Workshop 11

Nouveautés de PostgreSQL 11

18.12

Dalibo & Contributors

https://dalibo.com/formations

Nouveautés de PostgreSQL 11

Workshop 11

TITRE : Nouveautés de PostgreSQL 11
SOUS-TITRE : Workshop 11

REVISION: 18.12

Table des Matières

NOUVEAUTÉS DE POSTGRESQL 11

Figure 1: PostgreSQL

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 11.

[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

INTRODUCTION

- Développement depuis l'été 2017
- Version bêta 1 sortie 24 mai 2018
- Bêta 2 le 28 juin
- Bêta 3 le 9 août
- Bêta 4 le 17 septembre
- Release Candidate 1 le 11 octobre
- Version finale : 18 octobre 2018
- 11.1 : le 8 novembre 2018
- 11.2 : attendue pour le 14 février 2019
- Plus de 1,5 millions de lignes de code C
- Des centaines de contributeurs

Le développement de la version 11 a suivi l'organisation habituelle : un démarrage vers la mi-2017, des *Commit Fests* tous les deux mois, un *feature freeze* le 7 avril, une première version bêta fin mai, une quatrième le 17 septembre.

La version finale est parue le 18 octobre 2018.

La version 11 de PostgreSQL contient plus de 1,5 millions de lignes de code, essentiellement du C avec environ 23 % de commentaires, garants de la qualité du code (Source : openhub.net[4]).

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

Si vous voulez en savoir plus sur le fonctionnement de la communauté PostgreSQL, une présentation récente de *Daniel Vérité* est disponible en ligne :

- Vidéo[5]
- Slides[6]

[4] https://www.openhub.net/p/postgres/analyses/latest/languages_summary
[5] https://youtu.be/NPRw0oJETGQ
[6] https://dali.bo/daniel-verite-communaute-dev-pgday

AU MENU

- Partitionnement
- Performances
- Sécurité et intégrité
- SQL & PL/pgSQL
- Outils
- Réplication
- Compatibilité
- Futur

PostgreSQL 11 apporte un grand nombre de nouvelles fonctionnalités, qui sont d'ores et déjà détaillées dans de nombreux articles. Voici quelques liens vers des articles en anglais :

- Major features of PostgreSQL 11[7] , annonce de la bêta 1 par le PGDG, 24 mai 2018
- Release Notes de la v11[8] , documentation officielle PostgreSQL
- What we (already) know about PostgreSQL 11[9] , Daniel Westermann, dbi services, pgDay.ch, 29 juin 2018
- PostgreSQL 11 New Features With Examples (Beta 1)[10] (PDF), Noriyoshi Shinoda, Hewlett-Packard Japon, mai 2018
- Waiting for PostgreSQL 11[11] , articles de blog de Teodor Sigaev tout le long du développement de la v11 entre septembre 2017 et juin 2018
- Postgres 11 Highlight[12] , série d'articles de Michael Paquier sur la même période

En français, Jean-Christophe Arnu a donné une conférence au PG Day de Marseille en juin[13] .

---

[7] https://www.postgresql.org/about/news/1855/
[8] https://www.postgresql.org/docs/11/static/release-11.html
[9] https://www.pgday.ch/common/slides/2018_westermann_WhatWeAlreadyKnowAboutPostgreSQL11.pdf
[10] https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/lcc/pdf/PostgreSQL_11_New_Features_beta1_en_20180525-1.pdf
[11] https://www.depesz.com/
[12] https://paquier.xyz/tag/11/
[13] https://www.youtube.com/watch?v=tVNo_RIZjdE

NOUVEAUTÉS SUR LE PARTITIONNEMENT

- Partitionnement par hachage
- Propagation des index
- Support de clés primaires et clés étrangères
- Mise à jour de la clé de partition
- Partition par défaut
- Amélioration des performances
- Clause `INSERT ON CONFLICT`
- Trigger `FOR EACH ROW`

Le partitionnement natif était une fonctionnalité très attendue de PostgreSQL 10. Cependant, elle souffrait de plusieurs limitations qui pouvaient dissuader de l'utiliser.

La version 11 corrige une bonne partie de ces limites.

PARTITIONNEMENT PAR HACHAGE

- Répartition des données suivant le hachage de la clé de partition
- Partitions destinées à grandir de manière uniforme

Le partitionnement par hachage permet de répartir les données équitablement sur plusieurs partitions selon la valeur de hachage de la clé de partition.

Ce mode de partitionnement est utile lorsqu'on cherche à séparer les données en plusieurs parties sans rechercher un classement particulier des enregistrements, par exemple pour répartir la charge des I/O sur plusieurs disques uniformément.

Les partitionnements par liste ou par intervalles permettent de facilement archiver ou supprimer des données. Ce n'est pas le but du partitionnement par hachage qui est plus destiné au cas où il n'y a pas de clé de partitionnement naturelle.

Le partitionnement par hachage, comme les autres modes de partitionnement permettent d'accélérer les opérations de `VACUUM`.

EXEMPLE DE PARTITIONNEMENT PAR HACHAGE

- Créer une table partitionnée :
  `CREATE TABLE t1(c1 int) PARTITION BY HASH (c1)`
- Ajouter une partition :
  ```
  CREATE TABLE t1_a PARTITION OF t1
    FOR VALUES WITH (modulus 3,remainder 0)
  ```
- Augmentation du nombre de partitions délicat

On fixe la valeur initiale du diviseur au nombre de partitions à créer. On doit créer les tables partitionnées pour tous les restes de la division entière car il n'est pas possible de définir de table par défaut avec les partitions par hachage.

```
v11=# CREATE TABLE t1(c1 int PRIMARY KEY) PARTITION BY HASH (c1);
CREATE TABLE
v11=# CREATE TABLE t1_default PARTITION OF t1 DEFAULT;
ERROR:  a hash-partitioned table may not have a default partition
v11=# CREATE TABLE t1_a PARTITION OF t1 FOR VALUES WITH (modulus 3,remainder 0);
CREATE TABLE
v11=# CREATE TABLE t1_b PARTITION OF t1 FOR VALUES WITH (modulus 3,remainder 1);
CREATE TABLE
v11=# INSERT INTO t1 SELECT generate_series(0,10000);
ERROR:  no partition of relation "t1" found for row
DÉTAIL : Partition key of the failing row contains (c1) = (0).
v11=# CREATE TABLE t1_c PARTITION OF t1 FOR VALUES WITH (modulus 3,remainder 2);
CREATE TABLE

v11=# \d+ t1
                          Table « public.t1 »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut | Stockage |
---------+---------+-----------------+-----------+------------+----------+
 c1      | integer |                 | not null  |            | plain    |
Clé de partition : HASH (c1)
Index :
    "t1_pkey" PRIMARY KEY, btree (c1)
Partitions: t1_a FOR VALUES WITH (modulus 3, remainder 0),
            t1_b FOR VALUES WITH (modulus 3, remainder 1),
            t1_c FOR VALUES WITH (modulus 3, remainder 2)
```

Pour trier les données dans la bonne colonne, la classe d'opérateur par hachage par défaut des colonnes de la clé est utilisée. Il ne s'agit pas de l'opération modulo mathématique. On le voit bien en regardant le nombre d'insertions dans chaque partition pour une liste d'entiers de 0 à 10 000, très homogène, et les valeurs dans une partition :

```
v11=# INSERT INTO t1 SELECT generate_series(0,10000);
INSERT 0 10001
```

```
v11=# SELECT count(*) FROM t1;
 count
-------
 10001
(1 ligne)

v11=# SELECT count(*) FROM t1_a;
 count
-------
  3277
(1 ligne)
CREATE TABLE t1_a PARTITION OF t1 FOR VALUES WITH (modulus 3,remainder 0);
v11=# SELECT count(*) FROM t1_b;
 count
-------
  3369
(1 ligne)

v11=# SELECT COUNT(*) FROM t1_c;
 count
-------
  3355
(1 ligne)

v11=# SELECT * FROM t1_c limit 10 ;
 c1
----
  0
  1
  5
  9
 11
 12
 17
 22
 23
 24
```

Il n'existe pas de commande permettant d'étendre automatiquement le nombre de partitions d'une table partitionnée par hachage. Cependant, la documentation officielle[14] dit ceci : « il n'est pas obligatoire que chaque partition ait le même diviseur, juste que chaque diviseur apparaissant dans une table partitionnée par hachage soit un facteur du diviseur immédiatement supérieur. Cela permet d'augmenter le nombre de partitions de manière incrémentale sans avoir besoin de déplacer toutes les données d'un coup. »

[14] https://docs.postgresql.fr/11/sql-createtable.html

On peut donc contourner en détachant une partition et créant des « sous-partitions » (en changeant le diviseur) de cette partition et réinsérer les données de la table détachée dans la table mère.

```
v11=# BEGIN;
BEGIN
v11=# ALTER TABLE t1 DETACH PARTITION t1_a;
ALTER TABLE
v11=# CREATE TABLE t1_aa PARTITION OF t1 FOR VALUES WITH (modulus 6,remainder 0);
CREATE TABLE
v11=# CREATE TABLE t1_ab PARTITION OF t1 FOR VALUES WITH (modulus 6,remainder 3);
CREATE TABLE
v11=# INSERT INTO t1 SELECT * from t1_a;
INSERT 0 3277
v11=# DROP TABLE t1_a;
DROP TABLE
v11=# COMMIT;
COMMIT
v11=# SELECT SUM(c) count_aa_ab FROM (
  SELECT count(*) c FROM t1_aa
  UNION SELECT count(*) FROM t1_ab) t;
 count_aa_ab
-------------
        3277
(1 ligne)

v11=# \d+ t1
                          Table « public.t1 »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut | Stockage |
---------+---------+-----------------+-----------+------------+----------+
 c1      | integer |                 | not null  |            | plain    |
Clé de partition : HASH (c1)
Index :
    "t1_pkey" PRIMARY KEY, btree (c1)
Partitions: t1_aa FOR VALUES WITH (modulus 6, remainder 0),
            t1_ab FOR VALUES WITH (modulus 6, remainder 3),
            t1_b FOR VALUES WITH (modulus 3, remainder 1),
            t1_c FOR VALUES WITH (modulus 3, remainder 2)
```

Toutes les lignes de la table recoupée `t1_a` ont bien été insérées dans les 2 nouvelles partitions `t1_aa` et `t1_ab`.

PROPAGATION DES INDEX SUR LES PARTITIONS

- Index sur une table partitionnée
- Index créé sur chaque partition
  - Hors partitions distantes
- Création automatique sur toute nouvelle partition

Soit la table partitionnée par intervalles :

```
CREATE TABLE livres (titre text, parution timestamp with time zone)
  PARTITION BY RANGE (titre);
CREATE TABLE livres_a_m PARTITION OF livres FOR VALUES FROM ('a') TO ('m');
CREATE TABLE livres_m_z PARTITION OF livres FOR VALUES FROM ('m') TO ('zzz');
```

En version 10, il n'était pas possible de créer un index sur une table partitionnée :

```
v10=# CREATE INDEX ON livres (titre);
ERROR:  cannot create index on partitioned table "livres"
```

En version 11, les index apparaissent sur la table partitionnée mais sont bien créés sur chaque partition :

```
v11=# CREATE INDEX ON livres (titre);
CREATE INDEX
v11=# \d livres
                          Table « public.livres »
 Colonne  |           Type           | Collationnement | NULL-able | Par défaut
----------+--------------------------+-----------------+-----------+------------
 titre    | text                     |                 |           |
 parution | timestamp with time zone |                 |           |
Clé de partition : RANGE (titre)
Index :
    "livres_titre_idx" btree (titre)
Nombre de partitions : 2 (utilisez \d+ pour les lister)
v11=# \d livres_a_m
                        Table « public.livres_a_m »
 Colonne  |           Type           | Collationnement | NULL-able | Par défaut
----------+--------------------------+-----------------+-----------+------------
 titre    | text                     |                 |           |
 parution | timestamp with time zone |                 |           |
Partition de : livres FOR VALUES FROM ('a') TO ('m')
Index :
    "livres_a_m_titre_idx" btree (titre)
```

Si on crée une nouvelle partition, l'index sera créé automatiquement :

```
v11=# CREATE TABLE livres_0_9 PARTITION OF livres FOR VALUES FROM ('0') TO ('a');
CREATE TABLE
```

```
v11=# \d livres_0_9
                        Table « public.livres_0_9 »
 Colonne  |           Type           | Collationnement | NULL-able | Par défaut
----------+--------------------------+-----------------+-----------+------------
 titre    | text                     |                 |           |
 parution | timestamp with time zone |                 |           |
Partition de : livres FOR VALUES FROM ('0') TO ('a')
Index :
    "livres_0_9_titre_idx" btree (titre)
```

L'exemple ci-dessus concerne une colonne de la clé de partitionnement, mais cela fonctionne avec toute colonne.

La propagation ne fonctionne pas sur les partitions qui sont des tables distantes : la création d'index y est impossible, il faut le faire sur la table source. Pire : des index à propager existants interdisent d'attacher une partition distante, et la présence d'une partition distante interdit de créer tout index sur la table partitionnée. Il faut créer les index manuellement sur chaque partition.

SUPPORT DES CLÉS PRIMAIRES

- Support des index `UNIQUE`
- Permet la création de clés primaires
- Uniquement si l'index comprend la clé de partition

La version 11 offre la possibilité de créer des index sur des tables partitionnées. Si l'index contient la clé de partition, il est possible de créer un index unique :

```
v11=# CREATE UNIQUE INDEX ON livres (titre);
CREATE INDEX
v11=# \d livres;
                        Table « public.livres »
 Colonne  |           Type           | Collationnement | NULL-able | Par défaut
----------+--------------------------+-----------------+-----------+------------
 titre    | text                     |                 |           |
 parution | timestamp with time zone |                 |           |
Clé de partition : RANGE (titre)
Index :
    "livres_titre_idx" UNIQUE, btree (titre)
Nombre de partitions : 3 (utilisez \d+ pour les lister)
```

Cela n'est pas possible sur des colonnes en dehors de la clé de partition :

```
v11=# CREATE UNIQUE INDEX ON livres (parution);
ERROR:  insufficient columns in UNIQUE constraint definition
DÉTAIL : UNIQUE constraint on table "livres" lacks column "titre" which is part
         of the partition key.
```

Cette nouvelle fonctionnalité permet la création de clés primaires sur la clé de partition :

```
v11=# CREATE TABLE livres_primary_key (
    titre text PRIMARY KEY, parution timestamp with time zone)
  PARTITION BY RANGE (titre);
CREATE TABLE
v11=# \d livres_primary_key;
                    Table « public.livres_primary_key »
 Colonne  |           Type           | Collationnement | NULL-able | Par défaut
----------+--------------------------+-----------------+-----------+------------
 titre    | text                     |                 | not null  |
 parution | timestamp with time zone |                 |           |
Clé de partition : RANGE (titre)
Index :
    "livres_primary_key_pkey" PRIMARY KEY, btree (titre)
Number of partitions: 0
```

SUPPORT DES CLÉS ÉTRANGÈRES

- Clé étrangère depuis une table partitionnée
- Clé étrangère vers une table partitionnée toujours impossible
  - mais possible vers une partition spécifique

En version 10 les clés étrangères ne sont pas supportées dans une partition :

```
v10=# CREATE TABLE auteurs (nom text PRIMARY KEY);
CREATE TABLE
v10=# CREATE TABLE bibliographie
   (titre text, auteur text REFERENCES auteurs(nom))
   PARTITION BY RANGE (titre);
ERROR:  foreign key constraints are not supported on partitioned tables
LIGNE 2 :                      auteur text REFERENCES auteurs(nom))
                                          ^
```

La version 11 supporte les clés étrangères sur les partitions. Il faut bien sûr une contrainte :

```
v11=# CREATE TABLE auteurs (nom text PRIMARY KEY);
CREATE TABLE
v11=# CREATE TABLE bibliographie
```

```
    (titre text PRIMARY KEY, auteur text REFERENCES auteurs(nom))
    PARTITION BY RANGE (titre);
CREATE TABLE
v11=# \d bibliographie
              Table « public.bibliographie »
 Colonne | Type | Collationnement | NULL-able | Par défaut
---------+-- ---+-----------------+-----------+------------
 titre   | text |                 |           |
 auteur  | text |                 |           |
Clé de partition : RANGE (titre)
Contraintes de clés étrangères :
    "bibliographie_auteur_fkey" FOREIGN KEY (auteur) REFERENCES auteurs(nom)
Number of partitions: 0
```

Les clés étrangères depuis n'importe quelle table vers une table partitionnée sont cependant toujours impossibles :

```
v11=# CREATE TABLE avis_livre (avis text, livre text REFERENCES bibliographie(titre));
ERROR:  cannot reference partitioned table "livres"
```

On peut cependant créer une clé étrangère vers une partition donnée de la table. Ceci ne correspondra qu'à des cas d'usage bien spécifiques :

```
v11=# CREATE TABLE avis_livres_a_m (
  nom text, livre text REFERENCES livres_a_m(titre))
    PARTITION BY RANGE (nom);
CREATE TABLE
```

MISE À JOUR D'UNE VALEUR DE LA CLÉ DE PARTITION

- En version 10 : `DELETE` puis `INSERT` obligatoires si clé modifiée
- En version 11 : `UPDATE` fonctionne
  - Ligne déplacée dans une nouvelle partition

En version 10, il n'était pas possible de mettre à jour une clé de partition entre deux partitions différentes avec la commande `UPDATE`, il était nécessaire de faire un `DELETE` puis un `INSERT`.

En version 11, PostgreSQL rend la chose transparente.

