Partitionnement déclaratif et réplication logique

Mixons les nouveautés PostgreSQL 10

novembre 2017

Thibaut Madelaine

Mixons les nouveautés PostgreSQL 10

Partitionnement déclaratif et réplication logique

TITRE : Mixons les nouveautés PostgreSQL 10
SOUS-TITRE : Partitionnement déclaratif et réplication logique

DATE: novembre 2017

QUI SUIS-JE ?

- Thibaut Madelaine
- chez Dalibo depuis 1 an
- Longtemps développeur logiciel
- Désormais également DBA, formateur, consultant

POSTGRESQL VERSION 10

- Sortie le 5 octobre 2017
- supportée jusqu'en octobre 2022

DE TRÈS NOMBREUSES ÉVOLUTIONS

- le changement de numérotation
- les changements de nommages
- des améliorations de performances
- ...

Ancienne numérotation exprimée sur 3 nombres :

```
9 . 6 . 3
Majeure1 . Majeure2 . Mineure
```

Nouvelle numérotation exprimée sur 2 nombres uniquement :

```
10 . 2
Majeure . Mineure
```

Les renommages : * Au niveau des répertoires

* `pg_xlog` -> `pg_wal`

* `pg_clog` -> `pg_xact`

* Au niveau des fonctions

* `xlog` -> `wal`

* `location` -> `lsn`

* Au niveau des outils

* `xlog` -> `wal`

DEUX BELLES NOUVEAUTÉS

- Partitionnement natif
- Réplication logique

PARTITIONNEMENT

- Petit rappel sur l'ancien partitionnement
- Nouveau partitionnement
- Nouvelle syntaxe
- Quelques limitations

PostgreSQL dispose d'un contournement permettant de partitionner certaines tables. La mise en place et la maintenance de ce contournement étaient complexes. La version 10 améliore cela en proposant une intégration bien plus poussée du partitionnement.

ANCIEN PARTITIONNEMENT

- Le partitionnement par héritage se base sur
  - la notion d'héritage (1 table mère et des tables filles)
  - des triggers pour orienter les insertions vers les tables filles
  - des contraintes d'exclusion pour optimiser les requêtes
- Disponible depuis longtemps

L'ancienne méthode de partitionnement dans PostgreSQL se base sur un contournement de la fonctionnalité d'héritage. L'idée est de créer des tables filles d'une table parent par le biais de l'héritage. De ce fait, une lecture de la table mère provoquera une lecture des données des tables filles. Un ajout ultérieur à PostgreSQL a permis de faire en sorte que certaines tables filles ne soient pas lues si une contrainte CHECK permet de s'assurer qu'elles ne contiennent pas les données recherchées. Les lectures sont donc assurées par le biais de l'optimiseur.

LIMITATIONS DE L'ANCIEN PARTITIONNEMENT

- maintenance fastidieuse
- performance dégradée à cause du trigger
- pas de contrainte d'unicité globale

Il n'en va pas de même pour les écritures. Une insertion dans la table mère n'est pas redirigée automatiquement dans la bonne table fille. Pour cela, il faut ajouter un trigger qui annule l'insertion sur la table mère pour la réaliser sur la bonne table fille. Les mises à jour sont gérées tant qu'on ne met pas à jour les colonnes de la clé de partitionnement. Enfin, les suppressions sont gérées correctement de façon automatique.

Tout ceci génère un gros travail de mise en place. La maintenance n'est pas forcément plus aisée, car il est nécessaire de s'assurer que les partitions sont bien créées en avance, à moins de laisser ce travail au trigger sur insertion.

D'autres inconvénients sont également présents, notamment au niveau des index. Comme il n'est pas possible de créer un index global (ie, sur plusieurs tables), il n'est pas possible d'ajouter une clé primaire globale pour la table partitionnée. En fait, toute contrainte unique est impossible.

En d'autres termes, ce contournement pouvait être intéressant dans certains cas très particuliers et il fallait bien s'assurer que cela ne générait pas d'autres soucis, notamment en termes de performances. Dans tous les autres cas, il était préférable de s'en passer.

