=99999 loops=1)
 Group Key: tablea.ac1
 Batches: 1  Memory Usage: 34065kB
 -> Hash Join (actual time=216.018..467.546 rows=999990 loops=1)
     Hash Cond: (tableb.bc1 = tablea.ac1)
     -> Seq Scan on tableb (actual time=0.018..60.117 rows=1000000 loops=1)
     -> Hash (actual time=213.442..213.443 rows=1000000 loops=1)
         Buckets: 1048576  Batches: 1  Memory Usage: 47255kB
         ->  Seq Scan on tablea (actual time=0.010..63.532 rows=1000000 loops=1)
Settings: hash_mem_multiplier = '20'
Planning Time: 0.247 ms
Execution Time: 639.538 ms
```

Cette fois-ci, nous constatons que :

- concernant le nœud *HashAggregate* :
  - la quantité de mémoire utilisée est de 33 Mo, c'est moins que ce que le maximum disponible (4 Mo * 20) ;
  - le planificateur n'a pas prévu d'écrire de partition sur disque ;
  - l'exécuteur n'a eu besoin de réaliser qu'une seule passe. Ce qui est conforme aux prévisions du planificateur ;
  - le temps d'exécution est similaire au plan précédent: `156.811`
- concernant les nœuds *Hash* et *Hash Join* :
  - ils utilisent maintenant plus de mémoire avec 46 Mo :
  - l'algorithme n'utilise plus qu'un seul batch ;
  - le temps de création de la table de hachage est marginalement plus long ;
  - la jointure est plus rapide `254.103` < `387.004`.

L'augmentation de la mémoire disponible pour les tables de hachage a ici permit d'accélérer la jointure entre les tables.

Note : l'optimisation la plus efficace est de modifier la requête en groupant les données sur la colonne `bc1`. Le plan utilisé est un plan parallélisé.

```
=> EXPLAIN (ANALYZE, SETTINGS)
   SELECT bc1, count(ac2), sum(bc2)
   FROM tableA INNER JOIN TABLEB ON ac1 = bc1
   GROUP BY bc1;
                                     QUERY PLAN
-------------------------------------------------------------------------------
Finalize GroupAggregate
Group Key: tableb.bc1
-> Gather Merge
   Workers Planned: 2
   Workers Launched: 2
   -> Sort
      Sort Key: tableb.bc1
      Sort Method: quicksort  Memory: 3635kB
      Worker 0: Sort Method: quicksort Memory: 3787kB
      Worker 1: Sort Method: quicksort Memory: 3864kB
      -> Partial HashAggregate
         Group Key: tableb.bc1
         Planned Partitions: 4 Batches: 5 Memory Usage: 4145kB Disk Usage: 11712kB
         Worker 0: Batches: 5 Memory Usage: 4145kB Disk Usage: 11744kB
         Worker 1: Batches: 5 Memory Usage: 4145kB Disk Usage: 11696kB
         -> Parallel Hash Join
            Hash Cond: (tableb.bc1 = tablea.ac1)
            -> Parallel Seq Scan on tableb
            -> Parallel Hash
               Buckets: 131072 Batches: 32 Memory Usage: 3552kB
               -> Parallel Seq Scan on tablea
Planning Time: 0.300 ms
Execution Time: 841.684 ms
```

2.7 TP - STATISTIQUES D'UTILISATION DES WAL

- dans `pg_stat_statements` ;
- dans les logs applicatifs ;
- dans les plans d'exécution.

2.7.1 PRÉREQUIS

Il est nécessaire d'effectuer la configuration suivante dans PostgreSQL pour faire ce TP :

- `log_autovacuum_min_duration = 0` : ce paramètre va permettre de voir les statistiques liées à l'autovacuum dans les traces de PostgreSQL.
- `shared_preload_libraries = 'pg_stat_statements, auto_explain'` : ce paramètre va précharger les extensions `pg_stat_statements` et `auto_explain`.
- `auto_explain.log_min_duration = 0` : déclencher la trace des plans d'exécution sur toutes les requêtes.
- `auto_explain.log_analyze = on` : activer ce paramètre est un prérequis à l'utilisation du paramètre `auto_explain.log_wal`. Il signale PostgreSQL qu'il doit effectuer des `EXPLAIN ANALYZE`.
- `auto_explain.log_wal = on` : ce paramètre permet d'ajouter les statistiques d'utilisation des wals au plan écrits dans les traces.
- `auto_explain.sample_rate = .01` : ce paramètre permet de limiter le nombre de requêtes écrites dans le traces en effectuant un échantillonnage. La valeur `1` signifie toutes les requêtes.

Il faudra ensuite installer `pg_stat_statements` dans la base de données `postgres` avec la commande :

```
CREATE EXTENSION pg_stat_statements;
```

2.7.2 MISE EN PLACE DU TEST

Remettre à zéro les statistiques de `pg_stat_statements`.

```
$ psql -Atc "SELECT pg_stat_statements_reset()"
```

Créer une base de données `bench` pour l'outil `pgbench` et initialiser la base de l'outil.

```
$ psql -c "CREATE DATABASE bench;"
CREATE DATABASE

$ pgbench -I -d bench
```

Lancer l'outil `pgbench` sur une durée de 60 secondes dans la base de données `bench` :

```
$ pgbench -T60 -d bench
```

2.7.3 CONSULTATION DES STATISTIQUES

Consulter `pg_stat_statements` et repérer les commandes qui ont généré des journaux de transactions.