PARTITION PAR DÉFAUT

- Pour les données n'appartenant à aucune autre partition :
  `CREATE TABLE livres_default PARTITION OF livres DEFAULT;`

PostgreSQL génère une erreur lorsque les données n'appartiennent à aucune partition :

```
v11=# INSERT INTO livres VALUES ('zzzz', now());
ERROR:  no partition of relation "livres" found for row
DÉTAIL : Partition key of the failing row contains (titre) = (zzzz).
```

En version 11, il est possible de définir une partition par défaut où iront les données sans partition explicite :

```
v11=# CREATE TABLE livres_default PARTITION OF livres DEFAULT;
CREATE TABLE
v11=# INSERT INTO livres VALUES ('zzzz', now());
INSERT 0 1
```

Attention, on ne pourra pas ensuite créer de partition dont la contrainte contiendrait des lignes présentes dans la partition par défaut :

```
v11=# CREATE TABLE livres_zzz_zzzzz PARTITION OF livres
  FOR VALUES FROM ('zzz') TO ('zzzzz');
ERROR:  updated partition constraint for default partition "livres_default"
        would be violated by some row
```

Le contournement est le suivant : créer la partition en dehors de la table partitionnée, insérer les enregistrements de la table par défaut dans la nouvelle table, supprimer ces enregistrements de la table par défaut et attacher la table comme nouvelle partition.

```
v11=# BEGIN;
BEGIN
v11=# CREATE TABLE livres_zzz_zzzzz (
  titre text CHECK (titre>='zzz' AND titre<'zzzzz'),
  parution timestamp with time zone);
CREATE TABLE
v11=# INSERT INTO livres_zzz_zzzzz
  SELECT * FROM livres_default WHERE (titre>='zzz' AND titre<'zzzzz');
INSERT 0 1
v11=# DELETE FROM livres_default WHERE (titre>='zzz' AND titre<'zzzzz');
DELETE 1
v11=# ALTER TABLE livres ATTACH PARTITION livres_zzz_zzzzz
  FOR VALUES FROM ('zzz') TO ('zzzzz');
ALTER TABLE
v11=# COMMIT;
COMMIT
```

PERFORMANCE & PARTITIONS

- Amélioration de l'algorithme d'élagage
- `enable_partition_pruning`
- Élagage dynamique des partitions, à l'exécution

La possibilité d'élaguer l'arbre de recherche lors de la planification était déjà présente depuis la version 8.4. Il s'agit d'utiliser les contraintes des tables pour exclure de la recherche certaines tables. Il était contrôlé par le paramètre `constraint_exclusion`. Le calcul des tables à exclure impacte le temps de planification. Pour cette raison, le paramètre est fixé à la valeur `partition` pour indiquer que ce mode doit être activé uniquement pour des tables partitionnées.

Si ce paramètre existe toujours en version 11, il ne s'applique plus qu'aux tables partitionnées par héritage (l'ancienne méthode, toujours utilisable), donc si des contraintes `CHECK` sont aussi utilisées. Le paramètre `enable_partition_pruning`, activé par défaut, a été ajouté pour gèrer l'élagage des partitions natives.

L'algorithme d'élagage a été amélioré. D'une recherche exhaustive et linéaire, on passe à une recherche binaire pour les partitions par liste ou intervalles. Une fonction de hachage est utilisée dans le cas des partitions par hachage.

Une autre nouveauté est la possibilité pour le moteur, non seulement d'élaguer à la planification, mais aussi lors de l'exécution de la requête ! Cette amélioration est visible en effectuant un `EXPLAIN ANALYZE` de la requête.

Insérons des données dans la table partitionnée `livres` déjà utilisée plus haut :

```
INSERT INTO livres
SELECT md5(i::text)::text, now() - i*'1 week'::interval
FROM generate_series(1,100000) i;
```

Nous allons rechercher tous les livres avec des titres commençant par les lettres `a` à `c`. On va cependant émuler le cas où le deuxième paramètre provient d'une sous-requête. Si on désactive l'élagage des partitions, on obtient un parcours de toutes les partitions :

```
v11=# SET enable_partition_pruning = off;
SET
v11=# EXPLAIN (COSTS off)
SELECT * FROM livres WHERE titre BETWEEN 'a' AND (SELECT 'c');
                        QUERY PLAN
----------------------------------------------------------
 Append
   InitPlan 1 (returns $0)
     ->  Result
   ->  Seq Scan on livres_0_9 l
```

```
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
   ->  Seq Scan on livres_a_m l_1
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
   ->  Seq Scan on livres_m_z l_2
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
(9 lignes)
```

Lorsque l'élagage est activé, le moteur détecte qu'il n'a pas besoin de parcourir la partition `livres_0_9`, et ce dès la phase de planification, et sans qu'il y ait besoin d'un index sur la clé, puisque que la valeur minimum 'a' est en clair dans la requête.
Par contre le planificateur prévoit de parcourir `livres_m_z` car il n'est pas immédiat pour lui que l'on s'arrêtera à 'c' :

```
v11=# SET enable_partition_pruning = on;
SET
v11=# EXPLAIN (COSTS off)
SELECT * FROM livres l WHERE titre BETWEEN 'a' AND (SELECT 'c');
                        QUERY PLAN
----------------------------------------------------------
 Append
   InitPlan 1 (returns $0)
     ->  Result
   ->  Seq Scan on livres_a_m l
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
   ->  Seq Scan on livres_m_z l_1
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
(7 lignes)
```

Regardons toutefois le plan d'une exécution réelle :

```
v11=# EXPLAIN (ANALYSE, COSTS off)
SELECT * FROM livres l WHERE titre BETWEEN 'a' and (select 'c');
                                  QUERY PLAN
------------------------------------------------------------------------------
 Append (actual time=0.063..13.350 rows=12405 loops=1)
   InitPlan 1 (returns $0)
     ->  Result (actual time=0.003..0.003 rows=1 loops=1)
   ->  Seq Scan on livres_a_m l (actual time=0.041..12.206 rows=12405 loops=1)
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
         Rows Removed by Filter: 25065
   ->  Seq Scan on livres_m_z l_1 (never executed)
         Filter: ((titre >= 'a'::text) AND (titre <= $0))
 Planning Time: 0.479 ms
 Execution Time: 14.059 ms
(10 lignes)
```

Nous constatons que lors de l'exécution, le parcours de la partition `livres_m_z` prévu par

le planificateur est annulé : `never executed`.

Cet élagage dynamique pourra être effectué sur toute expression stable. Par exemple, un appel à une fonction *stable* ou *immutable*, donc une expression constante, un calcul, la fonction `now()`, mais pas la fonction `random()`.

L'élagage dynamique est également utilisable dans les instructions préparées et les jointures en *Nested Loops* comme décrit dans ce billet de blog de Thomas Reiss[15] .

AUTRES NOUVEAUTÉS DU PARTITIONNEMENT

- Clause `INSERT ON CONFLICT`
  - sauf mise à jour de clé
- *Partition-Wise Aggregate* (par défaut : `off`)
- Triggers `AFTER ... FOR EACH ROW`
- Partitions distantes : routage pour les insertions
  - uniquement postgres_fdw
  - pas de propagation des index
  - sharding !

ON CONFLICT

En version 10, la clause `ON CONFLICT` n'était pas supportée sur le partitionnement :

```
v10=# INSERT INTO livres VALUES ('mon titre') ON CONFLICT DO NOTHING;
ERROR:  ON CONFLICT clause is not supported with partitioned tables
```

En version 11 la clause fonctionne :

```
v11=# INSERT INTO livres VALUES ('mon titre') ON CONFLICT DO NOTHING;
INSERT 0 1
v11=# INSERT INTO livres VALUES ('mon titre') ON CONFLICT DO NOTHING;
INSERT 0 0
```

Il reste une limite : une clause `ON CONFLICT UPDATE` ne doit pas mettre à jour la clé de partition, ce qui ne devrait pas être un problème majeur dans les cas d'utilisation des partitions.

Partition-Wise Aggregate

Les paramètres `enable_partitionwise_join` et `enable_partitionwise_aggregate` ont été ajoutés. Ils sont désactivés par défaut en raison du coût supplémentaire qu'ils entraînent dans la planification.

[15] http://blog.frosties.org/post/2018/05/23/PostgreSQL-11-dynamic_pruning

En cas de jointure entre plusieurs tables partitionnées avec les mêmes contraintes, le moteur va d'abord effectuer des jointures entre les différentes partitions de chaque table. Il joindra dans un deuxième temps ces résultats entre eux.

L'activation de ces nouveaux paramètres va permettre au moteur d'effectuer dans un premier temps les jointures entre les partitions de différentes tables possédant les mêmes contraintes. Il effectuera dans un deuxième temps la jointure des résultats entre eux.

```
CREATE TABLE t2(c1 int) PARTITION BY HASH (c1);
CREATE TABLE t2_a PARTITION OF t2 FOR VALUES WITH (modulus 2,remainder 0);
CREATE TABLE t2_b PARTITION OF t2 FOR VALUES WITH (modulus 2,remainder 1);
INSERT INTO t2 SELECT generate_series(0,200000);
CREATE TABLE t3(c1 int) PARTITION BY HASH (c1);
CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 2,remainder 0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 2,remainder 1);
INSERT INTO t3 SELECT generate_series(0,400000);
VACUUM ANALYSE t2, t3;
```

Pour plus de simplicité dans la lecture des plans d'exécution, nous désactivons la parallélisation. Il faut noter que les optimisations décrites fonctionnent en mode parallélisé :

```
v11=# SET max_parallel_workers_per_gather=0;
SET
```

Voici le plan sans les optimisations. Les jointures sont effectuées entre les partitions d'une même table :

```
v11=# EXPLAIN (COSTS off) SELECT count(*) FROM t2 INNER JOIN t3 ON t2.c1=t3.c1;
               QUERY PLAN
----------------------------------------
 Aggregate
   ->  Hash Join
         Hash Cond: (t3_a.c1 = t2_a.c1)
         ->  Append
               ->  Seq Scan on t3_a
               ->  Seq Scan on t3_b
         ->  Hash
               ->  Append
                     ->  Seq Scan on t2_a
                     ->  Seq Scan on t2_b
(10 lignes)
```

Voici le plan avec l'activation de la jointure des partitions, `enable_partitionwise_join`. On remarque que les jointures se font en premier lieu entre les partitions de même type :

```
v11=# SET enable_partitionwise_join = on;
SET
v11=# EXPLAIN (COSTS off) SELECT count(*) FROM t2 INNER JOIN t3 ON t2.c1=t3.c1;
                  QUERY PLAN
```

```
-----------------------------------------------
 Aggregate
   ->  Append
         ->  Hash Join
               Hash Cond: (t3_a.c1 = t2_a.c1)
               ->  Seq Scan on t3_a
               ->  Hash
                     ->  Seq Scan on t2_a
         ->  Hash Join
               Hash Cond: (t3_b.c1 = t2_b.c1)
               ->  Seq Scan on t3_b
               ->  Hash
                     ->  Seq Scan on t2_b
(12 lignes)
```

Voici le plan avec l'activation de l'agrégation et le regroupement des partitions, `enable_partitionwise_aggregate`. Une fois les jointures entre les partitions de même type effectuées, une agrégation partielle de celles-ci sont effectuées avant l'agrégation finale entre les différentes jointures :

```
v11=# SET enable_partitionwise_aggregate = on;
SET
v11=# EXPLAIN (COSTS off) SELECT count(*) FROM t2 INNER JOIN t3 ON t2.c1=t3.c1;
                      QUERY PLAN
-----------------------------------------------------
 Finalize Aggregate
   ->  Append
         ->  Partial Aggregate
               ->  Hash Join
                     Hash Cond: (t3_a.c1 = t2_a.c1)
                     ->  Seq Scan on t3_a
                     ->  Hash
                           ->  Seq Scan on t2_a
         ->  Partial Aggregate
               ->  Hash Join
                     Hash Cond: (t3_b.c1 = t2_b.c1)
                     ->  Seq Scan on t3_b
                     ->  Hash
                           ->  Seq Scan on t2_b
(14 lignes)
```

Les tables partitionnées acceptent à présent les triggers `AFTER UPDATE ... FOR EACH ROW`. La propagation du trigger, comme les index, est automatique sur chaque partition. Les triggers `BEFORE UPDATE` ne sont pas supportés mais il reste possible de les créer sur chaque partition.

Partitions distantes

En v10, les tables partitionnées pouvaient déjà être utilisées comme partition, et dès la déclaration :

```
CREATE FOREIGN TABLE capteur_2020
PARTITION OF capteur
FOR VALUES FROM ('01-01-2020') TO ('01-01-2021')
SERVER loin OPTIONS (table_name 'capteur_2020') ;
```

Il était possible de lire sans problème, mais on ne pouvait insérer dans la table distante qu'en la désignant explicitement, sous peine d'erreur :

```
v10=#INSERT INTO capteur SELECT '01-01-2020' ;
ERROR:  cannot route inserted tuples to a foreign table
v10=#INSERT INTO capteur_2020 SELECT '01-01-2020' ;
INSERT 0 1
```

En v11, cette restriction a disparu, l'insertion directement dans la table partitionnée fonctionne.

Cela ouvre d'intéressantes perspectives en terme de *sharding* (répartition d'une table sur plusieurs instances).

Cependant, la création d'un index sur une table distante n'étant pas possible, la propagation des index reste donc manuelle.

PERFORMANCES

- Compilation *Just In Time* (JIT)
- Parallélisme étendu à plusieurs commandes
- `ALTER TABLE ADD COLUMN ... DEFAULT ...` sans réécriture

JIT : LA COMPILATION À LA VOLÉE

- Compilation *Just In Time* des requêtes
- Utilise le compilateur LLVM
- Vérifier que l'installation est fonctionnelle
- Désactivé par défaut

Une des nouveautés les plus visibles et techniquement pointues de la v11 est la « compilation à la volée » (*Just In Time compilation*, ou JIT) des requêtes SQL. C'est le fruit de deux ans de travail d'Andres Freund notamment.

Dans certaines requêtes, l'essentiel du temps est passé à décoder des enregistrements (*tuple deforming*), à analyser des clauses `WHERE`, à effectuer des calculs. L'idée du JIT est de transformer tout ou partie de la requête en un programme natif directement exécuté par le processeur.

Cette compilation est une opération lourde qui ne sera effectuée que pour des requêtes qui en valent le coup.

Le JIT de PostgreSQL s'appuie actuellement sur la chaîne de compilation LLVM, choisie pour sa flexibilité. L'utilisation nécessite un PostgreSQL compilé avec l'option `--with-llvm` et l'installation des bibliothèques de LLVM. Avec les paquets du PGDG, c'est le cas par défaut sur Debian/Ubuntu. Sur CentOS/Red Hat 7 il faut penser à installer le package `postgresql11-llvmjit`. CentOS/Red Hat 6 ne permettent actuellement pas d'utiliser le JIT.

Si PostgreSQL ne trouve pas les bibliothèques nécessaires, il ne renvoie pas d'erreur et continue sans tenter de JIT. Pour tester que le JIT est fonctionnel sur votre machine, il doit apparaître dans un plan quand on force son utilisation ainsi :

```
SET jit=on;
SET jit_above_cost TO 0 ;
EXPLAIN (ANALYZE) SELECT 1;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=4) (actual time=1.041..1.041 rows=1 loops=1)
```

```
Planning Time: 0.016 ms
JIT:
  Functions: 1
  Generation Time: 0.155 ms
  Inlining: false
  Inlining Time: 0.000 ms
  Optimization: false
  Optimization Time: 0.093 ms
  Emission Time: 0.881 ms
Execution Time: 1.242 ms
```

Il a été décidé que le JIT serait désactivé par défaut[16] en version 11. Cela est fréquemment le cas pour les nouvelles fonctionnalités pouvant avoir des effets de bords négatifs, en attendant les retours d'expérience. On verra plus bas que l'activer est très simple.

La documentation officielle est assez accessible : https://doc.postgresql.fr/11/jit.html

---

JIT : QU'EST-CE QUI COMPILÉ ?

- *Tuple deforming*
- Évaluation d'expressions :
  - `WHERE`
  - Agrégats, `GROUP BY`
- Appels de fonctions (*inlining*)
- Mais pas les jointures

Le JIT ne peut pas encore compiler toute une requête. La version actuelle se concentre sur des goulots d'étranglements classiques :

- le décodage des enregistrements (*tuple deforming*) pour en extraire les champs intéressants ;
- les évaluations d'expressions, notamment dans les clauses `WHERE` pour filtrer les lignes ;
- les agrégats, les `GROUP BY`…

Les jointures ne sont pas (encore ?) concernées par le JIT.

Le code résultant est utilisable plus efficacement avec les processeurs actuels qui utilisent les pipelines et les prédictions de branchement.

[16] https://www.postgresql.org/message-id/flat/20180914222657.mw25esrzbcnu6qlu%40alap3.anarazel.de

Pour les détails, on peut consulter notamment cette conférence très technique au FOSDEM 2018[17] par l'auteur principal du JIT, Andres Freund.

JIT : ALGORITHME « NAÏF »

- `jit` (défaut : `off`)
- `jit_above_cost` (défaut : 100 000)
- `jit_inline_above_cost` (défaut : 500 000)
- `jit_optimize_above_cost` (défaut : 500 000)
- à comparer au coût de la requête… I/O comprises
- Seuils arbitraires !

De l'avis même de son auteur, l'algorithme de déclenchement du JIT est « naïf ». Quatre paramètres existent (hors débogage).

Par défaut, `jit` est à `off`. `jit = on` active le JIT **si** l'environnement technique évoqué plus haut le permet. Il est préférable de n'activer le JIT qu'au cas par cas pour les utilisateurs, bases, ou requêtes qui pourraient en profiter.

La compilation n'a cependant lieu que pour un coût de requête calculé d'au moins `jit_above_cost`, valeur par défaut assez élevée.

Puis, si le coût atteint `jit_inline_above_cost`, certaines fonctions utilisées par la requête et supportées par le JIT sont intégrées dans la compilation. Si `jit_optimize_above_cost` est atteint, une optimisation du code compilé est également effectuée. Ces deux dernières opérations étant longues, elles ne le sont que pour des coûts assez importants.

Ces seuils sont à comparer avec les coûts des requêtes, qui incluent les entrées-sorties, donc pas seulement le coût CPU. Ces seuils sont un peu arbitraires et nécessiteront sans doute un certain tuning en fonction de vos requêtes.