NOUVEAU PARTITIONNEMENT

- Mise en place et administration simplifiées car intégrées au moteur
- Plus de trigger
  - insertions plus rapides
  - routage des données insérées dans la bonne partition
  - erreur si aucune partition destinataire

La version *10* apporte un nouveau système de partitionnement se basant sur de l'infrastructure qui existait déjà dans PostgreSQL.

Au niveau de la simplification de la mise en place, on peut noter qu'il n'est plus nécessaire de créer une fonction trigger et d'ajouter des triggers pour gérer les insertions et les mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions. Si les données insérées ne trouvent pas de partition cible, l'insertion est tout simplement en erreur. Du fait de ce routage automatique, les insertions se révèlent aussi plus rapides.

Aucune donnée n'est stockée dans la table partitionnée. Il est possible de le vérifier en utilisant un SELECT avec la clause `ONLY`.

CHANGEMENT DU CATALOGUE SYSTÈME

- nouvelles colonnes dans `pg_class`
- nouveau catalogue `pg_partitioned_table`

Le but est de simplifier la mise en place et l'administration des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL déjà existants, comme `CREATE TABLE` et `ALTER TABLE` , pour ajouter, attacher, et détacher des partitions.

Le catalogue `pg_class` a été modifié et indique désormais :

- si une table est une partition (dans ce cas : `relispartition = 't'`)
- si une table est partitionnée (`relkind = 'p'`) ou si elle est ordinaire (`relkind = 'r'`)
- la représentation interne des bornes de partitionnement (`relpartbound`)

Le catalogue `pg_partitioned_table` contient quant à lui les colonnes suivantes :

Colonne	Contenu
partrelid	OID de la table partitionnée référencé dans `pg_class`
partstrat	Stratégie de partitionnement ; l = par liste, r = par intervalle
partnatts	Nombre de colonnes de la clé de partitionnement
partattrs	Tableau de partnatts valeurs indiquant les colonnes de la table faisant partie de la clé de partitionnement
partclass	Pour chaque colonne de la clé de partitionnement, contient l'OID de la classe d'opérateur à utiliser
partcollation	Pour chaque colonne de la clé de partitionnement, contient l'OID du collationnement à utiliser pour le partitionnement
partexprs	Arbres d'expression pour les colonnes de la clé de partitionnement qui ne sont pas des simples références de colonne

---

NOUVEAUX ORDRES SQL

- attacher/détacher une partition
- contrainte implicite de partitionnement
- expression possible pour la clé de partitionnement
- sous-partitions possibles

Le but est de simplifier la mise en place et l'administration des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL déjà existants, comme `CREATE`

`TABLE` et `ALTER TABLE` , pour ajouter, attacher, et détacher des partitions.

EXEMPLE DE PARTITIONNEMENT PAR LISTE

- Créer une table partitionnée :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
```

- Ajouter une partition :

```
CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
```

- Détacher la partition :

```
ALTER TABLE t1 DETACH PARTITION t1_a;
```

- Attacher la partition :

```
ALTER TABLE t1 ATTACH PARTITION t1_a FOR VALUES IN (1, 2, 3);
```

Lors de l'insertion, les données sont correctement redirigées vers leurs partitions.

On peut remarquer que la table partitionnée est vide.

Si aucune partition correspondant à la clé insérée n'est trouvée, une erreur se produit.

EXEMPLE DE PARTITIONNEMENT PAR INTERVALLES

- Créer une table partitionnée :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
```

- Ajouter une partition :

```
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
```

- Détacher une partition :

```
ALTER TABLE t2 DETACH PARTITION t2_1;
```

Lors de l'insertion, les données sont correctement redirigées vers leurs partitions.

Si aucune partition correspondant à la clé insérée n'est trouvée, une erreur se produit.

CLÉ DE PARTITIONNEMENT MULTI-COLONNES

- Clé sur plusieurs colonnes acceptée
  - uniquement pour le partitionnement par intervalles
- Créer une table partitionnée avec une clé multi-colonnes :

```
CREATE TABLE t3(c1 integer, c2 text, c3 date)    PARTITION BY RANGE
(c1, c3);
```

- Ajouter une partition :

```
CREATE TABLE t3_a PARTITION of t3 FOR VALUES   FROM (1,'2017-08-10')
TO (100, '2017-08-11');
```

Quand on utilise le partitionnement par intervalles, il est possible de créer les partitions en utilisant plusieurs colonnes.