```
$ psql <<EOF
SELECT substring(query,1,40) AS query, wal_records, wal_fpi, wal_bytes
FROM pg_stat_statements
ORDER BY wal_records DESC
LIMIT 10;
EOF
                  query                   | wal_records | wal_fpi | wal_bytes
------------------------------------------+-------------+---------+-----------
 UPDATE pgbench_accounts SET abalance = a |      103866 |       0 |   6979917
 UPDATE pgbench_tellers SET tbalance = tb |       51261 |       0 |   3798136
 UPDATE pgbench_branches SET bbalance = b |       51184 |       0 |   3781693
 INSERT INTO pgbench_history (tid, bid, a |       50934 |       0 |   4023786
 copy pgbench_accounts from stdin         |        1738 |       0 |  10691074
 vacuum analyze pgbench_accounts          |        1654 |       4 |    131678
 EXPLAIN (ANALYZE, WAL) INSERT INTO test  |        1000 |       0 |     65893
 EXPLAIN (ANALYZE, WAL, COSTS OFF) INSERT |        1000 |       0 |     65893
 alter table pgbench_accounts add primary |         303 |     276 |   2038141
 CREATE TABLE test (i int, t text)        |         115 |      28 |    128163
(10 rows)
```

Éditer le fichier de trace :

```
view $PGDATA/$(psql -tAc "SELECT pg_current_logfile()")
```

Observer l'activité de l'autovacuum, noter l'ajout des statistiques d'accès aux journaux de transactions :

```
LOG:  automatic vacuum of table "bench.public.pgbench_branches": index scans: 0
  pages: 0 removed, 1 remain, 0 skipped due to pins, 0 skipped frozen
  tuples: 150 removed, 1 remain, 0 are dead but not yet removable, oldest xmin: ...
  buffer usage: 49 hits, 0 misses, 0 dirtied
  avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
  system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
  WAL usage: 3 records, 0 full page images, 646 bytes
```

Observer les plans écrits dans les traces, noter l'ajout des statistiques d'accès aux journaux de transactions :

```
LOG:  duration: 0.313 ms  plan:
  Query Text:
     UPDATE pgbench_accounts SET abalance = abalance + 4802 WHERE aid = 8849318;
  Update on pgbench_accounts  (...) (...)
     WAL: records=3 fpi=2 bytes=15759
       ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts  (...) (...)
             Index Cond: (aid = 8849318)
```

2.8 TP - RÉPLICATION LOGIQUE ET PARTITIONNEMENT

- Ajout d'une table partitionnée à une publication ;
- Réplication vers une table partitionnée de même schéma ;
- Réplication vers une table partitionnée de schéma différent.

2.8.1 PRÉREQUIS

Il faut deux instances PostgreSQL avec le paramètre `wal_level = 'logical'` pour faire ce tp. On désignera ces deux instances comme **source** et **cible** dans la suite.

2.8.2 AJOUT D'UNE TABLE PARTITIONNÉE À UNE PUBLICATION

Créer une base de données, ajouter une table partitionnée et créer une publication pour cette table :

```
CREATE DATABASE rl;
\c rl

CREATE TABLE livres (
    id int,
    titre text,
    date_sortie timestamp with time zone)
PARTITION BY RANGE (date_sortie);

CREATE TABLE livres_2020_05
PARTITION OF livres FOR VALUES
    FROM ('20200501'::timestamp with time zone)
    TO   ('20200601'::timestamp with time zone);

CREATE PUBLICATION thepub
    FOR TABLE livres;
```

Créer une base de données sur une seconde instance, créer les mêmes tables et créer une souscription avec une chaîne de connexion adaptée (ici 5433 est le port de l'instance source):

```
CREATE DATABASE rl;
\c rl

CREATE TABLE livres (
```

```
    id int,
    titre text,
    date_sortie timestamp with time zone)
PARTITION BY RANGE (date_sortie);

CREATE TABLE livres_2020_05
PARTITION OF livres FOR VALUES
    FROM ('20200501'::timestamp with time zone)
    TO   ('20200601'::timestamp with time zone);


CREATE SUBSCRIPTION thesub
    CONNECTION 'host=/tmp port=5433 dbname=rl'
    PUBLICATION thepub;
```

Observer les tables dans la publication :

```
[source] postgres@rl=# SELECT oid, prpubid, prrelid::regclass
                         FROM pg_publication_rel ;

  oid  | prpubid | prrelid
-------+---------+---------
 17337 |   17336 | livres
(1 row)

[source] postgres@rl=# SELECT * FROM pg_publication_tables ;

 pubname | schemaname |   tablename
---------+------------+----------------
 thepub  | public     | livres_2020_05
(1 row)
```

Observer les tables dans la souscription :

```
[cible] postgres@rl=# SELECT srsubid, srrelid::regclass AS tablename,
                             srsubstate, srsublsn
                        FROM pg_subscription_rel;

 srsubid |   tablename    | srsubstate | srsublsn
---------+----------------+------------+-----------
   17402 | livres_2020_05 | r          | 0/7254B28
```

```
(1 row)
```

Ajoutez une partition sur les deux instances :

```
CREATE TABLE livres_2020_06
PARTITION OF livres FOR VALUES
    FROM ('20200601'::timestamp with time zone)
    TO   ('20200701'::timestamp with time zone);
```

Observer à nouveau les tables présentes dans la publication et la souscription.

```
[source] postgres@rl=# SELECT oid, prpubid, prrelid::regclass
                         FROM pg_publication_rel ;

  oid  | prpubid | prrelid
-------+---------+---------
 17337 |   17336 | livres
(1 row)

[source] postgres@rl=# SELECT * FROM pg_publication_tables ;
 pubname | schemaname |   tablename
---------+------------+----------------
 thepub  | public     | livres_2020_05
 thepub  | public     | livres_2020_06
(2 rows)

---

[cible] postgres@rl=# SELECT srsubid, srrelid::regclass AS tablename,
                             srsubstate, srsublsn
                        FROM pg_subscription_rel;


 srsubid |   tablename    | srsubstate | srsublsn
---------+----------------+------------+-----------
   17402 | livres_2020_05 | r          | 0/7254B28
```

On constate que la nouvelle partition est ajoutée automatiquement à la publication mais qu'elle n'est pas présente dans la souscription. En effet, il faut lancer la commande suivante pour rafraîchir la liste des tables dans la souscription à partir de la publication associée :

```
ALTER SUBSCRIPTION thesub REFRESH PUBLICATION;
```

On peut alors voir les deux partitions dans la souscription :