---

[17] https://archive.fosdem.org/2018/schedule/event/jiting_postgresql_using_llvm/

EXEMPLE DE PLAN D'EXÉCUTION AVEC JIT

```
Planning Time: 0.553 ms
JIT:
  Functions: 27
  Generation Time: 7.058 ms
  Inlining: true
  Inlining Time: 16.028 ms
  Optimization: true
  Optimization Time: 617.294 ms
  Emission Time: 425.744 ms
Execution Time: 29402.666 ms
```

Si le JIT est activé dans une requête, le plan d'exécution est complété, à la fin, des informations suivantes :

- le nombre de fonctions concernées ;
- les temps de génération, d'inclusion des fonctions, d'optimisation du code compilé…

Dans l'exemple ci-dessus, on peut constater que ces coûts ne sont pas négligeables par rapport au temps total. Il reste à voir si ce temps perdu est récupéré sur le temps d'exécution de la requête, ce qui en pratique n'a rien d'évident.

Sans JIT la durée de la requête était d'environ 33 s.

QUAND LE JIT EST-IL UTILE ?

- Pas de limitation par les I/O
- Requêtes complexes (calculs, agrégats, appels de fonctions…)
- Beaucoup de lignes, filtres
- Assez longues pour « rentabiliser » le JIT
- Analytiques, pas ERP

Vu son coût élevé, le JIT n'a d'intérêt que pour les requêtes utilisant beaucoup le CPU, donc effectuant des opérations sur beaucoup de lignes : calculs d'expression, filtrage, agrégats.

Ce seront donc plus des requêtes analytiques brassant beaucoup de lignes que les petites requêtes d'un ERP.

Il n'y a pas non plus de mise en cache du code compilé.

Si gain il y a, il est relativement modeste en-deçà de quelques millions de lignes, et devient de plus important au fur et à mesure que la volumétrie augmente. Cela à condition bien sûr que d'autres limites n'apparaissent pas (bande passante…).

Documentation officielle : https://docs.postgresql.fr/11/jit-decision.html

---

PARALLÉLISME : NOUVELLES AMÉLIORATIONS

- Nœuds Append (`UNION ALL`)
- Jointures type Hash
- `CREATE TABLE AS SELECT...`
- `CREATE MATERIALIZED VIEW`
- `SELECT INTO`
- `CREATE INDEX` (B-tree)
  - nouveau paramètre `max_parallel_maintenance_workers`

La parallélisation des requêtes avait été introduite en version 9.6, sur certains nœuds d'exécution seulement, et pour les requêtes en lecture seule uniquement. La version 10 l'avait étendue à d'autres nœuds.

Des nœuds supplémentaires peuvent à présent être parallélisés, notamment ceux de type *Append*, qui servent aux `UNION ALL` notamment :

Jointure type *Hash*

Un nœud déjà parallélisé a été amélioré, le *Hash join* (jointure par hachage).
Soit les tables suivantes :

```
CREATE TABLE a AS SELECT i FROM generate_series(1,10000000) i ;
CREATE TABLE b as SELECT i FROM generate_series(1,10000000) i ;
CREATE INDEX ON a(i) ;

SET work_mem TO '1GB' ;
SET max_parallel_workers_per_gather TO 2;
```

Dans la version 10, le *hash join* est déjà parallélisé :

```
v10=# EXPLAIN (COSTS off) SELECT * FROM a INNER JOIN b on (a.i=b.i)
        WHERE a.i BETWEEN 500000 AND 900000;
                              QUERY PLAN
---------------------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Hash Join
```

```
         Hash Cond: (b.i = a.i)
         ->  Parallel Seq Scan on b
         ->  Hash
               ->  Index Only Scan using a_i_idx on a
                     Index Cond: ((i >= 500000) AND (i <= 900000))
(8 lignes)
```

Mais les deux *hashs* en s'exécutant font le travail en double. En version 11, ils partagent la même table de travail et peuvent donc paralléliser sa construction (ici en parallélisant l'*Index Scan*) :

```
v11=# SET enable_parallel_hash = on;
SET
v11=# EXPLAIN (COSTS off) SELECT * FROM a INNER JOIN b on (a.i=b.i)
        WHERE a.i BETWEEN 500000 AND 900000;
                              QUERY PLAN
----------------------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Hash Join
         Hash Cond: (b.i = a.i)
         ->  Parallel Seq Scan on b
         ->  Parallel Hash
               ->  Parallel Index Only Scan using a_i_idx on a
                     Index Cond: ((i >= 500000) AND (i <= 900000))
(8 lignes)
```

L'auteur de cette optimisation a écrit un article assez complet[18] .

Création d'index

La création d'index B-tree peut à présent être parallélisée, ce qui va permettre de gros gains de temps dans certains cas. La parallélisation est activée par défaut et est contrôlée par un nouveau paramètre, `max_parallel_maintenance_workers`, par défaut à 2, et non l'habituel `max_parallel_workers_per_gather`.

```
v11=# SET maintenance_work_mem TO '2GB';
SET
v11=# CREATE TABLE t9 AS SELECT random() FROM  generate_series(1,5000000);
CREATE TABLE
v11=# SET max_parallel_maintenance_workers TO 0;
SET
v11=# \timing on
Chronométrage activé.

v11=# CREATE index ix_t9 ON t9 (random);
```

[18] https://write-skew.blogspot.com/2018/01/parallel-hash-for-postgresql.html

```
CREATE INDEX
Durée : 86731,660 ms (01:26,732)
v11=# DROP INDEX ix_t9 ;
DROP INDEX
v11=# SET max_parallel_maintenance_workers TO 4;
SET
v11=# CREATE index ix_t9 ON t9 (random) ;
CREATE INDEX
Durée : 67278,338 ms (01:07,278)
```

Le gain en temps est dans cet exemple de plus de 20 % pour 4 workers.

La commande `ALTER TABLE t9 SET (parallel_workers = 4);` permet de fixer le nombre de workers au niveau de la définition de la table, mais attention cela va aussi impacter vos requêtes !

Pour de plus amples détails, Cybertec a publié un article sur le sujet[19] .

ALTER TABLE ADD COLUMN ... DEFAULT ... SANS RÉÉCRITURE

- `ALTER TABLE ADD COLUMN ... DEFAULT ...`
  - v10 : réécriture complète de la table !
  - v11 : valeur par défaut mémorisée, ajout instantané
  - ... si le défaut n'est pas une fonction volatile

Jusqu'en version 10 incluse, l'ajout d'une colonne avec une valeur `DEFAULT` (à raison de plus avec `NOT NULL`) provoquait la réécriture complète de la table, en bloquant tous les accès. Sur de grosses tables, l'interruption de service était parfois intolérable et menait à des mises à jour par étapes délicates.

La version 11 prend simplement note de la valeur par défaut de la nouvelle colonne et n'a pas besoin de l'écrire physiquement pour la restituer ensuite.

Cette valeur par défaut doit être soit une constante pendant l'ordre, ce qui est le cas de `DEFAULT 1234`, `DEFAULT now()` ou de toute fonction déclarée comme `STABLE` ou `IMMUTABLE`, mais pas de `DEFAULT clock_timestamp()` par exemple. Si la valeur par défaut est fournie par une fonction déclarée comme, ou implicitement `VOLATILE`, la réécriture de la table est nécessaire.

Le verrou *Access Exclusive* reste nécessaire, et peut entraîner quelques attentes, mais il est relâché beaucoup plus rapidement que si la réécriture était nécessaire.

[19] https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-better-performance/

La table n'est donc pas réécrite ni ne change de taille. Par la suite, chaque ligne modifiée sera réécrite en intégrant la valeur par défaut. De même, un `VACUUM FULL` réécrira la table avec ces valeurs par défaut, donnant au final une table potentiellement beaucoup plus grande qu'avant le `VACUUM` !

La table système `pg_attribute` contient 2 nouveaux champs `atthasmissing` et `attmissingval` indiquant si un champ possède une telle valeur par défaut :

```
v11=# ALTER TABLE ajouts ADD COLUMN d3 timetz DEFAULT (now()) ;
ALTER TABLE

v11=# SELECT * FROM pg_attribute
   WHERE attrelid = (SELECT oid FROM pg_class WHERE relname='ajouts')
   and atthasmissing = 't' \gx

-[ RECORD 1 ]-+---------------------
attrelid      | 69352
attname       | d3
atttypid      | 1266
attstattarget | -1
attlen        | 12
attnum        | 7
attndims      | 0
attcacheoff   | -1
atttypmod     | -1
attbyval      | f
attstorage    | p
attalign      | d
attnotnull    | f
atthasdef     | t
atthasmissing | t
attidentity   |
attisdropped  | f
attislocal    | t
attinhcount   | 0
attcollation  | 0
attacl        |
attoptions    |
attfdwoptions |
attmissingval | {16:55:40.017082+02}
```

Pour les détails, voir https://brandur.org/postgres-default.

SÉCURITÉ ET INTÉGRITÉ

- Nouveaux rôles
- Vérification d'intégrité

NOUVEAUX RÔLES

- **pg_read_server_files** : permet la lecture de fichier sur le serveur
- **pg_write_server_files** : permet la modification de fichier sur le serveur
- **pg_execute_server_program** : permet l'exécution de fichier sur le serveur
- Rappel : `\COPY` sans limitation depuis le client

PostgreSQL 11 ajoute de nouveaux rôles de sécurité permettant d'affiner les permissions des utilisateurs. Ces nouveaux rôle pourront notamment être utiles pour l'import ou l'export de fichier de données situés **sur le serveur** avec l'ordre `COPY`. (Rappelons que des fichiers situés sur un poste **client** peuvent être chargés depuis psql avec `\COPY`, comme le rappellent les messages d'erreurs ci-dessous.)

Nous voulons charger le fichier `t_read.csv` :

```
$ cat /tmp/t_read.csv
1
2
3
4
5
6
7
8
9
10
```

En version 10 il était nécessaire d'être super-utilisateur pour pouvoir importer les données d'une table depuis un fichier externe.

Création d'un utilisateur standard :

```
postgres@v10=# CREATE USER user_r;
CREATE ROLE
```

Création de la table qui récupérera les données :

```
user_r@v10=> CREATE TABLE t_read(data int);
```

Avec l'utilisateur standard l'import de données depuis un fichier externe retourne l'erreur suivante :

```
user_r@v10=> COPY t_read FROM '/tmp/t_read.csv' CSV ;
ERROR:  must be superuser to COPY to or from a file
HINT : Anyone can COPY to stdout or from stdin. psql's \copy command also works for anyone.
```

En version 11 le rôle `pg_read_server_files` permet à un utilisateur standard d'importer les données depuis un fichier externe.

Création de l'utilisateur `user_r`, membre du rôle `pg_read_server_files` :

```
postgres@v11=# CREATE USER user_r;
CREATE ROLE
postgres@v11=# GRANT pg_read_server_files TO user_r;
GRANT ROLE
```

Import des données depuis un fichier externe CSV :

```
user_r@v11=> CREATE TABLE t_read(data int);
CREATE TABLE

user_r@v11=> COPY t_read FROM '/tmp/t_read.csv' CSV ;
COPY 10
```

Vérification des données sur la table `t_read` :

```
user_r@v11=> select * from t_read;
 data
------
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
```

Par la suite, si l'utilisateur tente d'envoyer les données d'une table vers un fichier externe, le message suivant apparaît :

```
user_r@v11=> COPY t_read TO '/tmp/t_read.csv' CSV ;
ERROR:  must be superuser or a member of the pg_write_server_files role to
        COPY to a file
ASTUCE : Anyone can COPY to stdout or from stdin. psql's \copy command
         also works for anyone.
```

Le rôle `pg_write_server_file` va permettre d'envoyer les données d'une table vers un fichier externe.

Création de l'utilisateur `user_w` membre du rôle `pg_write_server_files` :

```
postgres@v11=# CREATE USER user_w;
CREATE ROLE
postgres@v11=# GRANT pg_write_server_files TO user_w;
GRANT ROLE
```

Création de la table `t_write` (l'utilisateur doit être le propriétaire de la table) :

```
user_w@v11=> CREATE TABLE t_write(data int);
CREATE TABLE
user_w@v11=> INSERT INTO t_write SELECT * from generate_series(1,5);
INSERT 0 5
```

Contenu de la table `t_write` :

```
user_w@v11=> select * from t_write;
 data
------
    1
    2
    3
    4
    5
```

Export des données de la table dans un fichier CSV :

```
user_w@v11=> COPY t_write TO '/tmp/t_write.csv' CSV ;
COPY 10
```

Vérification des données dans le fichier :

```
$ cat /tmp/t_write.csv
1
2
3
4
5
```

VÉRIFICATION D'INTÉGRITÉ

- Nouvelle commande `pg_verify_checksums` (à froid)
- Vérification des sommes de contrôles dans `pg_basebackup`
- Amélioration d'`amcheck`
  - v10 : 2 fonctions de vérification de l'intégrité des index
  - v11 : vérification de la cohérence avec la table (probabiliste)

La commande `pg_verify_checksums` vérifie les sommes de contrôles sur les bases de données à froid. L'instance doit être arrêtée proprement avant le lancement de la commande.

Les sommes de contrôles, si elles sont là, sont à présent vérifiées par défaut sur `pg_basebackup`. En cas de corruption des données, l'opération sera interrompue. Cependant le début de la sauvegarde ne sera pas effacé automatiquement (similaire au comportement de l'option `--no-clean`). Il est possible de désactiver cette vérification avec l'option `--no-verify-checksums`.

Le module `amcheck` était apparu en version 10 pour vérifier la cohérence des index et de leur structure interne, et ainsi détecter des bugs, des corruptions dues au système de fichier voire à la mémoire. Il définit deux fonctions :

- `bt_index_check` est destinée aux vérifications de routine. Elle ne pose qu'un verrou *AccessShareLock* peu gênant.
- `bt_index_parent_check` est plus minutieuse, mais son exécution gêne les modifications dans la table (verrou *ShareLock* sur la table et l'index). Elle ne peut pas être exécutée sur un serveur secondaire.

En v11 apparaît le nouveau paramètre booléen `heapallindex`.

Si ce paramètre vaut `true`, chaque fonction effectue une vérification supplémentaire en recréant temporairement une structure d'index et en la comparant avec l'index original. `bt_index_check` vérifiera que chaque entrée de la table possède une entrée dans l'index. `bt_index_parent_check` vérifiera en plus qu'à chaque entrée de l'index correspond une entrée dans la table.

Les verrous posés par les fonctions ne changent pas. Néanmoins, l'utilisation de ce mode a un impact sur la durée d'exécution des vérifications.
Pour limiter l'impact, l'opération n'a lieu qu'en mémoire, et dans la limite du paramètre `maintenance_work_mem`. Ce paramètre atteint ou dépasse souvent le gigaoctet sur les serveurs récents.
C'est cette restriction mémoire qui implique que la détection de problèmes est probabiliste pour les plus grosses tables (selon la documentation, la probabilité de rater une

incohérence est de 2 % si l'on peut consacrer 2 octets de mémoire à chaque ligne). Mais rien n'empêche de relancer les vérifications régulièrement, diminuant ainsi les chances de rater une erreur.

`amcheck` ne fournit aucun moyen de corriger une erreur, puisqu'il détecte des choses qui ne devraient jamais arriver. `REINDEX` sera souvent la solution la plus simple et facile, mais tout dépend de la cause du problème.

Soit `unetable_pkey`, un index de 10 Go sur un entier :

```
v11=# CREATE EXTENSION amcheck ;
CREATE EXTENSION

v11=# SELECT bt_index_check('unetable_pkey');
Durée : 63753,257 ms (01:03,753)

v11=# SELECT bt_index_check('unetable_pkey', true);
Durée : 234200,678 ms (03:54,201)
```

Ici, la vérification exhaustive multiplie le temps de vérification par un facteur 4.

SQL ET PL/PGSQL

- Index couvrants
- Objets `PROCEDURE`
- Contrôle transactionnel en PL
- JSON
- PL/pgSQL
- Fonctions de fenêtrage
- Autres nouveautés

INDEX COUVRANTS

- Déclaration grâce au mot clé `INCLUDE`
- Uniquement pour les index B-Tree
- Permet des *Index Only Scan* en complétant des index uniques

Cette nouvelle fonctionnalité permet d'inclure des colonnes d'une table uniquement dans les feuilles d'un index de type B-Tree.

L'index ne pourra pas être utilisé pour faire des recherches sur ces colonnes incluses. L'index sera cependant utilisable pour récupérer directement les informations de ces colonnes incluses sans avoir besoin d'accéder à la table grâce à un `Index Only Scan`.

La déclaration se fait par le mot clé `INCLUDE` à la fin de la déclaration de l'index :

```
CREATE INDEX index_couvrant ON ma_table
  (lookup_col1, lookup_col2) INCLUDE (autre_col);
```

La version 9.2 de PostgreSQL a apporté `Index Only Scan`. Si l'information est présente dans l'index, il n'est alors pas nécessaire de lire la table pour récupérer les données recherchées : on les lit directement dans l'index pour des gains substantiels de performance ! Mais pour que ce nœud s'active, il faut évidemment que toutes les colonnes recherchées soient présentes dans l'index.

Une colonne sur laquelle aucune recherche n'est faite mais dont on a besoin dans la requête peut être ajoutée à la fin de la liste des colonnes indexées. La requête pourra alors utiliser un `Index Only Scan`.

Dans un index couvrant, le nouveau mot clé `INCLUDE` permet de ne pas l'ajouter à la liste des colonnes indexées, mais en plus de ces colonnes.

Les colonnes incluses ne sont pas triées et ne peuvent donc pas directement servir aux tris et recherches.

Les index PostgreSQL étant des objets distincts des tables, ajouter des colonnes dans un index duplique de l'information. Cela a un impact en terme de volume sur disque mais également en terme de performance d'insertion et de mise à jour de la table. L'intérêt premier des index couvrants est de pouvoir ajouter des colonnes dans un index déjà présent (unique notamment) sans devoir déclarer un index distinct.