LIMITATIONS

- La table mère ne peut pas avoir de données
- La table mère ne peut pas avoir d'index
  - ni PK, ni UK, ni FK pointant vers elle
- Pas de colonnes additionnelles dans les partitions
- L'héritage multiple n'est pas permis
- Valeurs nulles acceptées dans les partitions uniquement si la table partitionnée le permet
- Pas d'insertion dans des partitions distantes

Toute donnée doit pouvoir être placée dans une partition. Dans le cas contraire, la donnée ne sera pas placée dans la table mère (contrairement au partitionnement traditionnel). À la place, une erreur sera générée :

```
ERROR:  no partition of relation "t2" found for row
```

De même, il n'est pas possible d'ajouter un index à la table mère, sous peine de voir l'erreur suivante apparaître :

```
ERROR:  cannot create index on partitioned table "t1"
```

Ceci sous-entend qu'il n'est toujours pas possible de mettre une clé primaire, et une contrainte unique sur ce type de table. De ce fait, il n'est pas non plus possible de faire pointer une clé étrangère vers ce type de table.

ET DANS LA VERSION 11 ?

- Nouvelle méthode de partitionnement par clé de hachage

PLACE À LA DÉMO !

- création de table partitionnée en 9.6 et 10
- étude des limitations
- maintenance

DÉMO PARTITIONNEMENT : CRÉATION

Nous allons étudier les différences entre la version 9.6 et la version 10 en termes d'utilisation des tables partitionnées.

Nous allons créer une table de mesure des températures suivant le lieu et la date. Nous allons partitionner ces tables pour chaque lieu et chaque mois.

Ordre de création de la table en version 9.6 :

```
CREATE TABLE meteo (
   t_id serial,
   lieu text NOT NULL,
   heure_mesure timestamp DEFAULT now(),
   temperature real NOT NULL
 );
CREATE TABLE meteo_lyon_201709 (
   CHECK ( lieu = 'Lyon'
           AND heure_mesure >= TIMESTAMP '2017-09-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-10-01 00:00:00' )
) INHERITS (meteo);
CREATE TABLE meteo_lyon_201710 (
   CHECK ( lieu = 'Lyon'
           AND heure_mesure >= TIMESTAMP '2017-10-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-11-01 00:00:00' )
) INHERITS (meteo);
CREATE TABLE meteo_nantes_201709 (
   CHECK ( lieu = 'Nantes'
           AND heure_mesure >= TIMESTAMP '2017-09-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-10-01 00:00:00' )
) INHERITS (meteo);
CREATE TABLE meteo_nantes_201710 (
   CHECK ( lieu = 'Nantes'
```

```
            AND heure_mesure >= TIMESTAMP '2017-10-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-11-01 00:00:00' )
) INHERITS (meteo);
CREATE TABLE meteo_paris_201709 (
   CHECK ( lieu = 'Paris'
            AND heure_mesure >= TIMESTAMP '2017-09-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-10-01 00:00:00' )
) INHERITS (meteo);
CREATE TABLE meteo_paris_201710 (
   CHECK ( lieu = 'Paris'
            AND heure_mesure >= TIMESTAMP '2017-10-01 00:00:00'
       AND heure_mesure < TIMESTAMP '2017-11-01 00:00:00' )
) INHERITS (meteo);
CREATE OR REPLACE FUNCTION meteo_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.lieu = 'Lyon' ) THEN
      IF ( NEW.heure_mesure >= TIMESTAMP '2017-09-01 00:00:00' AND
           NEW.heure_mesure < TIMESTAMP '2017-10-01 00:00:00' ) THEN
          INSERT INTO meteo_lyon_201709 VALUES (NEW.*);
      ELSIF ( NEW.heure_mesure >= TIMESTAMP '2017-10-01 00:00:00' AND
              NEW.heure_mesure < TIMESTAMP '2017-11-01 00:00:00' ) THEN
          INSERT INTO meteo_lyon_201710 VALUES (NEW.*);
      ELSE
        RAISE EXCEPTION 'Date non prévue dans meteo_insert_trigger(Lyon)';
      END IF;
    ELSIF ( NEW.lieu = 'Nantes' ) THEN
      IF ( NEW.heure_mesure >= TIMESTAMP '2017-09-01 00:00:00' AND
           NEW.heure_mesure < TIMESTAMP '2017-10-01 00:00:00' ) THEN
          INSERT INTO meteo_nantes_201709 VALUES (NEW.*);
      ELSIF ( NEW.heure_mesure >= TIMESTAMP '2017-10-01 00:00:00' AND
              NEW.heure_mesure < TIMESTAMP '2017-11-01 00:00:00' ) THEN
          INSERT INTO meteo_nantes_201710 VALUES (NEW.*);
      ELSE
        RAISE EXCEPTION 'Date non prévue dans meteo_insert_trigger(Nantes)';
      END IF;
    ELSIF ( NEW.lieu = 'Paris' ) THEN
      IF ( NEW.heure_mesure >= TIMESTAMP '2017-09-01 00:00:00' AND
           NEW.heure_mesure < TIMESTAMP '2017-10-01 00:00:00' ) THEN
```

```
            INSERT INTO meteo_paris_201709 VALUES (NEW.*);
        ELSIF ( NEW.heure_mesure >= TIMESTAMP '2017-10-01 00:00:00' AND
                NEW.heure_mesure < TIMESTAMP '2017-11-01 00:00:00' ) THEN
            INSERT INTO meteo_paris_201710 VALUES (NEW.*);
        ELSE
          RAISE EXCEPTION 'Date non prévue dans meteo_insert_trigger(Paris)';
        END IF;
    ELSE
        RAISE EXCEPTION 'Lieu non prévu dans meteo_insert_trigger() !';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
CREATE TRIGGER insert_meteo_trigger
    BEFORE INSERT ON meteo
    FOR EACH ROW EXECUTE PROCEDURE meteo_insert_trigger();
```