```
[cible] postgres@rl=# SELECT srsubid, srrelid::regclass AS tablename,
                            srsubstate, srsublsn
                        FROM pg_subscription_rel;

 srsubid |   tablename    | srsubstate | srsublsn
---------+----------------+------------+-----------
   17402 | livres_2020_05 | r          | 0/7254B28
   17402 | livres_2020_06 | r          | 0/72709A0
(2 rows)
```

Supprimer la partition `livres_2020_06`, rafraîchir la souscription et observer le résultat :

```
[source] postgres@rl=# DROP TABLE livres_2020_06;
DROP TABLE
[local]:5433 postgres@rl=# SELECT oid, prpubid, prrelid::regclass
                            FROM pg_publication_rel ;

  oid  | prpubid | prrelid
-------+---------+---------
 17337 |   17336 | livres
(1 row)

[source] postgres@rl=# SELECT * FROM pg_publication_tables ;
 pubname | schemaname |   tablename
---------+------------+----------------
 thepub  | public     | livres_2020_05
(1 row)

---

[cible] postgres@rl=# DROP TABLE livres_2020_06;
DROP TABLE
[cible] postgres@rl=# ALTER SUBSCRIPTION thesub REFRESH PUBLICATION;
ALTER SUBSCRIPTION
[cible] postgres@rl=# SELECT srsubid, srrelid::regclass AS tablename,
                            srsubstate, srsublsn
                        FROM pg_subscription_rel;
```

```
 srsubid |   tablename    | srsubstate | srsublsn
---------+----------------+------------+-----------
   17402 | livres_2020_05 | r          | 0/7254B28
(1 row)
```

2.8.3 RÉPLICATION VERS UNE TABLE PARTITIONNÉE AYANT LE MÊME SCHÉMA

Ajouter deux partitions à la table sur les deux instances :

```
CREATE TABLE livres_2020_06
PARTITION OF livres FOR VALUES
    FROM ('20200601'::timestamp with time zone)
    TO   ('20200701'::timestamp with time zone);

CREATE TABLE livres_2020_07
PARTITION OF livres FOR VALUES
    FROM ('20200701'::timestamp with time zone)
    TO   ('20200801'::timestamp with time zone);
```

Rafraîchir la souscription :

```
ALTER SUBSCRIPTION thesub REFRESH PUBLICATION;
```

Insérer des données dans la table sur l'instance source :

```
INSERT INTO livres (id, titre, date_sortie)
        SELECT  i,
                'Livre no ' || i,
                '20200501'::timestamp with time zone
            + INTERVAL '1 minute' * (random()  * 60 * 24 * 30 * 3 )::int
        FROM generate_series(1, 1000) AS F(i);
```

Lister les partitions alimentées sur les deux instances avec le nombre de lignes associées :

```
[source] postgres@rl=# SELECT tableoid::regclass, count(*)
                        FROM livres GROUP BY 1 ORDER BY 1;

    tableoid    | count
----------------+-------
 livres_2020_05 |   342
 livres_2020_06 |   319
```

```
 livres_2020_07 |   339
(3 rows)

---

[cible] postgres@rl=# SELECT tableoid::regclass, count(*)
                        FROM livres GROUP BY 1 ORDER BY 1;

    tableoid    | count
----------------+-------
 livres_2020_05 |   342
 livres_2020_06 |   319
 livres_2020_07 |   339
(3 rows)
```

On constate que le nombre de lignes présentes sur les deux instances est identique.

2.8.4 RÉPLICATION VERS UNE TABLE PARTITIONNÉE AYANT UN SCHÉMA DIFFÉRENT

Supprimer la souscription et la base de données sur le serveur cible :

```
\c rl
DROP SUBSCRIPTION thesub;
\c postgres
DROP DATABASE rl;
```

Modifier la publication :

```
ALTER PUBLICATION thepub
      SET (publish_via_partition_root = true);
```

Créer un schéma de partitionnement différent et recréer une souscription :

```
CREATE DATABASE rl;
\c rl

CREATE TABLE livres (
    id int,
    titre text,
    date_sortie timestamp with time zone)
PARTITION BY RANGE (id);
```

```
CREATE TABLE livres_500
PARTITION OF livres FOR VALUES
    FROM (1)
    TO   (500);

CREATE TABLE livres_1000
PARTITION OF livres FOR VALUES
    FROM (500)
    TO   (1000);

CREATE TABLE livres_1500
PARTITION OF livres FOR VALUES
    FROM (1000)
    TO   (1500);

CREATE SUBSCRIPTION thesub
    CONNECTION 'host=/tmp port=5433 dbname=rl'
    PUBLICATION thepub;
```

Lister les tables dans la publication et la souscription, ainsi que la liste des partitions alimentées et le nombre de lignes associées.

```
[source] postgres@rl=# SELECT oid, prpubid, prrelid::regclass
                        FROM pg_publication_rel ;

  oid  | prpubid | prrelid
-------+---------+---------
 17337 |   17336 | livres
(1 row)

[source] postgres@rl=# SELECT * FROM pg_publication_tables ;
 pubname | schemaname | tablename
---------+------------+-----------
 thepub  | public     | livres
(1 row)

[source] postgres@rl=# SELECT tableoid::regclass, count(*)
                        FROM livres GROUP BY 1 ORDER BY 1;
```

```
    tableoid    | count
----------------+-------
 livres_2020_05 |   342
 livres_2020_06 |   319
 livres_2020_07 |   339
(3 rows)
```

```
[cible] postgres@rl=# SELECT srsubid, srrelid::regclass AS tablename,
                             srsubstate, srsublsn
                      FROM pg_subscription_rel;
 srsubid | tablename | srsubstate | srsublsn
---------+-----------+------------+-----------
   17452 | livres    | r          | 0/72F6EE8
(1 row)