En effet, PostgreSQL utilise un index unique pour implémenter une contrainte d'unicité sur une ou un ensemble de colonnes. Si on veut pouvoir accéder par `Index Only Scan` à une de ces colonnes uniques ainsi qu'à une autre colonne, il faut créer un nouvel index. Un index couvrant va permettre de ne pas créer de nouvel index en intégrant l'autre colonne recherchée à l'index unique.

OBJET PROCEDURE

- Conforme à la norme SQL
- Création par `CREATE PROCEDURE`
- Appel avec `CALL`
- Ne retourne rien
- Permet un contrôle transactionnel en PL

Création et appel d'une procédure (ici en pur SQL) :

```
v11=# CREATE TABLE test1 (a int, b text);
CREATE TABLE

v11=# CREATE PROCEDURE insert_data(a integer, b integer)
      LANGUAGE SQL
      AS $$
        INSERT INTO test1 VALUES (a);
        INSERT INTO test1 VALUES (b);
      $$;
CREATE PROCEDURE

v11=# CALL insert_data(1, 2);
CALL

v11=# SELECT * FROM test1;
 a | b
---+---
 1 |
 2 |
(2 lignes)
```

Les objets de type PROCEDURE sont sensiblement les mêmes que les objets de type FUNCTION.

Les différences sont :

- l'appel se fait par le mot clé `CALL` et non `SELECT` ;
- les objets de type PROCEDURE ne peuvent rien retourner ;
- les objets de type PROCEDURE permettent un contrôle transactionnel, ce que ne peuvent pas faire les objets de type FUNCTION.

CONTRÔLE TRANSACTIONNEL EN PL

- Disponible en PL/pgSQL, PL/Perl, PL/Python, PL/Tcl, SPI (C)
- Utilisable :
  - dans des blocs `DO` / `CALL`
  - dans des objets de type PROCEDURE
- Ne fonctionne pas à l'intérieur d'une transaction
- Incompatible avec une clause `EXCEPTION`

Les mots clés sont différents suivants les langages :

- SPI : `SPI_start_transaction()`, `SPI_commit()` et `SPI_rollback()`
- PL/Perl : `spi_commit()` et `spi_rollback()`
- PL/pgSQL : `COMMIT` et `ROLLBACK`
- PL/Python : `plpy.commit` et `plpy.rollback`
- PL/Tcl : `commit` et `rollback`

Voici un exemple avec `COMMIT` ou `ROLLBACK` suivant que le nombre est pair ou impair :

```
v11=# CREATE TABLE test1 (a int) ;
CREATE TABLE

v11=# CREATE OR REPLACE PROCEDURE transaction_test1()
      LANGUAGE plpgsql
      AS $$
      BEGIN
        FOR i IN 0..5 LOOP
            INSERT INTO test1 (a) VALUES (i);
            IF i % 2 = 0 THEN
               COMMIT;
            ELSE
               ROLLBACK;
            END IF;
```

```
        END LOOP;
      END
      $$;
CREATE PROCEDURE

v11=# CALL transaction_test1();
CALL

v11=# SELECT * FROM test1;
 a | b
---+---
 0 |
 2 |
 4 |
 6 |
 8 |
(5 lignes)
```

Noter qu'il n'y a pas de `BEGIN` explicite dans la gestion des transactions.

On ne peut pas imbriquer des transactions :

```
v11=# BEGIN ; CALL transaction_test1() ;
BEGIN
Temps : 0,097 ms
ERROR:  invalid transaction termination
CONTEXTE : PL/pgSQL function transaction_test1() line 6 at COMMIT
```

On ne peut pas utiliser en même temps une clause `EXCEPTION` et le contrôle transactionnel :

```
v11=# DO LANGUAGE plpgsql $$
        BEGIN
          BEGIN
          INSERT INTO test1 (a) VALUES (1);
          COMMIT;
        INSERT INTO test1 (a) VALUES (1/0);
        COMMIT;
        EXCEPTION
        WHEN division_by_zero THEN
           RAISE NOTICE 'caught division_by_zero';
        END;
      END;
      $$;
ERREUR:  cannot commit while a subtransaction is active
CONTEXTE : fonction PL/pgsql inline_code_block, ligne 5 à COMMIT
```

Pour plus de détails, par exemple sur les curseurs : https://www.postgresql.org/docs/

11/static/plpgsql-transactions.html

PL/PGSQL

- Ajout d'une clause `CONSTANT` à une variable
- Contrainte `NOT NULL` à une variable

Déclarer une variable en tant que `CONSTANT` ou `NOT NULL` permettra de supprimer un certain nombre de bugs.

JSON

- Conversion de et vers du type jsonb
  - en SQL : booléen et nombre
  - en PL/Perl : tableau et *hash* (extension `jsonb_plperl`)
  - en PL/Python : `dict` et `list` (extension `jsonb_plpython`)
- Conversion JSON en tsvector pour la *Full text Search*

Conversion depuis et vers le type jsonb

jsonb <=> SQL

Il existe 4 types primitifs en JSON. Voici le tableau de correspondance avec les types PostgreSQL :

Type Primitif JSON	Type PostgreSQL
string	text
number	numeric
boolean	boolean
null	(aucun)

S'il était déjà possible de convertir des données PostgreSQL natives vers le type jsonb, l'inverse n'était possible que vers le type text :

```
v10=# SELECT 'true'::jsonb::boolean;
ERROR:  cannot cast type jsonb to boolean
LIGNE 1 : SELECT 'true'::jsonb::boolean;
```

```
                                ^
v10=# SELECT 'true'::jsonb::text::boolean;
 bool
------
 t
(1 ligne)

v10=# SELECT '3.141592'::jsonb::float;
ERROR:  cannot cast type jsonb to double precision
LIGNE 1 : SELECT '3.141592'::jsonb::float;
                                   ^
v10=# SELECT '3.141592'::jsonb::text::float;
  float8
----------
 3.141592
(1 ligne)
```

Il est dorénavant possible de convertir des données de type jsonb vers les types booléen et numérique :

```
v11=# SELECT 'true'::jsonb::boolean;
 bool
------
 t
(1 ligne)

v11=# SELECT '3.141592'::jsonb::float;
  float8
----------
 3.141592
(1 ligne)
```

jsonb <=> PL/Perl

Une transformation a été ajoutée en PL/Perl pour transformer les champs jsonb en champs natif Perl.

Cette fonctionnalité nécessite l'installation de l'extension `jsonb_plperl`. Celle-ci n'est pas installée par défaut. On doit installer le paquet `postgresql11-plperl-11.0` sur Red Hat/CentOS et le paquet `postgresql-plperl-11` sur Debian/Ubuntu.

Une fois l'extension activée, on précisera la transformation à utiliser pour charger les paramètres avec le mot clé `TRANSFORM` :

```
v11=# CREATE EXTENSION jsonb_plperl CASCADE;
NOTICE:  installing required extension "plperl"
CREATE EXTENSION
```

```
v11=# CREATE OR REPLACE FUNCTION fperl(val jsonb)
  RETURNS jsonb
  TRANSFORM FOR TYPE jsonb
  AS $$
    my $arg = shift;
    elog(NOTICE, "Arg is: [$arg]");
    my $keys_str = "";
    for my $key (keys %$arg) {
      $keys_str .= "'".$key."' "
    }
    elog(NOTICE, "JSON keys are: ".$keys_str);
  $$ LANGUAGE plperl;
CREATE FUNCTION

v11=# SELECT fperl('{"1":1,"example": null}'::jsonb);
NOTICE:  Arg is: [HASH(0x1d7e330)]
NOTICE:  jsonb keys are: '1' 'example'
 fperl
-------

(1 ligne)
```

jsonb <=> PL/Python

Une transformation a été ajoutée en PL/Python pour transformer les champs jsonb en champs natif Python.

Cette fonctionnalité nécessite l'installation de l'extension `jsonb_plpython`. Celle-ci n'est pas installée par défaut. On doit installer le paquet `postgresql11-plpython-11.0` sur Red Hat/CentOS. Sur Debian/Ubuntu_ on pourra installer l'extension en version 2 et/ou 3 de Python en utilisant les paquets `postgresql-plpython-11` et `postgresql-plpython3-11`.

Une fois l'extension activée, on précisera la transformation à utiliser pour charger les paramètres avec le mot clé `TRANSFORM` :

```
v11=# CREATE EXTENSION jsonb_plpythonu CASCADE;
NOTICE:  installing required extension "plperl"
CREATE EXTENSION

v11=# CREATE OR REPLACE FUNCTION fpython(val jsonb)
  RETURNS jsonb
  TRANSFORM FOR TYPE jsonb
  AS $$
    plpy.info(val)
    keys_str = ""
    for key in val:
```

```
      keys_str += "'"+key+"' "
    plpy.info("JSON keys are: " + keys_str)
  $$ LANGUAGE plpythonu;
CREATE FUNCTION

v11=# SELECT fpython('{"1":1,"example": null}'::jsonb);
INFO:  {'1': Decimal('1'), 'example': None}
INFO:  JSON keys are: '1' 'example'
 fpython
---------

(1 ligne)
```

JSON en tsvector pour la *Full Text Search*

La conversion en tsvector permet la recherche plein texte. Couplée à une indexation adéquate, GIN ou GiST, la *Full Text Search* offre de nombreuses fonctionnalités et des performances impressionnantes.

Jusqu'à maintenant, les champs JSON était analysés comme des textes, sans tenir compte de la sémantique. La nouvelle fonction `jsonb_to_tsvector` permet d'extraire des informations ciblées issues de champs JSON choisis.
La fonction prend en premier paramètre la langue et en deuxième paramètre la structure JSON à analyser. Le troisième paramètre permet de choisir les valeur à filter :

- *string* : les chaines de caractères,
- *numeric* : les valeur numérique,
- *boolean* : les booléen (`true` et `false`),
- *key* : pour inclure toutes les clés de la structure JSON,
- *all* : pour inclure tous les champs ci-dessus.

Voici ce que donnait la fonction `to_tsvector` :

```
v11=# select to_tsvector('french',
    '{ "a": "Vive la v11 !",
       "b": 5432,
       "c" : { "1": 42, "2": "question", "3": true } }'::jsonb);
        to_tsvector
------------------------------
 'question':5 'v11':3 'viv':1
(1 ligne)
```

En choisissant l'option de filtre `string`, on obtient le même résultat :

```
v11=# select jsonb_to_tsvector('french',
    '{ "a": "Vive la v11 !",
       "b": 5432,
```

```
       "c" : { "1": 42, "2": "question", "3": true } }'::jsonb, '["string"]');
      jsonb_to_tsvector
------------------------------
 'question':5 'v11':3 'viv':1
(1 ligne)
```

La nouvelle fonction donne cependant accès à de nombreux autres modes :

```
v11=# select jsonb_to_tsvector('french',
    '{ "a": "Vive la v11 !",
       "b": 5432,
       "c" : { "1": 42, "2": "question", "3": true } }'::jsonb,
     '["numeric", "boolean"]');
    jsonb_to_tsvector
------------------------
 '42':3 '5432':1 'tru':5
(1 ligne)

v11=# select jsonb_to_tsvector('french',
    '{ "a": "Vive la v11 !",
       "b": 5432,
       "c" : { "1": 42, "2": "question", "3": true } }'::jsonb,
     '["key"]');
       jsonb_to_tsvector
--------------------------------
 '1':6 '2':8 '3':10 'a':1 'b':3
(1 ligne)

v11=# select jsonb_to_tsvector('french',
    '{ "a": "Vive la v11 !",
       "b": 5432,
       "c" : { "1": 42, "2": "question", "3": true } }'::jsonb,
     '["all"]');
                jsonb_to_tsvector
-------------------------------------------------------
 '1':12 '2':16 '3':20 '42':14 '5432':9 'a':1 'b':7 \
 'question':18 'tru':22 'v11':5 'viv':3
(1 ligne)
```

FONCTIONS DE FENÊTRAGE

- Finalisation du support de la norme **SQL:2011**
  `{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]`
  `{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]`
- avec exclusion :
  `EXCLUDE {CURRENT ROW|GROUP|TIES|NO OTHERS}`

Le support de la syntaxe `RANGE`, incluant une clause `EXCLUDE {CURRENT ROW|GROUP|TIES|NO OTHERS}` est la fin d'un travail entamé depuis PostgreSQL 9.4 pour supporter les fonctions de fenêtrage.

Pour plus d'information et des exemples, voir l'explication de depesz[20] .

AUTRES NOUVEAUTÉS

- `ANALYSE` et `VACUUM` tables multiples
- `LOCK TABLE view`
- Définir le seuil de conversion en *TOAST* :
  `CREATE TABLE ... WITH (toast_tuple_target = N)`

On peut à présent passer plusieurs tables en paramètre à `ANALYZE`, `VACUUM` ou `VACUUM FULL` :

```
VACUUM VERBOSE t1, t2 ;
```

Poser un `LOCK TABLE` sur une vue pose un verrou sur les différentes tables impliquées dans cette vue. L'utilisateur n'a besoin des droits que sur la vue et pas les tables (si le propriétaire de la vue a ces droits sur les tables.)

Par défaut, les tables *TOAST* stockent les valeurs de plus de 2 ko dans une table *TOAST* séparée, et les compressent. Cela est transparent pour l'utilisateur. La version 11 permet de fixer une autre limite entre 128 et 8160 octets. L'utilité est ponctuelle, le défaut étant proche de l'optimal.

---

[20] https://www.depesz.com/2018/02/13/waiting-for-postgresql-11-support-all-sql2011-options-for-window-frame-clauses/

OUTILS

- psql
- initdb
- pg_dump et pg_dumpall
- pg_basebackup
- pg_rewind

PSQL

- SELECT ... FROM ... \gdesc
  - ou \gdesc seul après exécution
  - retourne le type des colonnes sans exécution
- Variables de suivi des erreurs de requêtes
  - ERROR, SQLSTATE et ROW_COUNT
- exit et quit à la place de \q pour quitter psql
- fonctionnalités psql, donc utilisables sur des instances < 11

PostgreSQL 11 apporte quelques améliorations notables au niveau des commandes psql.

La commande \gdesc retourne le nom et le type des colonnes de la dernière requête exécutée.

```
v11=# select * from t_write;
 c1
----
  1
  2
  3
  4
  5
(5 rows)

v11=# \gdesc
 Column |  Type
--------+---------
 c1     | integer
(1 row)
```

On peut aussi tester les types retournés par une requête sans l'exécuter :

```
v11=# select 3.0/2 as ratio, now() as maintenant \gdesc
   Column   |           Type
```

```
------------+--------------------------
 ratio      | numeric
 maintenant | timestamp with time zone
```

Les variables ERROR, SQLSTATE et ROW_COUNT permettent de suivre l'état de la dernière requête exécutée.

```
v11=# \d t1
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 c1     | integer |           |          |

v11=# select c2 from t1;
ERROR:  column "c2" does not exist
```

La variable ERROR renvoie une valeur booléenne précisant si la dernière requête exécutée a bien reçu un message d'erreur :

```
v11=# \echo :ERROR
true
```

La variable SQLSTATE retourne le code de l'erreur ou 00000 s'il n'y a pas d'erreur :

```
v11=# \echo :SQLSTATE
42703
```

La variable ROW_COUNT renvoie le nombre de lignes retournées lors de l'exécution de la dernière requête :

```
v11=# \echo :ROW_COUNT
0
```

Il existe aussi les variables LAST_ERROR_MESSAGE et LAST_ERROR_SQLSTATE qui renvoient le dernier message d'erreur retourné et le code de la dernière erreur.

```
v11=# \echo :LAST_ERROR_MESSAGE
column "c2" does not exist

v11=# \echo :LAST_ERROR_SQLSTATE
42703
```

Les commandes exit et quit ont été ajoutées pour quitter psql afin que cela soit plus intuitif pour les nouveaux utilisateurs.

Toutes ces fonctionnalités sont liées à l'outil client psql, donc peuvent être utilisées même si le serveur reste dans une version antérieure.

INITDB

- option `--wal-segsize` :
  - spécifie la taille des fichier WAL à l'initialisation (1 Mo à 1 Go)
- option `--allow-group-access` :
  - Droits de lecture et d'exécution au groupe auquel appartient l'utilisateur initialisant l'instance.
  - Droit sur les fichiers : `drwxr-x---`

L'option `--wal-segsize` permet de spécifier la taille des fichiers WAL lors de l'initialisation de l'instance (et uniquement à ce moment). Toujours par défaut à 16 Mo, ils peuvent à présent aller de 1 Mo à 1 Go.

Cela permet d'ajuster la taille en fonction de l'activité, principalement pour les instances générant de très nombreux journaux, surtout s'il faut les archiver. Des journaux plus gros et moins nombreux seront alors plus efficaces. Par contre, si les journaux sont trop gros par rapport à l'activité, ils ne seront pas archivés assez souvent. Le défaut reste à 16 Mo.

Exemple pour des WAL de 1 Go :

```
initdb -D /var/lib/postgresql/11/workshop --wal-segsize=1024
```

L'option `--allow-group-access` autorise les droits de lecture et d'exécution au groupe auquel appartient l'utilisateur initialisant l'instance. Droit sur les fichiers : `drwxr-x---`. Cela peut servir pour ne donner que des droits de lecture à un outil de sauvegarde.

SAUVEGARDE ET RESTAURATION

- `pg_dumpall`
  - option `--encoding` pour spécifier l'encodage de sortie
  - `-g` ne sort plus les permissions et les configurations de variables
  - Ajouter `--create` à `pg_dump -Fp` ou `pg_restore` pour cela !
  - Révisez vos scripts !
- `pg_dump --load-via-partition-root` : partitions en bloc
- `pg_basebackup`
  - option `--create-slot` pour créer un slot de réplication permanent

`pg_dumpall` bénéficie d'une nouvelle option`--encoding` permettant de spécifier l'encodage de sortie d'un dump (utile notamment sur Windows).