Ordre de création de la table en version 10 ;

```
CREATE TABLE meteo (
   t_id integer GENERATED BY DEFAULT AS IDENTITY,
   lieu text NOT NULL,
   heure_mesure timestamp DEFAULT now(),
   temperature real NOT NULL
 ) PARTITION BY RANGE (lieu, heure_mesure);
CREATE TABLE meteo_lyon_201709 PARTITION of meteo FOR VALUES
   FROM ('Lyon', '2017-09-01 00:00:00') TO ('Lyon', '2017-10-01 00:00:00');
CREATE TABLE meteo_lyon_201710 PARTITION of meteo FOR VALUES
   FROM ('Lyon', '2017-10-01 00:00:00') TO ('Lyon', '2017-11-01 00:00:00');
CREATE TABLE meteo_nantes_201709 PARTITION of meteo FOR VALUES
   FROM ('Nantes', '2017-09-01 00:00:00') TO ('Nantes', '2017-10-01 00:00:00');
CREATE TABLE meteo_nantes_201710 PARTITION of meteo FOR VALUES
   FROM ('Nantes', '2017-10-01 00:00:00') TO ('Nantes', '2017-11-01 00:00:00');
CREATE TABLE meteo_paris_201709 PARTITION of meteo FOR VALUES
   FROM ('Paris', '2017-09-01 00:00:00') TO ('Paris', '2017-10-01 00:00:00');
CREATE TABLE meteo_paris_201710 PARTITION of meteo FOR VALUES
   FROM ('Paris', '2017-10-01 00:00:00') TO ('Paris', '2017-11-01 00:00:00');
```

On remarque que la déclaration est bien plus facile en version 10. Comme nous le verrons le plus fastidieux est de faire évoluer la fonction trigger en version 9.6.