[cible] postgres@rl=# SELECT tableoid::regclass, count(*)
                       FROM livres GROUP BY 1 ORDER BY 1;

  tableoid   | count
-------------+-------
 livres_500  |   499
 livres_1000 |   500
 livres_1500 |     1
(3 rows)
```

On constate que :

- la table partitionnée est désormais la seule présente dans la liste des tables de la publication et de la souscription. C'est l'effet de `publish_via_partition_root` ;
- les lignes sont correctement réparties dans les partitions de la souscription qui a pourtant un schéma de partitionnement différent.

2.9 TP - CHANGEMENT À CHAUD DES INFORMATIONS DE RÉPLICATION

- Mise en place d'une réplication ;
- Changement de mot de passe ;
- Utilisateur dédié et mot de passe dans `.pgpass` sans redémarrage.

2.9.1 MISE EN PLACE DU PRIMAIRE

Sur une machine CentOS 7 ou 8 où les binaires de PostgreSQL sont installés, créer une instance primaire, si elle n'existe pas déjà, qui écoute sur le port 5432, et une instance secondaire accessible sur le port 5433.

En tant que root :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-13/bin/postgresql-13-setup initdb
# systemctl start postgresql-13
# sudo -iu postgres psql -c "ALTER ROLE postgres PASSWORD 'elephant' ;"
```

Vérifier que la réplication est bien autorisée sur localhost, avec un mot de passe :

```
# tail -3 ~postgres/13/data/pg_hba.conf
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            scram-sha-256
host    replication     all             ::1/128                 scram-sha-256
```

Si ce n'est pas le cas, ou si le serveur secondaire est sur une autre machine, adapter `pg_hba.conf`.

Ajouter un peu de volumétrie :

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench -i -s30 --no-vacuum
```

2.9.2 MISE EN PLACE DU SECONDAIRE

Répertoire et paramétrage systemd :

```
# install -o postgres -g postgres -m 0700 -d /var/lib/pgsql/13/secondaire

# cp /lib/systemd/system/postgresql-13.service \
    /etc/systemd/system/postgresql-13-secondaire.service

# cat <<EOF | EDITOR=tee systemctl edit postgresql-13-secondaire.service
[Service]
Environment=PGDATA=/var/lib/pgsql/13/secondaire
EOF
```

Mise en place de la réplication :

```
# sudo -iu postgres pg_basebackup -h localhost -U postgres \
  -D /var/lib/pgsql/13/secondaire --checkpoint=fast        \
  --create-slot  --slot='secondaire'                       \
  --write-recovery-conf --progress --verbose
# echo "port=5433" >> /var/lib/pgsql/13/secondaire/postgresql.conf
```

L'outil `pg_basebackup` demande de saisir le mot de passe de l'utilisateur `postgres`.

Il crée le slot (`--create-slot`), crée le fichier `standby.signal` et renseigne le paramétrage de réplication dans `postgresql.auto.conf`.

```
# tail -2 ~postgres/13/secondaire/postgresql.auto.conf
primary_conninfo = 'user=postgres password=elephant channel_binding=prefer
                    host=localhost port=5432
                    sslmode=prefer sslcompression=0 ssl_min_protocol_version=TLSv1.2
                    gssencmode=prefer krbsrvname=postgres target_session_attrs=any'
primary_slot_name = 'secondaire'
```

Par ailleurs, nous constatons que le mot de passe y est renseigné en clair, ainsi que de nombreuses options de réplication par défaut.

Démarrer le secondaire et vérifier la connexion :

```
# systemctl start postgresql-13-secondaire
# sudo -iu postgres psql -p5433 -c \\d
              List of relations
 Schema |       Name       | Type  |  Owner
--------+------------------+-------+----------
 public | pgbench_accounts | table | postgres
```

```
 public | pgbench_branches | table | postgres
 public | pgbench_history  | table | postgres
 public | pgbench_tellers  | table | postgres
```

Vérifier sur le primaire que tout fonctionne :

```
# sudo -iu postgres psql -xc 'TABLE pg_stat_replication'
-[ RECORD 1 ]----+------------------------------
pid              | 23986
usesysid         | 10
usename          | postgres
application_name | walreceiver
client_addr      | ::1
client_hostname  |
client_port      | 39892
backend_start    | 2020-10-15 17:36:55.207321+00
backend_xmin     |
state            | streaming
sent_lsn         | 0/2B000148
write_lsn        | 0/2B000148
flush_lsn        | 0/2B000148
replay_lsn       | 0/2B000148
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | 2020-10-15 18:39:00.30708+00
```

Notez que sur le secondaire, le mot de passe est visible de tout superutilisateur connecté avec un simple `SHOW primary_conninfo` !

2.9.3 ACTIVITÉ SUR LE SECONDAIRE

Dans deux sessions séparées, lancer de l'activité en écriture sur le primaire, et en lecture sur le secondaire.

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench --no-vacuum --rate 10 -T3600
```

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench --no-vacuum --rate 10 -T3600 \
    -p5433 --select
```

Nous allons étudier l'impact de différentes actions sur cette dernière session.

2.9.4 MODIFICATION DES INFORMATIONS DE CONNEXION ET REDÉMARRAGE

Le mot de passe est beaucoup trop simple. Nous décidons de le modifier sur le primaire :

```
# sudo -iu postgres psql \
  -c "ALTER ROLE postgres PASSWORD 'aixohph8Pienoxaec6nohp2oh' ;"
```

La connexion du standby au primaire est déjà établie et reste en place.

Toute nouvelle connexion est néanmoins refusée au prochain redémarrage du secondaire, parfois longtemps après :

```
# systemctl restart postgresql-13-secondaire
```

En conséquence du redémarrage, le pgbench sur le secondaire est évidemment coupé :

```
pgbench: fatal: Run was aborted; the above results are incomplete.
```

Le relancer :

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench --no-vacuum --rate 10 -T3600 \
    -p5433 --select
```

Dans les journaux du secondaire ~postgres/13/secondaire/log/postgresql-*.log, l'erreur de connexion apparaît :

```
2020-10-16 07:20:14.288 UTC [25189] FATAL:  could not connect to the primary server:
      FATAL:  password authentication failed for user "postgres"
```

Corriger la chaîne de connexion qui contient le mot de passe sur le secondaire :

```
# sudo -iu postgres psql -p5433
postgres=# ALTER SYSTEM SET primary_conninfo TO
  'user=postgres password=aixohph8Pienoxaec6nohp2oh host=localhost port=5432' ;
postgres=# SELECT pg_reload_conf() ;
```

Le pgbench n'est pas interrompu et le secondaire a repris la réplication sans redémarrage :

```
2020-10-16 07:29:16.333 UTC [25583] LOG:  started streaming WAL from primary
                                          at 0/49000000 on timeline 1
```

2.9.5 UTILISATEUR DÉDIÉ ET .PGPASS

Il est plus propre et plus sûr de dédier un utilisateur à la réplication, dont le mot de passe est beaucoup moins sujet au changement. De plus, sur le secondaire, le mot de passe ne doit figurer que dans le fichier `.pgpass` lisible uniquement par l'utilisateur système **postgres**.