Les permissions par `GRANT` et `REVOKE` sur une base de données et les configurations de variables par `ALTER DATABASE SET` et `ALTER ROLE IN DATABASE SET` sont à présent gérées

par `pg_dump` et `pg_restore` et non plus par `pg_dumpall`. `pg_dump -Fp` (format texte) et un `pg_restore` n'appliqueront ces modifications qu'avec l'option `--create`. Vérifiez vos scripts de restauration !

Avec l'option `--load-via-partition-root`, une table partitionnée est exportée en bloc et non partition par partition. Cela permet de réimporter une table partitionnée dans une table avec un partitionnement différent, ou sans partitionnement. Par défaut, sans cette option, les partitions sont exportées séparément, et, à la restauration, réimportées séparément, puis rattachées à la table partitionnée. Cela permet de paralléliser l'export et l'import.

Une nouvelle option `--no-comment` permet aussi de supprimer les commentaires.

Une nouvelle option `--create-slot` est disponible dans `pg_basebackup` permettant de créer directement un slot de réplication. Elle doit donc être utilisée en complément de l'option `--slot`. Le slot de réplication est conservé après la fin de la sauvegarde et peut être celui que le serveur secondaire utilisera par la suite, réduisant ainsi les manipulations. Si le slot de réplication existe déjà, la commande `pg_basebackup` s'interrompt et affiche un message d'erreur.

PG_REWIND

- `pg_rewind` : optimisations de fichiers inutiles
- interdit en tant que root
- possible avec un accès non-superuser sur le maître

`pg_rewind` est un outil permettant de reconstruire une instance secondaire qui a « décroché » sans la reconstruire complètement, à partir d'un primaire.

Quelques fichiers inutiles sont à présent ignorés. La sécurité pour certains environnements a été améliorée en interdisant le fonctionnement du binaire sous root, et en permettant au besoin de n'utiliser qu'un utilisateur « normal » sur le serveur primaire (voir le blog de Michael Paquier[21]).

---

[21] https://paquier.xyz/postgresql-2/postgres-11-superuser-rewind/

PG_PREWARM

- `pg_prewarm` : chargement de données en cache (*shared buffers* ou OS)
- En v11 :
  - mémorisation régulière des blocs dans les *shared buffers*
  - chargement automatique de ces blocs au démarrage

`pg_prewarm` est un module permettant de charger des tables (ou un index, ou une partie de table) en mémoire cache (les *shared buffers* ou le cache de l'OS). Il permettait jusqu'à maintenant de charger des relations dans le cache via la fonction `pg_prewarm`, donc de façon manuelle uniquement.

Une nouvelle fonctionnalité de la version 11 permet de sauvegarder périodiquement les blocs dans le cache de PostgreSQL. Cette sauvegarde peut être effectuée de façon régulière, toutes les 5 minutes par défaut. Elle sera effectuée de toute façon lors d'un arrêt normal de l'instance.

Grâce à cette sauvegarde, il est désormais possible d'automatiser le chargement de la dernière sauvegarde au démarrage de l'instance. La mise en place s'opère dans le fichier `postgresql.conf` :

```
shared_preload_libraries = 'pg_prewarm'
pg_prewarm.autoprewarm = true
```

On peut ainsi éviter que des requêtes soient ralenties parce que les données ne sont pas encore chargées en mémoire, notamment en cas de redémarrage.

Le paramètre `pg_prewarm.autoprewarm_interval`, exprimé en secondes, permet de préciser le rythme des sauvegardes. Les sauvegardes seront stockées dans le fichier `$PGDATA/autoprewarm.blocks`.

Deux nouvelles fonctions font leur apparition. Elles sont surtout utiles si le préchauffage n'est pas activé :

- `autoprewarm_start_worker()` : permet de lancer le processus de sauvegarde automatique des blocs du *shared buffers*, le *autoprewarm worker*,
- `autoprewarm_dump_now()` : permet de procéder immédiatement à la sauvegarde.

Le préchauffage du cache est typiquement plus utile au démarrage, quand les caches sont majoritairement vides. Il n'est cependant pas garanti que les données chargées soient utiles aux requêtes et qu'elles restent dans le cache si la base est active et manipule de grands volumes de données.

Documentation officielle : https://docs.postgresql.fr/11/pgprewarm.html

RÉPLICATION

- Réplication logique
- Taille des WALs et checkpoint

RÉPLICATION LOGIQUE

- Réplication de l'ordre `TRUNCATE`
- Réduction de l'empreinte mémoire
- Migration majeure par réplication logique

La version 11 lève une des contraintes les plus gênantes de la réplication logique apparue en version 10 : les ordres `TRUNCATE` sont à présent dupliqués.

La documentation précise que la réplication de l'ordre peut échouer si des clés étrangères ont été ajoutées vers cette table. Cela est cohérent avec le principe de la réplication logique : les données et schémas répliquées sont modifiables, et la base cible impose la cohérence de ses données au dépend des données source au besoin.

La gestion de la mémoire a été améliorée grâce à un nouvel allocateur mémoire en mode FIFO idéal pour ce besoin[22] .

Enfin, les premières migrations majeures utilisant la réplication logique sans outil tiers pourront avoir lieu entre des instances en versions 10 et 11 (voir le TP).

WAL ET CHECKPOINT

- Suppression du second checkpoint

Un checkpoint est un « point de vérification » au cours duquel les fichiers de données sont mis à jour pour refléter les informations des journaux de transactions.

Jusqu'en version 10, les fichiers de journaux de transactions étaient conservés le temps de faire 2 checkpoints. Les journaux précédents le premier checkpoint étaient alors recyclés. L'intérêt d'avoir deux checkpoints était de permettre de pouvoir revenir au précédent checkpoint au cas où le dernier soit introuvable ou illisible.

[22] https://commitfest.postgresql.org/14/1239/

Il a été décidé qu'il n'était finalement pas nécessaire de conserver ce second checkpoint et que cela pouvait même être plus dangereux qu'utile[23] . La suppression de ce second checkpoint permet aussi de simplifier un peu le code.

En conséquence, à `max_wal_size` égal, on va ainsi réduire d'environ 33% la fréquence des checkpoints, et on augmentera le temps pour terminer la récupération après un crash.

Michael Paquier a écrit un petit article sur le sujet[24] .

LES OUTILS DE LA SPHÈRE DALIBO

Outil	Compatibilité avec PostgreSQL 11
pitrery	Oui
ldap2pg	Oui
pgBadger	Oui
pgCluu	Oui
ora2Pg	Oui
powa-archivist	oui

Voici une grille de compatibilité des outils au 1er octobre 2018 :

Outil	Compatibilité avec PostgreSQL 11
pg_back	Oui
pitrery	Oui, depuis la version 2.2
ldap2pg	Oui, depuis la version 4.14
pgBadger	Oui
pgCluu	Oui
ora2Pg	Oui, support du partitionnement par *HASH* en 19.1
powa-archivist	oui, depuis la version 3.1.2
pg_qualstats	oui, depuis la version 1.0.5
pg_stat_kcache	oui, depuis la version 2.1.0
hypopg	oui depuis la version 1.1.2
pg_activity	En cours de développement
check_pgactivity	En cours de développement, partiellement compatible.
PAF	En cours de développement
temboard	En cours de développement

[23] https://www.postgresql.org/message-id/flat/20160201235854.GO8743%40awork2.anarazel.de
[24] https://paquier.xyz/postgresql-2/postgres-11-secondary-checkpoint/

FUTUR

- Développement de la version 12 entamé durant l'été 2018
- Déjà présent ou à venir, **sans garantie** :
  - Amélioration du partitionnement
  - Amélioration du parallélisme
  - Amélioration du JIT
  - Index couvrants sur GiST
  - Clause `SQL MERGE`
  - Filtrage des lignes pour la réplication logique
  - Support de GnuTLS
  - `ANALYZE nom_index`
  - Pluggable Storage API : alternatives à MVCC ?
  - …

La roadmap[25] du projet détaille les prochaines grandes étapes.

Les *commit fests* nous laissent entrevoir une continuité dans l'évolution des thèmes principaux suivants : parallélisme, partitionnement et JIT.

Un bon nombre de commits ont déjà eu lieu. Vous pouvez consulter l'ensemble des modifications validées (ou reportées…) pour chaque commit fest :

- juillet 2018[26]
- septembre 2018[27]
- novembre 2018[28]
- janvier 2019[29]
- mars 2019[30]

Quelques sources :

- clause SQL MERGE[31]
- support de GnuTLS[32]
- filtrage des ligne pour la réplication logique[33]
- le moteur zheap comme alternative à MVCC[34]

[25] https://dali.bo/pg-roadmap
[26] https://commitfest.postgresql.org/18/?status=4
[27] https://commitfest.postgresql.org/19/?status=4
[28] https://commitfest.postgresql.org/20/?status=4
[29] https://commitfest.postgresql.org/21/?status=4
[30] https://commitfest.postgresql.org/22/?status=4
[31] https://commitfest.postgresql.org/19/1446/
[32] https://commitfest.postgresql.org/19/1277/
[33] https://commitfest.postgresql.org/19/1710/
[34] https://www.slideshare.net/EnterpriseDB/postgres-vision-2018-the-promise-of-zheap

> Tout cela est encore en développement et test. Rien ne garantit que ces améliorations seront présentes dans la version finale de PostgreSQL 12 : si elles ne sont pas prêtes, elles seront rejetées ou repoussées.

QUESTIONS

```
SELECT * FROM questions;
```

ATELIER

À présent, place à l'atelier…

- Installation
- Mise à jour d'une partition avec un `UPDATE`
- Manipulation du partitionnement par hachage
- `TRUNCATE` avec la réplication logique
- Mise à jour PostgreSQL 10 vers 11 avec la réplication logique
- Index couvrants
- Parallélisation
- Sauvegarde des droits avec `pg_dump`
- l'extension pg_prewarm
- Test du JIT

INSTALLATION

Les machines de la salle de formation utilisent CentOS 6. L'utilisateur dalibo peut utiliser sudo pour les opérations système.

Le site postgresql.org propose son propre dépôt RPM, nous allons donc l'utiliser pour installer PostgreSQL 11.

On commence par installer le RPM du dépôt `pgdg-centos11-11-2.noarch.rpm` depuis https://yum.postgresql.org/:

```
# pgdg_yum_11=https://download.postgresql.org/pub/repos/yum
# pgdg_yum_11+=/11/redhat/rhel-6-x86_64/pgdg-centos11-11-2.noarch.rpm
# yum install -y $pgdg_yum_11
Installed:
  pgdg-centos11.noarch 0:11-2

# yum install -y postgresql11 postgresql11-contrib postgresql11-server

Installed:
  postgresql11.x86_64 0:11.0-beta2_1PGDG.rhel6
  postgresql11-contrib.x86_64 0:11.0-beta2_1PGDG.rhel6
  postgresql11-server.x86_64 0:11.0-beta2_1PGDG.rhel6
Dependency Installed:
  libicu.x86_64 0:4.2.1-14.el6
  libxslt.x86_64 0:1.1.26-2.el6_3.1
  postgresql11-libs.x86_64 0:11.0-beta2_1PGDG.rhel6
```

On peut ensuite initialiser une instance :

```
# service postgresql-11 initdb
Initializing database:                                     [  OK  ]
```

Enfin, on démarre l'instance, car ce n'est par défaut pas automatique sous Red Hat et CentOS :

```
# service postgresql-11 start
Starting postgresql-11 service:                            [  OK  ]
```

Pour se connecter à l'instance sans modifier `pg_hba.conf` :

```
# sudo -iu postgres /usr/pgsql-11/bin/psql
```

Enfin, on vérifie la version :

```
postgres=# select version();
                          version
------------------------------------------------------------------
 PostgreSQL 11beta2 on x86_64-pc-linux-gnu,
   compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-18), 64-bit
```

On répète ensuite le processus d'installation de façon à installer PostgreSQL 10 aux côtés de PostgreSQL 11.

Le RPM du dépôt est `pgdg-centos10-10-2.noarch.rpm` :

```
# pgdg_yum_10=https://download.postgresql.org/pub/repos/yum
# pgdg_yum_10+=/10/redhat/rhel-6.9-x86_64/pgdg-centos10-10-2.noarch.rpm
# yum install -y $pgdg_yum_10

Installed:
  pgdg-centos10.noarch 0:10-2


# yum install -y postgresql10 postgresql10-contrib postgresql10-server
Installed:
  postgresql10.x86_64 0:10.4-1PGDG.rhel6
  postgresql10-contrib.x86_64 0:10.4-1PGDG.rhel6
  postgresql10-server.x86_64 0:10.4-1PGDG.rhel6

Dependency Installed:
  postgresql10-libs.x86_64 0:10.4-1PGDG.rhel6

# service postgresql-10 initdb
Initializing database:                                     [  OK  ]

# sed -i "s/#port = 5432/port = 5433/" \
  /var/lib/pgsql/10/data/postgresql.conf

# service postgresql-10 start
Starting postgresql-10 service:                            [  OK  ]

# sudo -iu postgres /usr/pgsql-10/bin/psql -p 5433
```

Dans cet atelier, les différentes sorties des commandes `psql` utilisent :

```
\pset columns 80
\pset format wrapped
```

MISE À JOUR D'UNE CLÉ DE PARTITION AVEC UPDATE

La table partitionné est créé sur les deux instances en version 10 et 11.

- Création d'une table partitionné par intervalle :

```
CREATE TABLE liste_dates (d timestamptz) PARTITION BY RANGE(d);
```

- Création des partitions :

```
CREATE TABLE liste_dates_a PARTITION OF liste_dates
       FOR VALUES FROM ('2018-01-01') TO ('2018-04-01');
CREATE TABLE liste_dates_b PARTITION OF liste_dates
       FOR VALUES FROM ('2018-04-01') to ('2018-07-01');
CREATE TABLE liste_dates_c partition of liste_dates
       FOR VALUES FROM ('2018-07-01') to ('2018-10-01');
CREATE TABLE liste_dates_d partition of liste_dates
       FOR VALUES FROM ('2018-10-01') to ('2018-12-31');
```

- Insertion de données dans les partitions :

```
INSERT INTO liste_dates VALUES ('2018-01-15');
INSERT INTO liste_dates VALUES ('2018-02-10');
INSERT INTO liste_dates VALUES ('2018-03-12');
INSERT INTO liste_dates VALUES ('2018-05-25');
INSERT INTO liste_dates VALUES ('2018-06-02');
INSERT INTO liste_dates VALUES ('2018-08-12');
INSERT INTO liste_dates VALUES ('2018-10-20');
INSERT INTO liste_dates VALUES ('2018-11-30');
INSERT INTO liste_dates VALUES ('2018-12-19');
```

- Vérification du contenu des tables sur les deux instances :

```
=# SELECT * FROM liste_dates ;
           d
------------------------
 2018-01-15 00:00:00+01
 2018-02-10 00:00:00+01
 2018-03-12 00:00:00+01
 2018-05-25 00:00:00+02
 2018-06-02 00:00:00+02
 2018-08-12 00:00:00+02
 2018-10-20 00:00:00+02
 2018-11-30 00:00:00+01
 2018-12-19 00:00:00+01
(9 lignes)

=# SELECT * FROM liste_dates_a;
          d
------------------------
```

```
 2018-01-15 00:00:00+01
 2018-02-10 00:00:00+01
 2018-03-12 00:00:00+01

=# SELECT * FROM liste_dates_b;
          d
------------------------
 2018-05-25 00:00:00+02
 2018-06-02 00:00:00+02

=# SELECT * FROM liste_dates_c;
          d
------------------------
 2018-08-12 00:00:00+02

=# SELECT * FROM liste_dates_d;
           d
------------------------
 2018-10-20 00:00:00+02
 2018-11-30 00:00:00+01
 2018-12-19 00:00:00+01
```

UPDATE EN VERSION 10

En version 10, la mise à jour avec `UPDATE` retourne une erreur :

```
v10=# UPDATE liste_dates SET d='2018-09-22' WHERE d='2018-01-15';
ERROR:  new row for relation "liste_dates_a" violates partition constraint
DÉTAIL : Failing row contains (2018-09-22 00:00:00).
```

L'opération fonctionnera seulement si la donnée mise à jour se trouve sur la même partition :

```
v10=# UPDATE liste_dates set d='2018-09-22' WHERE d='2018-08-12';
UPDATE 1
v10=# SELECT * FROM liste_dates_c;
           d
------------------------
 2018-09-22 00:00:00+02
```

Si la donnée mise à jour doit se retrouver dans une autre partition, il est nécessaire de supprimer la donnée de l'ancienne partition et d'insérer la donnée souhaiter dans la nouvelle partition.

```
v10=# DELETE FROM liste_dates_a WHERE d='2018-03-12';
DELETE 1
v10=# SELECT * FROM liste_dates_a;
```

```
           d
------------------------
 2018-01-15 00:00:00+01
 2018-02-10 00:00:00+01

v10=# INSERT INTO liste_dates values ('2018-07-14');
INSERT 0 1

v10=# SELECT * FROM liste_dates ;
           d
------------------------
 2018-01-15 00:00:00+01
 2018-02-10 00:00:00+01
 2018-05-25 00:00:00+01
 2018-06-02 00:00:00+02
 2018-09-22 00:00:00+02
 2018-07-14 00:00:00+02
 2018-10-20 00:00:00+02
 2018-11-30 00:00:00+01
 2018-12-19 00:00:00+01

v10=# SELECT * FROM liste_dates_a;
           d
------------------------
 2018-01-15 00:00:00+01
 2018-02-10 00:00:00+01

v10=# SELECT * FROM liste_dates_c;
           d
------------------------
 2018-09-22 00:00:00+02
 2018-07-14 00:00:00+02
```

UPDATE EN VERSION 11

La mise à jour avec `UPDATE` fonctionne :

```
v11=# UPDATE liste_dates SET d='2018-09-22' WHERE d='2018-01-15';
UPDATE 1
```

Les données sont automatiquement redirigées vers les bonnes partitions :

```
v11=# SELECT * FROM liste_dates ;
           d
------------------------
 2018-02-10 00:00:00+01
 2018-03-12 00:00:00+01
```

```
 2018-05-25 00:00:00+02
 2018-06-02 00:00:00+02
 2018-08-12 00:00:00+02
 2018-09-22 00:00:00+02
 2018-10-20 00:00:00+02
 2018-11-30 00:00:00+01
 2018-12-19 00:00:00+01
(9 lignes)

v11=# SELECT * FROM liste_dates_a ;
           d
------------------------
 2018-02-10 00:00:00+01
 2018-03-12 00:00:00+01

v11=# SELECT * FROM liste_dates_c ;
           d
------------------------
 2018-08-12 00:00:00+02
 2018-09-22 00:00:00+02
```

PARTITIONNEMENT PAR HACHAGE

Nous allons manipuler deux tables contenant les mêmes information : une table non partitionnée et une table partitionnée par hachage. Nous allons comparer les plans d'exécution et les performances entre ces 2 tables.