Voici une fonction permettant d'ajouter des entrées aléatoires dans la table :

```
CREATE OR REPLACE FUNCTION peuple_meteo()
RETURNS TEXT AS $$
DECLARE
   lieux text[] := '{}';
   v_lieu text;
   v_heure timestamp;
   v_temperature real;
   v_nb_insertions integer := 500000;
   v_insertion integer;
BEGIN
   lieux[0]='Lyon';
   lieux[1]='Nantes';
   lieux[2]='Paris';
   FOR v_insertion IN 1 .. v_nb_insertions LOOP
      v_lieu=lieux[floor((random()*3))::int];
      v_heure='2017-09-01'::timestamp
                  + make_interval(days => floor((random()*60))::int,
                              secs => floor((random()*86400))::int);
      v_temperature:=round(((random()*14))::numeric+10,2);
      IF EXTRACT(MONTH FROM v_heure) = 10 THEN
          v_temperature:=v_temperature-4;
      END IF;
      IF EXTRACT(HOUR FROM v_heure) <= 9
         OR EXTRACT(HOUR FROM v_heure) >= 20 THEN
          v_temperature:=v_temperature-5;
      ELSEIF EXTRACT(HOUR FROM v_heure) >= 12
         AND EXTRACT(HOUR FROM v_heure) <= 17 THEN
          v_temperature:=v_temperature+5;
      END IF;
      INSERT INTO meteo (lieu,heure_mesure,temperature)
        VALUES (v_lieu,v_heure,v_temperature);
   END LOOP;
   RETURN v_nb_insertions||' mesures de température insérées';
END;
$$
LANGUAGE plpgsql;
```

Insérons des lignes dans les 2 tables :

```
pg96=# EXPLAIN ANALYSE SELECT peuple_meteo();
                          QUERY PLAN
-------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=32)
         (actual time=20154.769..20154.769 rows=1 loops=1)
 Planning time: 0.031 ms
 Execution time: 20154.790 ms
(3 lignes)

pg10=# EXPLAIN ANALYSE SELECT peuple_meteo();
                          QUERY PLAN
-------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=32)
         (actual time=15823.882..15823.882 rows=1 loops=1)
 Planning time: 0.042 ms
 Execution time: 15823.920 ms
(3 lignes)
```

Nous constatons un gain de 25% en version 10 sur l'insertion de données.

DÉMO PARTITIONNEMENT : LIMITATIONS

- création d'index
- mise à jour
- insertion de données hors limite

Index

La création d'index n'est toujours pas disponible en version 10 :

```
pg10=# CREATE INDEX meteo_heure_mesure_idx ON meteo (heure_mesure);
ERROR:  cannot create index on partitioned table "meteo"
```

Il est donc toujours impossible de créer une clé primaire, une contrainte unique ou une contrainte d'exclusion pouvant s'appliquer sur toutes les partitions.

De ce fait, il est également impossible de référencer via une clé étrangère une table partitionnée.

Il est cependant possible de créer des index sur chaque partition fille, comme avec la version 9.6 :

```
pg10=# CREATE INDEX meteo_lyon_201710_heure_idx
  ON meteo_lyon_201710 (heure_mesure);
CREATE INDEX
```

Mise à jour

Une mise à jour qui déplacerait des enregistrements d'une partition à une autre n'est pas possible par défaut en version 10 :

```
pg10=# UPDATE meteo SET lieu='Nantes' WHERE lieu='Lyon';
ERROR:  new row for relation "meteo_lyon_201709" violates partition constraint
DÉTAIL : Failing row contains (5, Nantes, 2017-09-15 05:09:23, 9.43).
```

FIXME : créer une fonction ??

Insertion de données hors limite

FIXME test insertion de données hors limite puis avec l'utilisation de MINVALUE

Le partitionnement en version 10 permet de déclarer

```
CREATE TABLE meteo_lyon_ancienne PARTITION of meteo FOR VALUES
   FROM ('Lyon', MINVALUE) TO ('Lyon', '2017-09-01 00:00:00');
CREATE TABLE meteo_nantes_ancienne PARTITION of meteo FOR VALUES
   FROM ('Nantes', MINVALUE) TO ('Nantes', '2017-09-01 00:00:00');
CREATE TABLE meteo_paris_ancienne PARTITION of meteo FOR VALUES
   FROM ('Paris', MINVALUE) TO ('Paris', '2017-09-01 00:00:00');
```

DÉMO PARTITIONNEMENT : MAINTENANCE

Avec les tables partitionnées via l'héritage, il était nécessaire de lister toutes les tables partitionnées pour effectuer des tâches de maintenance.

```
pg96=# SELECT 'VACUUM ANALYZE '||relname AS operation
  FROM pg_stat_user_tables WHERE relname LIKE 'meteo_%';
            operation
-----------------------------------
 VACUUM ANALYZE meteo_lyon_201709
 VACUUM ANALYZE meteo_lyon_201710
 VACUUM ANALYZE meteo_nantes_201709
 VACUUM ANALYZE meteo_nantes_201710
 VACUUM ANALYZE meteo_paris_201709
 VACUUM ANALYZE meteo_paris_201710
```

```
(6 lignes)

pg96=# \gexec
VACUUM
VACUUM
VACUUM
VACUUM
VACUUM
VACUUM
```