Sur le primaire :

```
postgres=# CREATE ROLE replicator LOGIN REPLICATION PASSWORD 'evuzahs3ien0bah2haiJ' ;
```

Suite à la configuration mise en place plus haut, il n'y a pas besoin de modifier `pg_hba.conf`.

Sur le secondaire :

```
# echo "localhost:5432:replication:replicator:evuzahs3ien0bah2haiJ" \
  > ~postgres/.pgpass
# chown postgres: ~postgres/.pgpass
# chmod 600 ~postgres/.pgpass

# cat <<EOS | sudo -iu postgres psql -p 5433

ALTER SYSTEM SET primary_conninfo
  TO 'user=replicator host=localhost port=5432';

SELECT pg_reload_conf();

EOS
```

Notez que nous utilisons la pseudo-base `replication` dans le fichier `.pgpass`.

Pendant ces dernières opérations, le pgbench sur le secondaire n'a pas été interrompu… même si les données n'étaient pas de première fraîcheur lorsque la réplication était bloquée.

2.9.6 CHANGEMENT D'INSTANCE PRINCIPALE

Nous créons dans cet exercice une nouvelle instance secondaire nommée `ter` et basculons la production dessus, sans coupure de service pour `postgresql-13-secondaire`.

Création de la nouvelle instance :

```
# install -o postgres -g postgres -m 0700 -d /var/lib/pgsql/13/ter

# cp /lib/systemd/system/postgresql-13.service \
    /etc/systemd/system/postgresql-13-ter.service

# cat <<EOF | EDITOR=tee systemctl edit postgresql-13-ter.service
[Service]
Environment=PGDATA=/var/lib/pgsql/13/ter
EOF

# sudo -iu postgres pg_basebackup -h localhost -U replicator \
  -D /var/lib/pgsql/13/ter --checkpoint=fast                 \
  --create-slot  --slot='ter'                                \
  --write-recovery-conf --progress --verbose

# echo "port=5434" >> /var/lib/pgsql/13/ter/postgresql.conf

# systemctl start postgresql-13-ter
# sudo -iu postgres psql -p5434 -c \\d
              List of relations
 Schema |       Name       | Type  |  Owner
--------+------------------+-------+----------
 public | pgbench_accounts | table | postgres
 public | pgbench_branches | table | postgres
 public | pgbench_history  | table | postgres
 public | pgbench_tellers  | table | postgres
```

Vérifier sur le primaire que tout fonctionne :

```
# sudo -iu postgres psql -Atc 'SELECT count(*) FROM pg_stat_replication'
2
```

Bascule de l'instance primaire vers `ter`. Il est nécessaire pour cela d'interrompre l'instance primaire actuelle (1), vérifier la niveau de réplication (2), puis de promouvoir `ter` (3) :

```
## (1)
# systemctl stop postgresql-13

## (2)
# /usr/pgsql-13/bin/pg_controldata ~postgres/13/data/|grep 'REDO location'
Latest checkpoint's REDO location:    0/2D416C60

# sudo -iu postgres psql -qp 5434 -c checkpoint

# /usr/pgsql-13/bin/pg_controldata ~postgres/13/ter/|grep 'REDO location'
Latest checkpoint's REDO location:    0/2D416C60

## (3)
# sudo -iu postgres psql -Atp 5434 -c "SELECT pg_promote(true)"
t
```

L'instance secondaire a perdu la connexion à l'instance primaire.

```
# sudo -iu postgres psql -p5433 -Atc "SELECT count(*) FROM pg_stat_wal_receiver"
0
```

L'étape suivante consiste à établir la connexion vers la nouvelle instance :

```
# sudo -iu postgres psql -p5434 -At \
    -c "SELECT pg_create_physical_replication_slot('secondaire')"
(secondaire,)

# cat <<EOS | sudo -iu postgres psql -p 5433

ALTER SYSTEM SET primary_conninfo
  TO 'user=replicator host=localhost port=5434';

SELECT pg_reload_conf();

EOS
```

La précédente ligne du fichier `.pgpass` est limitée au seul port 5432. Nous le modifions afin de fournir un mot de passe quelque soit le port de connexion :

```
# echo "localhost:*:replication:replicator:evuzahs3ien0bah2haiJ" \
  > ~postgres/.pgpass
```

Peu de temps après, l'instance secondaire entre en réplication avec la nouvelle instance

primaire `ter` :

```
# sudo -iu postgres psql -p5434 \
    -c "select pid, state, sent_lsn from pg_stat_replication"
 pid  |   state   |  sent_lsn
------+-----------+------------
 7639 | streaming | 0/2D416E28


# sudo -iu postgres psql -p5433 \
    -c "select pid, status, flushed_lsn from pg_stat_wal_receiver"
 pid  |  status   | flushed_lsn
------+-----------+-------------
 7638 | streaming | 0/2D416E28
```

2.10 TP - VOLUME MAXIMAL DE JOURNAUX CONSERVÉ PAR LES SLOTS

- Création d'un slot ;
- Modification de la configuration des slots ;
- Simulation d'un décrochage.

2.10.1 CRÉATION D'UN SLOT

Sur une machine CentOS 7 ou 8 où les binaires de PostgreSQL sont installés, créer une instance si elle n'existe pas déjà.