Les performances vont être très dépendantes de l'infrastructure (disque, CPU), du type de données et du nombre de partitions. Si vous souhaitez utiliser les tables partitionnées par hachage, il est important de tester l'impact sur chaque type d'opération.

Créons les tables `commandes_normale` et `commandes` :

```
CREATE TABLE commandes_normale (
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  date_commande timestamp DEFAULT now(),
  c1 integer, c2 text
  );

CREATE TABLE commandes (
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  date_commande timestamp DEFAULT now(),
  c1 integer, c2 text
  ) PARTITION BY HASH (id);
```

Si nous essayons dès maintenant d'insérer des données dans la table partitionnée, nous obtenons l'erreur suivante :

```
v11=# INSERT INTO commandes (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
ERROR:  no partition of relation "commandes" found for row
DÉTAIL : Partition key of the failing row contains (id) = (1).
```

Nous n'avons pas encore fixé le nombre de partitions. Fixons-le à 5 et créons toutes les partitions :

```
CREATE TABLE commandes_0_5 PARTITION OF commandes
  FOR VALUES WITH (modulus 5,remainder 0);
CREATE TABLE commandes_1_5 PARTITION OF commandes
  FOR VALUES WITH (modulus 5,remainder 1);
CREATE TABLE commandes_2_5 PARTITION OF commandes
  FOR VALUES WITH (modulus 5,remainder 2);
CREATE TABLE commandes_3_5 PARTITION OF commandes
  FOR VALUES WITH (modulus 5,remainder 3);
CREATE TABLE commandes_4_5 PARTITION OF commandes
  FOR VALUES WITH (modulus 5,remainder 4);
```

Fixons certains paramètres :

```
SET jit TO off;
SET max_parallel_workers TO 0;
\timing on
```

Nous allons maintenant pouvoir comparer les performances en insertion :

```
INSERT INTO commandes_normale (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

INSERT INTO commandes (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
```

Insérons d'autres lignes, pour un total de 3 millions par table :

```
INSERT INTO commandes_normale (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT INTO commandes (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT INTO commandes_normale (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT INTO commandes (c1, c2)
  SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
```

Remarquons que les tailles des partitions sont quasi identiques :

```
v11=# \d+
                                  Liste des relations
 Schéma |           Nom            |   Type   | Propriétaire |   Taille   |
--------+--------------------------+----------+--------------+------------+
 public | commandes                | table    | postgres     | 0 bytes    |
 public | commandes_0_5            | table    | postgres     | 39 MB      |
 public | commandes_1_5            | table    | postgres     | 39 MB      |
 public | commandes_2_5            | table    | postgres     | 39 MB      |
 public | commandes_3_5            | table    | postgres     | 39 MB      |
 public | commandes_4_5            | table    | postgres     | 39 MB      |
 public | commandes_id_seq         | séquence | postgres     | 8192 bytes |
 public | commandes_normale        | table    | postgres     | 193 MB     |
 public | commandes_normale_id_seq | séquence | postgres     | 8192 bytes |
```

Le nombre de lignes dans chaque partition n'est cependant pas strictement égal :

```
v11=# SELECT count(*) com1 FROM commandes_0_5;
 count
--------
 600337
(1 ligne)

v11=# SELECT count(*) com1 FROM commandes_1_5;
 count
--------
```

```
 600316
(1 ligne)
```

Testons ensuite les performances en mise à jour en mettant à jour 15 % des lignes :

```
UPDATE commandes SET
  date_commande=now(),c1=c1+1000000,c2='Ligne '||c1+1000000
  WHERE random()>0.85;

UPDATE commandes_normale SET
  date_commande=now(),c1=c1+1000000,c2= 'Ligne '||c1+1000000
  WHERE random()>0.85;
```

Effaçons 15 % des lignes :

```
DELETE FROM commandes WHERE random()>0.85;

DELETE FROM commandes_normale WHERE random()>0.85;
```

Regardons les plans d'exécution des requêtes précédentes sur la table partitionnée :

```
v11=# EXPLAIN (costs OFF) UPDATE commandes SET
  date_commande=now(),c1=c1+1000000,c2='Ligne '||c1+1000000
  WHERE random()>0.85;
                      QUERY PLAN
------------------------------------------------------
 Update on commandes
   Update on commandes_0_5
   Update on commandes_1_5
   Update on commandes_2_5
   Update on commandes_3_5
   Update on commandes_4_5
   ->  Seq Scan on commandes_0_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_1_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_2_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_3_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_4_5
         Filter: (random() > '0.85'::double precision)
(16 lignes)

v11=# EXPLAIN (costs OFF) DELETE FROM commandes WHERE random()>0.85;
                      QUERY PLAN
------------------------------------------------------
 Delete on commandes
   Delete on commandes_0_5
```

```
   Delete on commandes_1_5
   Delete on commandes_2_5
   Delete on commandes_3_5
   Delete on commandes_4_5
   ->  Seq Scan on commandes_0_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_1_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_2_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_3_5
         Filter: (random() > '0.85'::double precision)
   ->  Seq Scan on commandes_4_5
         Filter: (random() > '0.85'::double precision)
(16 lignes)
```

Ici, nous agissons sur toutes les lignes, il ne peut y avoir d'élagage de partition. Cette fonctionnalité est cependant disponible pour les tables partitionnées par hachage :

```
v11=# EXPLAIN (costs off) SELECT * FROM commandes WHERE id=400;
                        QUERY PLAN
-----------------------------------------------------------
 Append
   ->  Index Scan using commandes_3_5_pkey on commandes_3_5
         Index Cond: (id = 400)
(3 lignes)
```

Testons les performances du *VACUUM* :

```
VACUUM commandes;

VACUUM commandes_normale;
```

L'avantage des tables partitionnées est que l'on pourra paralléliser les *VACUUM* sur chaque partition :

```
for i in $( seq 0 4 ) ;do vacuumdb -v -t commandes_${i}_5 v11 & done; wait
```

SUPPORT DU TRUNCATE DANS LA RÉPLICATION LOGIQUE

Le test se déroulera à partir de deux instances : L'instance `data` est en écoute sur le port 5432. L'instance `data2` est en écoute sur le port 5433.

Sur la première instance `data` dans la base `workshop11`, création de la table `t1` et insertion de quelques valeurs :

```
workshop11=# CREATE TABLE t1 (c1 int);
CREATE TABLE
workshop11=# INSERT INTO t1 SELECT generate_series(1,10);
INSERT 0 10
workshop11=# SELECT * FROM t1;
 c1
----
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
(10 rows)
```

Création de la publication `p1` :

```
workshop11=# CREATE PUBLICATION p1 FOR TABLE t1;
CREATE PUBLICATION
```

Sur la deuxième instance `data2` dans la base `workshop11_2`, création d'une table `t1` sans aucune donnée.

```
workshop11_2=# CREATE TABLE t1 (c1 int);
CREATE TABLE
```

Création de la souscription `s1` :

```
workshop11_2=# CREATE SUBSCRIPTION s1
               CONNECTION  'host=/tmp/ port=5432 dbname=workshop11' PUBLICATION p1;
NOTICE:  created replication slot "s1" on publisher
CREATE SUBSCRIPTION
```

Vérification de la réplication des données :

```
workshop11_2=# SELECT * FROM t1;
 c1
----
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
(10 rows)
```

Sur l'instance `data` nous vidons la table avec la commande `TRUNCATE` :

```
workshop11=# TRUNCATE t1;
TRUNCATE TABLE
```

La table `t1` est vide :

```
workshop11=# select * from t1;
 c1
----
(0 rows)
```

Sur l'instance `data2` nous vérifions que la réplication a été effectuée et que la table a bien été vidée :

```
workshop11_2=# select * from t1;
 c1
----
(0 rows)
```

MISE À JOUR MAJEURE AVEC LA RÉPLICATION LOGIQUE

Le présent exemple de migration est réalisé avec la base pgbench, sous CentOS 6. Il est bien sûr conseillé de réaliser la migration dans un environnement de test avant de passer en production.

Limitations : la réplication logique en version 10 ne réplique que les données. L'ordre `TRUNCATE` est à exclure lors de la réplication, ainsi que les ordres DDL, et toute modification de schéma de manière générale.

L'atelier est réalisé sur 2 instances différenciées par leur port - 5432 pour PG10 et 5433 pour PG11 - avec comme adresse commune 127.0.0.1.

> Par convention, l'invite de commande indique l'utilisateur à utiliser :
>
> `$` : à exécuter par l'utilisateur système `postgres`
>
> `#` : à exécuter par l'utilisateur système `root`

AU PROGRAMME

- Installation de Postgres 10 et 11
- Configuration PostgreSQL pour la réplication logique, à réaliser sur les 2 instances.
- Clé primaire
- Réplication du schéma
- Mise en œuvre de la réplication
- Préparation de la bascule
- Bascule

PRÉPARER L'ENVIRONNEMENT DE L'ATELIER.

- **Installer PostgreSQL 10 et 11**

```
# pg10=https://yum.postgresql.org/10/redhat/rhel-6.10-x86_64/
# pg10+=pgdg-centos10-10-2.noarch.rpm
# yum install ${pg10} -y
# pg11=https://yum.postgresql.org/11/redhat/rhel-6.10-x86_64/
# pg11+=pgdg-centos11-11-2.noarch.rpm
# yum install ${pg11} -y
# yum makecache
# yum install -y postgresql10 postgresql10-contrib postgresql10-server
```

```
# yum install -y postgresql11 postgresql11-contrib postgresql11-server
```

- **Initialiser les instances**

```
# service postgresql-10 initdb
# service postgresql-11 initdb
```

- **Changer le port d'écoute de l'instance PG11 en 5433**

```
$ sed -i "s/#port = 5432/port = 5433/" /var/lib/pgsql/11/data/postgresql.conf
```

L'instance PG10 restera sur le port 5432.

- **Démarrer les instances PG au démarrage de l'OS**

```
# chkconfig postgresql-10 on
# chkconfig postgresql-11 on
```

- **Démarrer les services**

```
# service postgresql-10 start
# service postgresql-11 start
```

- **Créer la base pgbench sur l'instance PG10** (port 5432) qui sera répliquée vers l'instance PG11 (port 5433)

Créer l'utilisateur dédié (entrer le mot de passe « *pass* ») :

```
# su - postgres
$ /usr/pgsql-10/bin/createuser -p 5432 -P bench
$ cat >>~postgres/.pgpass<<EOF
*:*:*:bench:pass
EOF
$ chmod 600 ~postgres/.pgpass
```

Créer la base bench (initialisée avec une taille de 648 Mo environ) :

```
$ /usr/pgsql-10/bin/createdb -p 5432 -O bench bench
$ /usr/pgsql-10/bin/pgbench -s 50 -i -U bench -h 127.0.0.1 -p 5432 bench
```

Autoriser l'accès à la base `bench` en local en utilisant le mode d'authentification `md5`. Les accès sont définis par le fichier `/var/lib/pgsql/10/data/pg_hba.conf`. Y ajouter les deux lignes suivantes :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    bench           repli           127.0.0.1/32            md5
host    bench           all             0.0.0.0/32              md5

local   all             all                                     peer
```

```
host    all             all             127.0.0.1/32            ident
host    all             all             ::1/128                 ident
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            ident
host    replication     all             ::1/128                 ident
```

Recharger la configuration :

```
$ psql -p 5432 -U postgres -c "select pg_reload_conf();"
```

CONFIGURER POSTGRESQL POUR LA RÉPLICATION LOGIQUE

Configuration à réaliser sur les 2 instances !

- **Création du rôle `repli` avec le droit de réplication**

Comme mot de passe, entrez « *pass* » :

```
$ /usr/pgsql-10/bin/createuser -p 5432 --replication -P repli
$ cat >>~postgres/.pgpass<<EOF
*:*:*:repli:pass
EOF
```

- **Modifier le niveau de réplication sur les 2 instances**

On passe le niveau de réplication de `replica` à `logical` sur les 2 instances.

Ces scripts remplacent les lignes `#wal_level = replica` par `wal_level = logical` dans le fichier de configuration `postgresql.conf`.

```
$ cd /var/lib/pgsql/10/data
$ sed -i 's/#wal_level\ =\ replica/wal_level\ =\ logical/' postgresql.conf
$ cd /var/lib/pgsql/11/data
$ sed -i 's/#wal_level\ =\ replica/wal_level\ =\ logical/' postgresql.conf
```

- **Redémarrer les 2 instances**

```
$ exit
# service postgresql-10 restart
# service postgresql-11 restart
```

SIMULATION D'UNE APPLICATION CLIENTE

Dans un autre terminal, on lance le script suivant pour simuler des applications clientes qui modifieront la base pendant toutes les opérations, sauf la bascule :

```
$ /usr/pgsql-10/bin/pgbench -h 127.0.0.1 -p 5432 -U bench \
  -d bench -c3 -n -C -j1 -R 5 -T10000
```

CLÉS PRIMAIRES

Il est fortement recommandé d'avoir une clé primaire **sur chaque table à répliquer**. Si une PK est absente, les risques encourus sont :

- volume de données écrites plus important ;
- contenu des tables incohérents entre les 2 instances ;
- volume de données plus important à répliquer en cas de `DELETE` ou `UPDATE`.
- **On recherche les tables sans clés primaire et on l'ajoute** (quitte à créer une colonne si nécessaire)

```
# su - postgres
$ psql -h 127.0.0.1 -p 5432 -U bench -d bench
bench=> SELECT n.nspname, c.relname FROM pg_class c
  JOIN pg_namespace n ON n.oid=c.relnamespace
  WHERE c.relkind='r' AND n.nspname !~ '^(pg_.*|information_schema)$'
    AND NOT EXISTS (select 1 from pg_index i
                    where i.indrelid = c.oid and i.indisprimary);

 nspname |     relname
---------+-----------------
 public  | pgbench_history
(1 row)

bench=> ALTER TABLE public.pgbench_history ADD COLUMN id integer
        PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY;
ALTER TABLE
```

RÉPLICATION DU SCHÉMA

Il est nécessaire de préparer des bases vides côté cible avant la mise en réplication. Si ce n'est pas déjà fait, recréer les objets globaux dans l'instance de destination. Cela nous permet de simplifier la procédure en synchronisant les rôles, les mots de passe et les éventuels tablespaces :

- **Répliquer les objets globaux**

```
$ pg_dumpall  -p 5432 -U postgres  --globals | psql -p 5433
```

Il peut y avoir des erreurs si certains objets globaux existent déjà. Il est nécessaire de vérifier les erreurs renvoyées par psql. Puis nous créons une base vide avec le même schéma que la base d'origine dans l'instance de destination. Les données y seront répliquées par la suite.

- **Générer le schéma de la base bench sur l'instance secondaire (PG11)**

```
$ /usr/pgsql-11/bin/createdb -e -O bench bench -p 5433
$ pg_dump -U bench -d bench -h 127.0.0.1 -p 5432 -v --schema-only \
|psql -d bench -p 5433
```

MISE EN ŒUVRE DE LA RÉPLICATION

Nous pouvons désormais configurer la réplication logique entre les deux instances. Commençons par l'initialisation de la publication sur l'instance PG10.

- **Créer la publication**

```
$ psql -p 5432 -d bench -c 'CREATE PUBLICATION pub_bench_10 FOR ALL TABLES'
```

- **Donner les droits sur la base** `bench`

Nous devons ensuite nous assurer que l'utilisateur de réplication a au minimum le droit de lecture sur les données sur la base bench :

```
$ psql -p 5432 -c "GRANT bench to repli"
```

- **Créer l'abonnement sur PG11**

```
$ cat <<'EOQ' | psql bench -p 5433
CREATE SUBSCRIPTION sub_bench_11
       CONNECTION 'host=127.0.0.1 user=repli dbname=bench'
       PUBLICATION pub_bench_10
EOQ
```

- **Vérifier l'état de la réplication**

Côté abonnement (instance PG11), vérifier dans les logs de PostgreSQL, dans le répertoire */var/lib/pgsql/11/data/log/*, la présence des messages suivants :

```
logical replication apply worker for subscription "sub_bench_11" has started
logical replication table synchronization worker for subscription "sub_bench_11"
 ... table "pgbench_accounts" has started
 ... table "pgbench_branches" has started
 ... table "pgbench_branches" has finished
 ... table "pgbench_history" has started
 ... table "pgbench_history" has finished
 ... table "pgbench_tellers" has started
 ... table "pgbench_tellers" has finished
 ... table "pgbench_accounts" has finished
```

Vous pouvez regarder, pendant la synchronisation, le contenu des logs de l'instance PG10, et vérifier que la charge en processeur est supportable (dans le cas d'une petite configuration ou d'une charge en écriture bien plus importante que dans cet exemple).

Pour vérifier que les données sont bien transmises en permanence entre les deux bases, cette requête sur la clé primaire que nous avons ajoutée précédemment doit montrer une valeur qui s'incrémente régulièrement si on l'exécute simultanément des deux côtés :

```
bench=# SELECT max(id) FROM pgbench_history ;
```

Nous attendons que l'écart (*lag*) entre les deux serveurs soit entièrement résorbé. Depuis le serveur PG10, observez l'évolution des compteurs du lag présentés par `pg_stat_replication`. Notez que même si les autres compteurs évoluent encore, les écritures en question ne concernent plus la base qui nous intéresse (et ces écritures ne sont pas envoyées vers l'instance PG11). Nous patienterons juste quelques secondes que les écarts affichés soient résorbés.

```
$ watch -n2 "psql -p 5432 -d bench -xc \
\"select confirmed_flush_lsn from pg_replication_slots where slot_name='sub_bench_11'\""
```

Il est aussi possible de simplement comparer des données représentatives de la réplication. Par exemple :

- sur l'instance **PG10** :

```
$ psql -p 5432 -d bench -Atc "select max(id) from pgbench_history"
```

- sur l'instance **PG11** :

```
$ psql -p 5433 -U bench -d bench -Atc "select max(id) from pgbench_history"
```

> Aucune donnée ne doit plus être écrite dans les tables migrées avant la fin de la bascule !

- **Répliquer les séquences**

La réplication logique ne réplique pas les séquences. Aussi, il nous faut mettre à jour les séquences sur l'instance PG11 avant d'effectuer la bascule.

L'ordre suivant va générer des requêtes pour toutes les séquences concernées, comme par exemple :

```
psql -U postgres -d bench -c "SELECT setval('public.pgbench_history_id_seq', 36549);"
```

pour toutes les séquences concernées :

```
$ cat <<'EOQ' | psql -At -U repli bench -h 127.0.0.1 -p 5432 | psql bench -p 5433
SELECT format( 'SELECT setval(%L, %s)',
    schemaname||'.'||sequencename, last_value
)
FROM pg_sequences where last_value is not null;
EOQ
```

PRÉPARATION DE LA BASCULE (SWITCHOVER)

Avant de commencer, il est nécessaire de couper l'accès des applications clients. Afin de sécuriser l'ensemble, nous ajoutons les lignes suivantes en **début** du fichier `/var/lib/pgsql/10/data/pg_hba.conf` afin d'empêcher toute nouvelle connexion autre que celle concernant la réplication :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    bench           repli           127.0.0.1/32            md5
local   bench           postgres                                peer
local   bench           all                                     reject
host    bench           all             0.0.0.0/0               reject

local   all             all                                     peer
host    all             all             127.0.0.1/32            ident
host    all             all             ::1/128                 ident
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            ident
host    replication     all             ::1/128                 ident
```

Egalement on prépare les accès à l'instance PG11 `/var/lib/pgsql/11/data/pg_hba.conf` :

```
host    bench           repli           127.0.0.1/32            md5
local   bench           postgres                                peer
local   bench           all                                     reject
host    bench           all             0.0.0.0/0               reject

local   all             all                                     peer
host    all             all             127.0.0.1/32            ident
host    all             all             ::1/128                 ident
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            ident
host    replication     all             ::1/128                 ident
host    replication     repli           127.0.0.1/32            md5
host    db              user1           127.0.0.1/32            md5
```

- **Recharger les configurations**

On recharger la configuration pour prendre notre modification en compte (PG10) et (PG11) :

```
$ psql -p5432  -c "select pg_reload_conf();"
$ psql -p5433  -c "select pg_reload_conf();"
```

- **Arrêt des connexions distantes**

Vérifier que le script pgbench d'arrière-plan ne se connecte plus. Ne pas l'arrêter.

Tuer toute connexion restante à la base bench sur l'instance PG10 :

```
$ cat<<EOQ|psql -p 5432
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE backend_type = 'client backend'
  AND pid <> pg_backend_pid()
  AND datname='bench';
EOQ
```

BASCULE

Nous pouvons désormais effectuer la bascule qui consiste à échanger les rôles au sein de la réplication logique. Ces commandes sont à exécuter sur l'instance PG11 (port 5433) :

```
$ psql -p 5433 -d bench -c 'DROP SUBSCRIPTION sub_bench_11'
$ psql -p 5433 -c 'GRANT bench TO repli'
$ psql -p 5433 -d bench -c 'CREATE PUBLICATION pub_bench_11 FOR ALL TABLES'
```

Notez que les données étant déjà présentes sur PG10, nous demandons explicitement de ne pas initialiser les données lors de la création de l'abonnement :

```
$ cat <<'EOQ' | psql bench -p 5432
CREATE SUBSCRIPTION sub_bench_10
         CONNECTION 'host=127.0.0.1 user=repli port=5433 dbname=bench'
        PUBLICATION pub_bench_11
              WITH (copy_data = false)
EOQ
```

Vérifier dans les logs des deux instances que tout va bien.

- **Ouvrir les accès clients à la base bench sur PG11**

`/var/lib/pgsql/11/data/pg_hba.conf` :

```
host    bench           repli           127.0.0.1/32            reject
local   bench           postgres                                peer
local   bench           all                                     md5
host    bench           all             0.0.0.0/0               md5

local   all             all                                     md5
host    all             all             127.0.0.1/32            ident
host    all             all             ::1/128                 ident
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            ident
host    replication     all             ::1/128                 ident
host    replication     repli           127.0.0.1/32            md5
```

Puis recharger la configuration :

```
$ psql -p5433  -c "select pg_reload_conf();"
```

- **Relancer les applis clientes**

Ne pas oublier de modifier leur chaîne de connexion :

```
/usr/pgsql-11/bin/pgbench -h 127.0.0.1 -p 5433 \
-U bench -d bench -c3 -n -C -j1 -R 1 -T10000
```

> Rappel ! Si votre application utilise l'ordre `TRUNCATE` l'instance PG11 tentera de la répliquer vers l'instance PG10, provoquant alors une erreur et empêchant la réplication. Vous pouvez le remplacer par `DELETE` qui est supporté.

- **Vérifier l'état de la réplication**

On contrôle que la réplication se fait correctement vers l'ancienne instance PG10 :

```
psql -h 127.0.0.1 -p 5433 -U bench -d bench -Atc "select max(id) from pgbench_history"
psql -h 127.0.0.1 -p 5432 -U repli -d bench -Atc "select max(id) from pgbench_history"
```

SUPPRESSION DE L'ANCIENNE INSTANCE

La suppression de l'ancienne instance est simple : il suffit de supprimer la réplication logique entre les deux serveurs.

- **Suppression de l'abonnement sur l'instance PG10**

```
$ psql -p 5432 -d bench -c 'drop subscription sub_bench_10 cascade'
```

- **Suppression de la publication sur l'instance PG11**

```
psql -p 5433 -d bench -c 'drop publication pub_bench_11 cascade'
DROP PUBLICATION
```

INDEX COUVRANTS

Soit une table avec des données et une contrainte d'unicité sur 2 colonnes :

```
v11=# CREATE TABLE t2 (a int, b int, c varchar(10));
CREATE TABLE
v11=# INSERT INTO t2 (SELECT i, 2*i, substr(md5(i::text), 1, 10)
        FROM generate_series(1,10000000) AS i);
INSERT 0 10000000
v11=# CREATE UNIQUE INDEX t2_a_b_unique_idx ON t2 (a,b);
CREATE INDEX
```

Pour simplifier les plans, on désactive le parallélisme :

```
SET max_parallel_workers_per_gather TO 0 ;
```

En cas de recherche sur la colonne *a*, on va pouvoir récupérer les colonnes *a* et *b* grâce à un *Index Only Scan* :

```
v11=# EXPLAIN ANALYSE SELECT a,b FROM t2 WHERE a>110000 and a<158000;
                    QUERY PLAN
-----------------------------------------------------
 Index Only Scan using t2_a_b_unique_idx on t2
     (cost=0.43..1953.87 rows=1100 width=8)
     (actual time=0.078..28.066 rows=47999 loops=1)
   Index Cond: ((a > 1000) AND (a < 2000))
   Heap Fetches: 0
 Planning Time: 0.225 ms
 Execution Time: 12.628 ms
(5 lignes)
```

Cependant, si on veut récupérer également la colonne *c*, on passera par un *Index Scan* et un accès à la table :

```
v11=# EXPLAIN ANALYSE SELECT a,b,c FROM t2 WHERE a>110000 and a<158000;
                    QUERY PLAN
-----------------------------------------------------
 Index Scan using t2_a_b_unique_idx on t2
     (cost=0.43..61372.04 rows=46652 width=19)
     (actual time=0.063..13.073 rows=47999 loops=1)
   Index Cond: ((a > 110000) AND (a < 158000))
 Planning Time: 0.223 ms
 Execution Time: 16.034 ms
(4 lignes)
```

Dans notre exemple, le temps réel n'est pas vraiment différent entre les 2 requêtes. Si l'optimisation de cette requête est cependant cruciale, nous pouvons créer un index spécifique incluant la colonne *c* et permettre l'utilisation d'un *Index Only Scan* :

```
v11=# CREATE INDEX t2_a_b_c_idx ON t2 (a,b,c);
CREATE INDEX
v11=# EXPLAIN ANALYZE SELECT a,b,c FROM t2 WHERE a>110000 and a<158000;
                    QUERY PLAN
-----------------------------------------------------
 Index Only Scan using t2_a_b_c_idx on t2
     (cost=0.56..1861.60 rows=46652 width=19)
     (actual time=0.048..11.241 rows=47999 loops=1)
   Index Cond: ((a > 110000) AND (a < 158000))
   Heap Fetches: 0
 Planning Time: 0.265 ms
 Execution Time: 14.329 ms
(5 lignes)
```

La taille cumulée de nos index est de 602 Mo :

```
v11=# SELECT pg_size_pretty(pg_relation_size('t2_a_b_unique_idx'));
 pg_size_pretty
----------------
 214 MB
(1 ligne)

v11=# SELECT pg_size_pretty(pg_relation_size('t2_a_b_c_idx'));
 pg_size_pretty
----------------
 387 MB
(1 ligne)
```

En v11 nous pouvons utiliser à la place un seul index appliquant toujours la contrainte d'unicité sur les colonnes *a* et *b* **et** couvrant la colonne *c* :

```
v11=# CREATE UNIQUE INDEX t2_a_b_unique_covering_c_idx ON t2 (a,b) INCLUDE (c);
CREATE INDEX
v11=# EXPLAIN ANALYZE SELECT a,b,c FROM t2 WHERE a>110000 and a<158000;
                    QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using t2_a_b_unique_covering_c_idx on t2
     (cost=0.43..1857.47 rows=46652 width=19)
     (actual time=0.045..11.945 rows=47999 loops=1)
   Index Cond: ((a > 110000) AND (a < 158000))
   Heap Fetches: 0
 Planning Time: 0.228 ms
 Execution Time: 14.263 ms
(5 lignes)
v11=# SELECT pg_size_pretty(pg_relation_size('t2_a_b_unique_covering_c_idx'));
 pg_size_pretty
----------------
 386 MB
```

```
(1 ligne)
```

La nouvelle fonctionnalité sur les index couvrants nous a permit d'éviter la création de 2 index pour un gain de 35% d'espace disque !

Noter que la colonne `c` est renseignée depuis l'index, mais elle n'est pas triée (comme dans un index normal), et donc un `ORDER BY` n'en profite pas (étape *Sort* nécessaire) :

```
v11=# EXPLAIN SELECT * FROM t2 ORDER BY a,b ;
                    QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using t2_a_b_unique_covering_c_idx on t2
             (cost=0.43..347752.43 rows=10000000 width=19)

v11=# EXPLAIN SELECT * FROM t2 ORDER BY a,b,c ;
                    QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=1736527.83..1761527.83 rows=10000000 width=19)
   Sort Key: a, b, c
   ->  Seq Scan on t2  (cost=0.00..163695.00 rows=10000000 width=19)
```

Les performances en insertion vont également être meilleures car un seul index doit être maintenu :

```
v11=# EXPLAIN ANALYSE INSERT INTO t2 (SELECT i, 2*i, substr(md5(i::text), 1, 10)
        FROM generate_series(10000001,10100000) AS i);
                    QUERY PLAN
-------------------------------------------------------------
 Insert on t2
     (cost=0.00..25.00 rows=1000 width=46)
     (actual time=502.111..502.111 rows=0 loops=1)
   ->  Function Scan on generate_series i
           (cost=0.00..25.00 rows=1000 width=46)
       (actual time=14.356..107.205 rows=100000 loops=1)
 Planning Time: 0.132 ms
 Execution Time: 502.594 ms
(4 lignes)
```

Si on supprime l'index couvrant et que l'on recrée les 2 index :

```
v11=# DROP INDEX t2_a_b_unique_covering_c_idx ;
DROP INDEX
v11=# CREATE UNIQUE INDEX t2_a_b_unique_idx ON t2 (a,b);
CREATE INDEX
v11=# CREATE INDEX t2_a_b_c_idx ON t2 (a,b,c);
CREATE INDEX
v11=# EXPLAIN ANALYSE INSERT INTO t2 (SELECT i, 2*i, substr(md5(i::text), 1, 10)
        FROM generate_series(10100001,10200000) AS i);
                    QUERY PLAN
```

```
-------------------------------------------------------------
 Insert on t2
     (cost=0.00..25.00 rows=1000 width=46)
     (actual time=842.455..842.455 rows=0 loops=1)
   ->  Function Scan on generate_series i
           (cost=0.00..25.00 rows=1000 width=46)
       (actual time=14.708..127.441 rows=100000 loops=1)
 Planning Time: 0.155 ms
 Execution Time: 843.147 ms
(4 lignes)
```

On a un gain de performance à l'insertion de 40%.

PARALLÉLISATION

Parallélisation sur les requêtes `CREATE TABLE AS SELECT`

Création d'une table *numbers* comportant 10 millions de lignes :

```
CREATE TABLE numbers AS SELECT i FROM generate_series (1,10000000) i ;
```

Modifications des paramètres `max_parallel_workers` et `max_parallel_workers_per_gather` :

```
SET max_parallel_workers TO 2;
SET max_parallel_workers_per_gather TO 1;
```

En version 10, lorsque nous créons une autre table avec `CREATE TABLE ... AS`, on obtient le plan d'exécution suivant :

```
v10=# EXPLAIN ANALYSE
  CREATE TABLE numbers2 AS SELECT * FROM numbers WHERE i < 10000;
                          QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on numbers  (cost=0.00..169247.71 rows=8279 width=4)
                      (actual time=0.110..592.257 rows=9999 loops=1)
   Filter: (i < 10000)
   Rows Removed by Filter: 9990001
 Planning time: 0.865 ms
 Execution time: 621.130 ms
(5 lignes)
```

En version 11, l'optimiseur effectue un scan séquentiel en parallèle :

```
v11=# EXPLAIN ANALYSE
  CREATE TABLE numbers2 AS SELECT * FROM numbers WHERE i < 10000;
                          QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..119724.44 rows=9472 width=4)
         (actual time=1.932..332.705 rows=9999 loops=1)
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Seq Scan on numbers
          (cost=0.00..117777.24 rows=5572 width=4)
          (actual time=0.138..320.575 rows=5000 loops=2)
         Filter: (i < 10000)
         Rows Removed by Filter: 4995000
 Planning Time: 0.220 ms
 Execution Time: 363.189 ms
(8 lignes)
```

En version 11, désactivation du paramètre `max_parallel_worker_per_gather` :

```
v11=# SET max_parallel_workers_per_gather TO 0;
```

L'optimiseur utilise alors un scan séquentiel classique :

```
v11=# EXPLAIN ANALYSE
  CREATE TABLE numbers2_bis AS SELECT * FROM numbers WHERE i < 10000;
                          QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on numbers  (cost=0.00..169247.71 rows=9472 width=4)
                      (actual time=0.169..563.316 rows=9999 loops=1)
   Filter: (i < 10000)
   Rows Removed by Filter: 9990001
 Planning Time: 0.290 ms
 Execution Time: 592.385 ms
(5 lignes)
```

Parallélisation de `CREATE MATERIALIZED VIEW`

A partir de la même table *numbers*, création d'une vue materialisée :

En version 10 :

```
v10=# EXPLAIN ANALYSE
  CREATE MATERIALIZED VIEW view_numbers AS SELECT * FROM numbers WHERE i < 10000;
                          QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on numbers  (cost=0.00..169247.71 rows=11751 width=4)
                      (actual time=0.103..745.242 rows=9999 loops=1)
   Filter: (i < 10000)
   Rows Removed by Filter: 9990001
 Planning time: 0.061 ms
 Execution time: 1969.551 ms
(5 lignes)
```

En version 11 :

```
v11=# SET max_parallel_workers_per_gather TO 2;
SET
v11=# EXPLAIN ANALYSE
  CREATE MATERIALIZED VIEW view_numbers AS SELECT * FROM numbers WHERE i < 10000;
                          QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..98303.61 rows=9724 width=4)
         (actual time=704.848..815.472 rows=9999 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on numbers
          (cost=0.00..96331.21 rows=4052 width=4)
          (actual time=695.337..798.511 rows=3333 loops=3)
         Filter: (i < 10000)
         Rows Removed by Filter: 3330000
```

```
 Planning Time: 0.141 ms
 Execution Time: 829.369 ms
(8 lignes)
```

Parallélisation de la création d'une table avec `SELECT INTO`

En version 10 :

```
v10=# EXPLAIN ANALYSE SELECT * INTO numbers3 FROM numbers WHERE i < 10000;
                          QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on numbers  (cost=0.00..169247.71 rows=11751 width=4)
                      (actual time=0.154..790.222 rows=9999 loops=1)
   Filter: (i < 10000)
   Rows Removed by Filter: 9990001
 Planning time: 0.108 ms
 Execution time: 817.480 ms
(5 lignes)
```

En version 11 :

```
v11=# EXPLAIN ANALYSE SELECT * INTO numbers3 FROM numbers WHERE i < 10000;
                          QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..98303.61 rows=9724 width=4)
         (actual time=0.683..433.006 rows=9999 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on numbers
          (cost=0.00..96331.21 rows=4052 width=4)
          (actual time=0.047..429.656 rows=3333 loops=3)
         Filter: (i < 10000)
         Rows Removed by Filter: 3330000
 Planning Time: 0.144 ms
 Execution Time: 446.445 ms
(8 lignes)
```

SAUVEGARDE DES DROITS AVEC PG_DUMP

Nous allons tester les évolutions dans les outils de sauvegarde logique `pg_dump` et `pg_dumpall`.