Le volume maximal de WAL produit par la mécanique transactionnelle (`max_wal_size`) doit être diminué à 200 Mo afin de limiter l'espace nécessaire au test et rendre plus fréquents les `CHECKPOINT`.

Positionner `synchronous_commit` afin d'accélérer les futures écritures.

En tant que root :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-13/bin/postgresql-13-setup initdb
# echo "max_wal_size = 200MB" >> ~postgres/13/data/postgresql.conf
# echo "synchronous_commit = off" >> ~postgres/13/data/postgresql.conf
# systemctl start postgresql-13
```

Créer une base avec un peu de volumétrie :

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench -i -s20
```

Créer maintenant un slot physique `slot_tp` conservant dès à présent les WAL générés (second argument à `true`) :

```
# sudo -iu postgres psql -qc \
    "SELECT * FROM pg_create_physical_replication_slot('slot_tp', true)"
 slot_name |    lsn
-----------+------------
 slot_tp   | 0/1903F6E0

# sudo -iu postgres psql -qc \
    "SELECT slot_name, restart_lsn, wal_status FROM pg_replication_slots"
 slot_name | restart_lsn | wal_status
-----------+-------------+------------
 slot_tp   | 0/1D014018  | reserved
```

2.10.2 RÉTENTION DES SLOTS

Limiter à 400 Mo le volume de WAL que peuvent retenir les slots de réplication :

```
# sudo -iu postgres psql << _EOF_
ALTER SYSTEM SET max_slot_wal_keep_size = '400MB';
SELECT pg_reload_conf();
_EOF_
```

Vérifier la valeur de `max_slot_wal_keep_size` :

```
# sudo -iu postgres psql -XAtc "show max_slot_wal_keep_size"
400MB
```

2.10.3 GÉNÉRER DE L'ACTIVITÉ ET OBSERVER LE NOUVEAU FONCTIONNEMENT

Lancer en tâche de fond `pgbench` pour générer de l'activité en écriture et observer l'évolution du slot de réplication :

```
# sudo -iu postgres /usr/pgsql-13/bin/pgbench -T 300 &

# while sudo -iu postgres psql -AXtc "
  WITH s AS(
    SELECT count(*)-1 AS wals
    FROM pg_ls_dir('pg_wal')
  )
  SELECT
    slot_name, wal_status,
    pg_size_pretty(safe_wal_size)       AS safe_wal_size,
    pg_size_pretty(16*1024*1024*s.wals) AS wal_size
  FROM pg_replication_slots, s"
do
    sleep 1
done
slot_tp|reserved|287 MB|192 MB
slot_tp|reserved|274 MB|192 MB
slot_tp|reserved|263 MB|192 MB
slot_tp|reserved|252 MB|192 MB
slot_tp|reserved|241 MB|192 MB
slot_tp|reserved|233 MB|192 MB
slot_tp|reserved|225 MB|192 MB
slot_tp|reserved|217 MB|208 MB
```

```
slot_tp|reserved|208 MB|208 MB
slot_tp|reserved|201 MB|224 MB
slot_tp|reserved|194 MB|224 MB
slot_tp|extended|187 MB|240 MB
slot_tp|extended|181 MB|240 MB
slot_tp|extended|174 MB|256 MB
slot_tp|extended|168 MB|256 MB
slot_tp|extended|162 MB|256 MB
slot_tp|extended|154 MB|272 MB
slot_tp|extended|144 MB|288 MB
slot_tp|extended|134 MB|288 MB
slot_tp|extended|124 MB|304 MB
slot_tp|extended|115 MB|304 MB
slot_tp|extended|107 MB|320 MB
slot_tp|extended|98 MB|320 MB
slot_tp|extended|90 MB|336 MB
slot_tp|extended|82 MB|336 MB
slot_tp|extended|74 MB|352 MB
slot_tp|extended|67 MB|352 MB
slot_tp|extended|60 MB|368 MB
slot_tp|extended|53 MB|368 MB
slot_tp|extended|47 MB|384 MB
slot_tp|extended|41 MB|384 MB
slot_tp|extended|35 MB|384 MB
slot_tp|extended|29 MB|400 MB
slot_tp|extended|20 MB|400 MB
slot_tp|extended|10 MB|416 MB
slot_tp|extended|1416 kB|416 MB
slot_tp|unreserved|-7352 kB|432 MB
slot_tp|unreserved|-15 MB|432 MB
slot_tp|unreserved|-23 MB|448 MB
slot_tp|unreserved|-30 MB|448 MB
slot_tp|lost||464 MB
slot_tp|lost||464 MB
slot_tp|lost||464 MB
slot_tp|lost||464 MB
^C^C^C
```

Nous observons que la colonne `wal_status` prend la valeur `extended` lorsque la volumétrie conservée dépasse `max_wal_size` et que le `checkpoint` en cours se termine.

Ce dernier ne peut détruire les WAL en trop et les laisse donc à la charge des slots.

Peu de temps après que le volume de WAL dépasse la rétention maximale imposée, le statut passe ensuite brièvement à `unreserved`, là aussi le temps que le checkpoint en cours se termine, avant de passer définitivement au statut `lost`. La colonne `safe_wal_size` quant à elle devient négative lorsque le statut devient `unreserved`, puis devient nulle lorsque le slot est perdu.

Nous constatons aussi que la volumétrie des WAL ne redescend pas une fois le slot perdu. Même perdu, ce dernier conserve une fenêtre glissante de WAL. Pour récupérer l'espace disque et nettoyer les WAL de trop, il nous faut le supprimer et forcer un checkpoint pour éviter d'attendre le suivant :

```
# sudo -iu postgres psql << _EOF_
SELECT pg_drop_replication_slot('slot_tp');
CHECKPOINT;
_EOF_
```

La volumétrie dans `pg_wal` diminue immédiatement :

```
# du -Sh ~postgres/13/data/pg_wal
193M    /var/lib/pgsql/13/data/pg_wal
```

2.11 TP - OUTIL PG_REWIND

- Création d'une instance primaire ;
- Mettre en place la réplication ;
- Simulation d'un failover ;
- Utilisation de pg_rewind.