Créons une nouvelle base de données, créons 2 utilisateurs avec des droits spécifiques. Nous allons tout d'abord créer un fichier avec les ordres SQL puis les charger en version 10 et 11 :

```
cat  >> create_db_droits.sql <<EOF
CREATE DATABASE droits;
\c droits
REVOKE ALL ON DATABASE droits FROM PUBLIC;
CREATE ROLE alice;
ALTER DATABASE droits OWNER TO alice;
ALTER ROLE alice IN DATABASE droits SET work_mem to '100MB';
CREATE SCHEMA appli;
ALTER DATABASE droits SET search_path TO appli, public;
CREATE ROLE bob;
GRANT CONNECT,TEMPORARY ON DATABASE droits TO bob;
GRANT ALL ON SCHEMA appli TO bob;
ALTER DEFAULT PRIVILEGES IN SCHEMA appli GRANT ALL ON TABLES TO bob;
EOF

psql -f create_db_droits.sql
psql -p5433 -f create_db_droits.sql
```

Nous allons maintenant sauvegarder la base de données avec l'outil `pg_dump` avec et sans l'option `--create` en mode *plain* puis les objets globaux avec l'outil `pg_dumpall`. Ceci dans les version 10 et 11 :

```
/usr/pgsql-11/bin/pg_dump -d droits -Fp > /tmp/droits_base_v11.sql
/usr/pgsql-11/bin/pg_dump -d droits -Fp --create > /tmp/droits_create_v11.sql
/usr/pgsql-11/bin/pg_dumpall -g > /tmp/pg_dumpall_v11.sql
/usr/pgsql-10/bin/pg_dump -p5433 -d droits -Fp > /tmp/droits_base_v10.sql
/usr/pgsql-10/bin/pg_dump -p5433 -d droits -Fp --create > /tmp/droits_create_v10.sql
/usr/pgsql-10/bin/pg_dumpall -p5433 -g > /tmp/pg_dumpall_v10.sql
```

Avec la commande `grep work_mem /tmp/*.sql` vérifiez que l'ordre SQL `ALTER ROLE alice IN DATABASE droits SET work_mem TO '100MB';` apparaît :

- en version 10 dans la sortie de `pg_dumpall -g`,
- en version 11 dans la sortie de `pg_dump --create`.

Avec des commandes `grep` bien choisies, vérifiez que les ordres suivants n'apparaissent que dans `pg_dump --create` en version 11 :

- `ALTER DATABASE droits SET search_path TO 'appli', 'public';`

- `GRANT CONNECT,TEMPORARY ON DATABASE droits TO bob;`
- `REVOKE CONNECT,TEMPORARY ON DATABASE droits FROM PUBLIC;`

En mode de sauvegarde logique *custom* et *directory*, ces ordres seront sauvegardés. Ils ne seront cependant restaurés que si l'option `--create` de l'outil `pg_restore` est précisé.

Suite à ces changements, il est important avant un passage en version 11, de vérifier attentivement le fonctionnement de ses scripts de sauvegardes. Et au besoin, de les adapter pour ne pas perdre d'informations.

PG_PREWARM

Ce qui suit suppose un paramètre `shared_buffers` assez grand :

```
shared_buffers = '512MB'  # redémarrage nécessaire en cas de changement
```

Créons une table de 346 Mo puis exécutons une requête dessus :

```
CREATE TABLE matable AS SELECT i FROM generate_series(1,10000000) i;

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM matable ;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                     (actual time=0.007..694.690 rows=10000000 loops=1)
Buffers: shared hit=44248
Planning Time: 0.065 ms
Execution Time: 1080.083 ms
```

La requête a été lue depuis le cache de PostgreSQL comme en témoigne le nombre de blocs de 8 ko en *shared hits*.

On redémarre PostgreSQL :

```
service postgresql-11 restart
```

La première rééxécution de la requête est bien plus lente car les blocs sont lus depuis le disque (*shared read*), ou avec de la chance depuis la cache de l'OS. Cela reste valable pour le deuxième ou troisième appel, car lors d'un parcours complet d'une grosse table, tous les blocs ne sont pas chargés en mémoire d'entrée :

```
postgres=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM matable ;

                             QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                     (actual time=0.098..948.179 rows=10000000 loops=1)
Buffers: shared read=44248
Planning Time: 0.514 ms
Execution Time: 1428.741 ms

postgres=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM matable ;

                             QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                     (actual time=0.128..796.596 rows=10000000 loops=1)
Buffers: shared hit=32 read=44216
```

```
Planning Time: 0.091 ms
Execution Time: 1215.664 ms
(4 lignes)

postgres=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM matable ;

                            QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                     (actual time=0.147..829.618 rows=10000000 loops=1)
Buffers: shared hit=64 read=44184
Planning Time: 0.109 ms
Execution Time: 1263.430 ms
(4 lignes)
```

Avec pg_prewarm, on peut accélérer ce chargement :

```
postgres=# CREATE EXTENSION pg_prewarm ;
CREATE EXTENSION

postgres=# SELECT pg_prewarm ('matable','buffer') ;

pg_prewarm
------------
  44248
(1 ligne)

postgres=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM matable ;
                            QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                     (actual time=0.016..715.889 rows=10000000 loops=1)
Buffers: shared hit=44248
Planning Time: 0.038 ms
Execution Time: 1123.740 ms
(4 lignes)
```

L'extension pg_buffercache permet de voir le contenu des *shared buffers* (ici on filtre les tables et index systèmes pour la lisibilité) :

```
postgres=# CREATE EXTENSION pg_buffercache ;
CREATE EXTENSION

postgres=# SELECT c.relname, count(*) AS buffers,
   pg_size_pretty(count(*)*8192) as taille_mem
   FROM pg_buffercache b INNER JOIN pg_class c
   ON b.relfilenode = pg_relation_filenode(c.oid) AND
      b.reldatabase IN (0, (SELECT oid FROM pg_database
```

```
                          WHERE datname = current_database()))
   WHERE relname not like 'pg_%'
   GROUP BY c.relname ;

relname | buffers | taille_mem
---------+---------+------------
matable |   44248 | 346 MB
(1 ligne)
```

Faisons en sorte que PostgreSQL charge la table dès le démarrage. Dans `postgresql.conf` :

```
shared_preload_libraries = 'pg_prewarm'
pg_prewarm.autoprewarm = true
```

Avant de redémarrer PostgreSQL on demande à sauvegarder le contenu du cache :

```
postgres=# SELECT autoprewarm_dump_now() ;

autoprewarm_dump_now
----------------------
           44537
(1 ligne)
```

On redémarre :

```
service postgresql-11 restart
```

Et l'on vérifie qu'avant toute requête la table est déjà en cache avec la requête ci-dessus sur `pg_buffercache` :

```
relname | buffers | taille_mem
---------+---------+------------
matable |   44248 | 346 MB

postgres=#  EXPLAIN (ANALYZE,BUFFERS) select * from matable ;

                        QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on matable  (cost=0.00..144247.77 rows=9999977 width=4)
                      (actual time=0.042..713.662 rows=10000000 loops=1)
   Buffers: shared hit=44248
 Planning Time: 0.348 ms
 Execution Time: 1107.820 ms
(4 lignes)
```

JIT

Le JIT est difficile à reproduire sur une machine de bureau. Les gains n'étant visibles que pour des requêtes coûteuses, manipulant et agrégeant un grand volume de données. Dans le fichier de configuration `postgresql.conf`, monter les paramètres suivants au moins à :

```
shared_buffers = '2GB'
work_mem = '1500MB'
```

Puis redémarrer votre instance :

```
# service postgresql-11 restart
```

La table suivante imite une table de faits d'un *datawarehouse* de vente, avec des dizaines de millions de lignes et environ 2 Go de taille :

```
DROP TABLE IF EXISTS faits_commandes CASCADE;

CREATE TABLE faits_commandes AS
  SELECT
    extract('year' FROM date_commande) AS annee_commande,
    to_char(date_commande, 'YYYYMM') AS mois_commande,
    to_char(date_commande, 'IYYYIW') AS semaine_commande,
    x3.*,
    CASE WHEN mod(client_code, 3) > 0
          AND extract('month' FROM date_commande) < 11
      THEN round((random() / 10) ::numeric, 2)
      ELSE 0
    END AS remise,
    mod(quantite, 10) AS nb_paquets,
    (100000000 * random())::int AS numero_lot,
    md5(ligne_num::text) AS code_confirmation,
    round((random() * quantite * poids_unitaire) ::numeric, 0) AS cout_expedition
FROM (
  SELECT
    ligne_num,
    commande_num,
    CASE WHEN random() > 0.95 THEN TRUE ELSE FALSE END AS commande_annulee,
    client_code,
    mod(client_code, 5) AS type_client,
    date_commande,
    d0 + (commande_num+(4*dmois*random())::int) * interval '2 hour'
      AS date_production,
    d0 + (commande_num+50+(dmois*random())::int) * interval '2 hour'
      AS date_expedition,
    d0 + (commande_num+80+(dmois*random())::int) * interval '2 hour'
```

```
    AS date_livraison,
  d0 + (commande_num+(4*dmois*random())::int) * interval '2 hour'
    AS date_facturation,
  d0 + (commande_num+250+(1500*random())::int) * interval '1 hour'
    AS date_paiement,
  (random() > 0.5) AS flag1,
  (random() > 0.3) AS flag2,
  (random() > 0.1) AS flag3,
  (random() > 0.9) AS flag4,
  (random() > 0.99) AS flag5,
  (random() > 0.6) AS flag6,
  (random() > 0.99) AS flag7,
  (random() > 0.999) AS flag8,
  (random() > 0.88) AS flag9,
  article_code,
  mod(article_code, 54) AS fournisseur_code,
  prix_unitaire_base,
  prix_unitaire_base * (0.85 + 0.3 * random()) AS prix_unitaire,
  (client_code * random() / 3) ::int AS quantite,
  mod(article_code, 3) / 10 AS taux_tva,
  round((prix_unitaire_base * (0.3 + random()))::numeric, 2) AS poids_unitaire,
  CASE mod(client_code, 9)
    WHEN 1 THEN 'FR' WHEN 2 THEN 'FR' WHEN 3 THEN 'FR'
    WHEN 4 THEN 'DE'
    WHEN 5 THEN 'GB' WHEN 7 THEN 'GB'
    WHEN 6 THEN 'BE'
    ELSE 'UE'
  END AS pays_destination
  FROM (
    SELECT
        *,
        d0 + commande_num * interval '1 hour' AS date_commande,
        extract('month' FROM (d0 + commande_num*interval '1 hour')) AS dmois
    FROM (
        SELECT
            i AS ligne_num,
            round(100000 * random())::int AS commande_num,
            mod(round(100000 * random())::int, 333) AS client_code,
            (1000 * random())::int AS article_code,
            70 * random() AS prix_unitaire_base,
            '2007-01-01 00:00:00'::timestamptz AS d0
        FROM
            generate_series(1, 8000000) i
  ) x1
) x2
) x3;
```

```
\echo Petit extrait
SELECT * FROM faits_commandes LIMIT 6;

\echo Taille de la table de faits :
SELECT pg_size_pretty(pg_relation_size('faits_commandes')) AS taille_table;

\echo VACUUM ANALYZE
VACUUM ANALYZE faits_commandes;
```

Notre requête de test calcule des statistiques sur toute la table. Pour des raisons de praticité, elle est créée sous forme de vue :

```
DROP VIEW IF EXISTS stats_commandes_v;

CREATE OR REPLACE VIEW stats_commandes_v AS
SELECT
  annee_commande,
  mois_commande,
  sum(ca)::bigint AS ca_global_mois,
  sum(ca) filter (WHERE (
    extract(day FROM date_expedition) < 8))::bigint AS ca_semaine1,
  sum(ca) filter (WHERE (
    extract(day FROM date_expedition) BETWEEN 8 AND 16))::bigint AS ca_semaine2,
  sum(ca) filter (WHERE (
    extract(day FROM date_expedition) BETWEEN 17 AND 23))::bigint AS ca_semaine3,
  sum(ca) filter (WHERE (
    extract(day FROM date_expedition) > 23))::bigint AS ca_semaine4,
  --
  sum(ca) filter (WHERE commande_annulee IS TRUE) AS ca_annule,
  --
  sum(ca) filter (WHERE pays_destination = 'FR') AS ca_fr,
  sum(ca) filter (WHERE pays_destination = 'DE') AS ca_de,
  sum(ca) filter (WHERE pays_destination = 'GB') AS ca_gb,
  sum(ca) filter (WHERE pays_destination = 'BE') AS ca_be,
  sum(ca) filter (WHERE pays_destination = 'UE') AS ca_ue,
  --
  (sum(ca) filter (WHERE pays_destination = 'FR') / sum(ca)) AS ca_proportion_fr,
  sum(ca) filter (WHERE flag1 IS TRUE) AS ca_urgent,
  sum(ca) filter (WHERE flag1 IS TRUE AND flag2 IS TRUE) AS ca_urgent_bon_client,
  sum(ca) filter (WHERE flag5 IS TRUE AND commande_annulee IS FALSE)
    AS ca_annulation_interne,
  sum(quantite) AS qte_tot_mois,
  max(max(quantite)) OVER (PARTITION BY mois_commande) AS qte_commande_max_mois,
  count(DISTINCT commande_num) AS nb_commandes,
  count(DISTINCT commande_num) FILTER (WHERE commande_annulee IS FALSE)
    AS nb_commandes_annulees,
```

```
  (count(DISTINCT commande_num) FILTER (WHERE commande_annulee IS FALSE))
    / count(DISTINCT commande_num) AS ratio_annulation,
  --
   avg(date_livraison - date_expedition) AS delai_reception_moyen,
   avg(date_livraison - date_production) AS delai_prod_client_moyen,
   avg(date_livraison - date_commande) AS delai_livraison_total_moyen,
  --
   round(avg(nb_paquets) FILTER (WHERE commande_annulee IS FALSE), 1)
     AS nb_paquets_moyen,
   sum(quantite) FILTER (WHERE commande_annulee IS FALSE)
     / sum(nb_paquets) FILTER (WHERE commande_annulee IS FALSE)
     AS qte_moyenne_par_paquet,
  --
  ( avg(delai_facturation))::int AS delai_facturation_moyen,
  ( avg(sum(delai_facturation) / count(delai_facturation))
    OVER (PARTITION BY annee_commande))::int AS delai_facturation_moyen_annuel,
  ( avg(delai_paiement))::int AS delai_paiement_moyen,
  ( avg(sum(delai_paiement) / count(delai_paiement))
    OVER (PARTITION BY annee_commande))::int AS delai_paiement_moyen_annuel
  ----
FROM (
  SELECT
    l.*,
    l.quantite * prix_unitaire * (1 - remise) AS ca,
    l.quantite * prix_unitaire_base AS ca_base,
    extract('days' FROM date_facturation - date_commande) AS delai_facturation,
    extract('days' FROM date_paiement - date_facturation) AS delai_paiement
      FROM faits_commandes l
      WHERE date_expedition > '15-01-2007'
        AND commande_num > 7
        AND commande_num != 88
        AND commande_num != 666
        AND numero_lot > 5
        AND numero_lot NOT IN (666, 999, 888, 123456789)
        AND fournisseur_code != 1528
        AND article_code NOT IN (673, 1942)
        AND remise < 1
        AND (flag1 OR flag2 OR flag3 OR flag4 OR flag5
             OR flag6 OR flag7 OR flag8 OR flag9)
  ) details
GROUP BY annee_commande, mois_commande;
```

Avant de lancer les tests, figeons la configuration, et forçons le chargement de la table (ou du moins la plus grande partie possible) en mémoire partagée grâce à pg_prewarm :

```
SET max_parallel_workers_per_gather TO 1;
```

```
\echo Préchargement en mémoire autant que possible
CREATE EXTENSION pg_prewarm;
SELECT pg_prewarm ('faits_commandes');

\echo Shared buffers
SHOW shared_buffers;
\echo Work mem
SHOW work_mem;

\pset pager off
```

On teste en activant le JIT. Vue la taille de la requête il ne sera pas utile de le forcer en descendant `jit_above_cost` à 0.

```
\echo "Tests avec JIT"

SET jit TO on;
SET jit_above_cost TO default;
SET jit_inline_above_cost TO default;
SET jit_optimize_above_cost TO default;

SHOW jit;
SHOW jit_above_cost;
SHOW jit_inline_above_cost;
SHOW jit_optimize_above_cost;

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM stats_commandes_v
ORDER BY 1,2;
```

Rééxécuter la requête plusieurs fois pour vérifier que le temps d'exécution est reproductible.

Vous devez trouver en fin du plan mention du JIT et des optimisations effectuées. Ici on voit notamment que le temps de génération du code n'est que de quelques millisecondes, mais plus d'une demi-seconde a été perdu à optimiser au maximum le code (*Optimization time*), et presque autant à générer le code object final (*Emission Time*) :

```
 Planning Time: 0.520 ms
 JIT:
   Functions: 27
   Generation Time: 7.278 ms
   Inlining: true
   Inlining Time: 21.696 ms
   Optimization: true
   Optimization Time: 611.467 ms
   Emission Time: 421.446 ms
 Execution Time: 39285.440 ms
```

```
(37 lignes)

Durée : 39286,546 ms (00:39,287)
```

Comparons avec le temps d'exécution, sans le JIT :

```
SET jit TO off;

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM stats_commandes_v
ORDER BY 1,2;
```

Ce qui nous donne sur la même machine que précédemment :

```
 Execution Time: 44686.972 ms
```

Le gain en temps tourne donc ici autour des 10 %.

Si la machine disponible est assez puissante, on peut ensuite augmenter le nombre de lignes générées dans la table. L'écart devrait devenir de plus en plus important. Si possible, monter `shared_buffers` à 4 Go.