2.11.1 CRÉATION D'UNE INSTANCE PRIMAIRE

Nous créons pour cette démonstration des instances temporaires dans le répertoire `/tmp/rewind` :

```
export DATADIRS=/tmp/rewind
mkdir -p /tmp/rewind/archives
```

Créer une instance primaire en activant les checkpoints :

```
initdb --data-checksums --data-checksums $DATADIRS/pgrw_srv1 -U postgres
```

Note: Pour utiliser pg_rewind, il est nécessaire d'activer le paramètre `wal_log_hints` dans le `postgresql.conf` ou les sommes de contrôles au niveau de l'instance.

Configurer PostgreSQL :

```
cat <<_EOF_ >> $DATADIRS/pgrw_srv1/postgresql.conf
port = 5636
listen_addresses = '*'
logging_collector = on
archive_mode = on
archive_command = '/usr/bin/rsync -a %p $DATADIRS/archives/%f'
restore_command = '/usr/bin/rsync -a $DATADIRS/archives/%f %p'
_EOF_
```

Démarrer l'instance et y créer une base de données :

```
pg_ctl start -D $DATADIRS/pgrw_srv1 -w
psql -p 5636 -c "CREATE DATABASE bench;"
```

2.11.2 METTRE EN PLACE LA RÉPLICATION

Créer un utilisateur pour la réplication et ajouter le mot de passe au fichier `.pgpass` :

```
psql -p 5636 << _EOF_
CREATE ROLE replication
  WITH LOGIN REPLICATION PASSWORD 'replication';
_EOF_

cat << _EOF_ >> ~/.pgpass
*:5636:replication:replication:replication
*:5637:replication:replication:replication
_EOF_
chmod 600 ~/.pgpass
```

Créer une instance secondaire :

```
pg_basebackup -D $DATADIRS/pgrw_srv2 -p 5636 --progress --username=replication
```

Modifier la configuration :

```
touch $DATADIRS/pgrw_srv2/standby.signal

cat << _EOF_ >> $DATADIRS/pgrw_srv2/postgresql.conf
port = 5637
primary_conninfo =  'port=5636 user=replication application_name=replication'
_EOF_
```

Démarrer l'instance secondaire :

```
pg_ctl start -D $DATADIRS/pgrw_srv2 -w
```

La requête suivante doit renvoyer une ligne sur l'instance primaire :

```
psql -p 5636 -xc "SELECT * FROM pg_stat_replication;"
```

2.11.3 SIMULATION D'UN FAILOVER

Promouvoir l'instance secondaire :

```
pg_ctl promote -D $DATADIRS/pgrw_srv2 -w
psql -p 5637 -c CHECKPOINT
```

Ajouter des données aux deux instances afin de les faire diverger :

```
pgbench -p 5636 -i -s 20 bench;
pgbench -p 5637 -i -s 20 bench;
```

Les deux instances ont maintenant divergé. Sans action supplémentaire, il n'est donc pas possible de raccrocher l'ancienne primaire à la nouvelle.

Stopper l'ancienne instance primaire de manière brutale pour simuler une panne :

```
pg_ctl stop -D $DATADIRS/pgrw_srv1 -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`.

2.11.4 UTILISATION DE PG_REWIND

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

```
psql -p 5637 <<_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'ancienne instance primaire. En effet, les fichiers de configuration présents dans `PGDATA` seront écrasés par l'outil :

```
cp $DATADIRS/pgrw_srv1/postgresql.conf /tmp
```

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

```
$ pg_rewind --target-pgdata $DATADIRS/pgrw_srv1                          \
            --source-server "port=5637 user=replication dbname=postgres" \
            --write-recovery-conf --no-ensure-shutdown                   \
            --progress --dry-run


pg_rewind: connected to server
pg_rewind: fatal: target server must be shut down cleanly
```

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

```
$ pg_rewind --target-pgdata $DATADIRS/pgrw_srv1                          \
            --source-server "port=5637 user=replication dbname=postgres" \
            --write-recovery-conf --progress
pg_rewind: connected to server
pg_rewind: executing
+++"XXX/bin/postgres" for target server to complete crash recovery
LOG:  database system was interrupted; last known up at 2020-09-01 17:18:26 CEST
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/4000028
LOG:  invalid record length at 0/CB20E30: wanted 24, got 0
LOG:  redo done at 0/CB20D08


PostgreSQL stand-alone backend 13.0
backend> pg_rewind: servers diverged at WAL location 0/3000000 on timeline 1
pg_rewind: error: could not open file
+++ "XXX/pgrw_srv1/pg_wal/000000010000000000000005": No such file or directory
pg_rewind: fatal: could not find previous WAL record at 0/5000060
```

On constate que :

- pg_rewind a redémarré le cluster en mode mono-utilisateur afin de réaliser une récupération de l'instance ;
- l'opération échoue car PostgreSQL n'arrive pas à trouver un WAL dans PGDATA/pg_wal.

Vérifier la configuration de la restore_command dans le fichier de configuration de la

cible :

```
$ postgres -D /tmp/rewind/pgrw_srv1/ -C restore_command
/usr/bin/rsync -a $DATADIRS/archives/%f %p
```

Relancer la commande pg_rewind avec l'option `--restore-target-wal` :

```
$ pg_rewind --target-pgdata $DATADIRS/pgrw_srv1                          \
          --source-server "port=5637 user=replication dbname=postgres" \
          --write-recovery-conf --progress --restore-target-wal
pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/3000000 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/2000060 on timeline 1
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 282 MB (total source directory size is 308 MB)
     0/289512 kB (0%) copied
289512/289512 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

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

```
cp /tmp/postgresql.conf $DATADIRS/pgrw_srv1
```

Le paramètre `--write-recovery-conf` permet de générer le fichier `PGDATA/standby.signal` et ajoute le paramètre `primary_conninfo` au fichier `PGDATA/postgresql.auto.conf`. Vérifier leur présence.

```
$ ls $DATADIRS/pgrw_srv1/standby.signal
XXX/pgrw_srv1/standby.signal

$ postgres -D /tmp/rewind/pgrw_srv1/ -C primary_conninfo
primary_conninfo = 'user=replication passfile=''PATH_TO_HOME/.pgpass''
+++  channel_binding=disable host=''/tmp'' port=5637 sslmode=prefer
+++  sslcompression=0 ssl_min_protocol_version=TLSv1.2 gssencmode=disable
+++  krbsrvname=postgres target_session_attrs=any'
```

On constate que l'utilisateur mis en place dans la commande est celui utilisé pour faire le `pg_rewind`.

Démarrer l'ancienne primaire :

```
pg_ctl start -D $DATADIRS/pgrw_srv1 -w
```

Contrôler que la réplication fonctionne en vérifiant que la requête suivante renvoie une ligne sur la nouvelle instance primaire :

```
psql -p 5637 -xc "SELECT * FROM pg_stat_replication;"
```

À l'issue de l'opération, les droits donnés à l'utilisateur de réplication peuvent être révoqués :

```
psql -p 5637 <<_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_
```

2.12 TP - WAL_KEEP_SEGMENTS ET WAL_KEEP_SIZE

- Simulation de migration du paramétrage.

Nous allons simuler une migration de paramétrage vers PostgreSQL 13.

Ajouter `wal_keep_segments` au `postgresql.conf` :

```
$ cat << _EOF_ >> $PGDATA/postgresql.conf
wal_keep_segments  = 100
_EOF_
```

Essayer de démarrer PostgreSQL :

```
$ pg_ctl start -D $PGDATA -w

waiting for server to start....
LOG:  unrecognized configuration parameter "wal_keep_segments" in
+++ file "/home/benoit/var/lib/postgres/pgsql-13rc1/postgresql.conf" line 781
FATAL:  configuration file "/home/benoit/var/lib/postgres/pgsql-13rc1/postgresql.conf"
+++ contains errors
 stopped waiting
pg_ctl: could not start server
Examine the log output.
```

On constate que l'ancien paramètre n'est plus autorisé.

Pour déterminer la valeur de `wal_keep_size`, il faut multiplier `wal_keep_segments` par la taille d'un segment qui est généralement 16 Mo. Cette valeur peut être confirmée en consultant le *control file*.

```
$ pg_controldata $PGDATA | grep "Bytes per WAL segment"
Bytes per WAL segment:                16777216
```

Ou, si PostgreSQL est lancé :

```
# SHOW wal_segment_size ;
 wal_segment_size
------------------
 16MB
```

Éditer `postgresql.conf` avec la nouvelle valeur (`1600 MB`), puis redémarrer :

```
$ pg_ctl start -D $PGDATA -w
waiting for server to start....
LOG:  starting PostgreSQL 13beta3 on x86_64-pc-linux-gnu, compiled by gcc
```

```
+++ (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2), 64-bit
LOG:  listening on IPv6 address "::1", port 5436
LOG:  listening on IPv4 address "127.0.0.1", port 5436
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5436"
LOG:  database system was shut down at 2020-09-14 16:56:26 CEST
LOG:  database system is ready to accept connections
 done
server started
```

Vérifier la valeur de `wal_keep_size` :

```
$ psql << _EOF_
SELECT name, setting, unit
  FROM pg_settings
 WHERE name LIKE 'wal_keep_size';
_EOF_

     name      | setting | unit
---------------+---------+------
 wal_keep_size | 1600    | MB
```

2.13 TP - DÉDUPLICATION DES INDEX B-TREE

Test sur :

- la taille des index ;
- les temps de création ;
- les temps de sélection.

Mise en place

Création de la table suivante :

```
CREATE TABLE t_2col_large (
  i int,
  t text
);
```

On pourra lancer les commandes suivantes et étudier les temps pour chaque requête :

```
\timing on

INSERT INTO t_2col_large (i, t)
  SELECT g % 10000, md5((g % 10000)::text) FROM generate_series(1, 1000000) g;
CREATE INDEX t_2col_large_dedup_idx ON t_2col_large (i, t);
SELECT pg_size_pretty(pg_relation_size('t_2col_large_dedup_idx'));
DROP INDEX t_2col_large_dedup_idx;
CREATE INDEX t_2col_large_no_dedup_idx ON t_2col_large (i, t) WITH (deduplicate_items = OFF);
SELECT pg_size_pretty(pg_relation_size('t_2col_large_no_dedup_idx'));
DROP INDEX t_2col_large_no_dedup_idx;

TRUNCATE t_2col_large;
CREATE INDEX t_2col_large_dedup_idx ON t_2col_large (i, t);
INSERT INTO t_2col_large (i, t)
  SELECT g % 10000, md5((g % 10000)::text) FROM generate_series(1, 1000000) g;
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t_2col_large WHERE i>2000 and i<3000;
DROP INDEX t_2col_large_dedup_idx;

TRUNCATE t_2col_large;
CREATE INDEX t_2col_large_no_dedup_idx ON t_2col_large (i, t) WITH (deduplicate_items = OFF);
INSERT INTO t_2col_large (i, t)
  SELECT g % 10000, md5((g % 10000)::text) FROM generate_series(1, 1000000) g;
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t_2col_large WHERE i>2000 and i<3000;
DROP INDEX t_2col_large_no_dedup_idx;
```

Taille de l'index non dédupliqué : 56 Mo contre 7,3 Mo en mode dupliqué.

Création de l'index non dupliqué dans une table remplie : 1,8 s contre 0,8 s en mode dupliqué.

Insertion d'éléments dans une table : 7,2 secondes avec un index non dupliqué contre 6,1 secondes avec un index dupliqué.

Sélection en 25 ms avec un index non dupliqué contre 19 ms avec un index dupliqué.

On pourra tester en modifiant le nombre de lignes ou la proportion de lignes identiques.

NOTES

NOTES

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.