Avec la version 10, il est maintenant possible d'effectuer des opérations de VACUUM et ANALYSE sur toutes les tables partitionnées via la table mère.

```
pg10=# VACUUM ANALYZE meteo;
VACUUM
pg10=# SELECT now() AS date,relname,last_vacuum,last_analyze
  FROM pg_stat_user_tables WHERE relname LIKE 'meteo_nantes%';
-[ RECORD 1 ]+------------------------------
date         | 2017-09-01 08:39:02.052168-04
relname      | meteo_nantes_201709
last_vacuum  | 2017-09-01 08:38:54.068208-04
last_analyze | 2017-09-01 08:38:54.068396-04
-[ RECORD 2 ]+------------------------------
date         | 2017-09-01 08:39:02.052168-04
relname      | meteo_nantes_201710
last_vacuum  | 2017-09-01 08:38:54.068482-04
last_analyze | 2017-09-01 08:38:54.068665-04
```

RÉPLICATION LOGIQUE

- Petit rappel sur la réplication physique
- Qu'est-ce que la réplication logique ?
- Fonctionnement
- Limitations
- Exemples

RÉPLICATION PHYSIQUE

- Réplication de toute l'instance
  - au niveau bloc
  - par rejeu des journaux de transactions
- Quelques limitations :
  - intégralité de l'instance
  - même architecture (x86, ARM…)
  - même version majeure
  - pas de requête en écriture sur le secondaire

Dans le cas de la réplication dite « physique », le moteur ne réplique pas les requêtes, mais le résultat de celles-ci, et plus précisément les modifications des blocs de données. Le serveur secondaire se contente de rejouer les journaux de transaction.

Cela impose certaines limitations. Les journaux de transactions ne contenant comme information que le nom des fichiers (et pas les noms et / ou type des objets SQL impliqués), il n'est pas possible de ne rejouer qu'une partie. De ce fait, on réplique l'intégralité de l'instance.

La façon dont les données sont codées dans les fichiers dépend de l'architecture matérielle (32 / 64 bits, little / big endian) et des composants logiciels du système d'exploitation (tri des données, pour les index). De ceci, il en découle que chaque instance du cluster de réplication doit fonctionner sur un matériel dont l'architecture est identique à celle des autres instances et sur un système d'exploitation qui trie les données de la même façon.

Les versions majeures ne codent pas forcément les données de la même façon, notamment dans les journaux de transactions. Chaque instance du cluster de réplication doit donc être de la même version majeure.

Enfin, les serveurs secondaires sont en lecture seule. Cela signifie (et c'est bien) qu'on ne

peut pas insérer / modifier / supprimer de données sur les tables répliquées. Mais on ne peut pas non plus ajouter des index supplémentaires ou des tables de travail, ce qui est bien dommage dans certains cas.

RÉPLICATION LOGIQUE - PRINCIPE

- Réutilisation de l'infrastructure existante
  - réplication en flux
  - slots de réplication
- Réplique les changements sur une seule base de données
  - d'un ensemble de tables défini
- Uniquement INSERT / UPDATE / DELETE
  - pas les DDL, ni les TRUNCATE

Contrairement à la réplication physique, la réplication logique ne réplique pas les blocs de données. Elle décode le résultat des requêtes qui est transmis au secondaire. Celui-ci applique les modifications SQL issues du flux de réplication logique.

La réplication logique utilise un système de publication / abonnement avec un ou plusieurs abonnés qui s'abonnent à une ou plusieurs publications d'un nœud particulier.

Une publication peut être définie sur n'importe quel serveur primaire de réplication physique. Le nœud sur laquelle la publication est définie est nommé éditeur. Le nœud où un abonnement a été défini est nommé abonné.

Une publication est un ensemble de modifications généré par une table ou un groupe de table. Chaque publication existe au sein d'une seule base de données.

Un abonnement définit la connexion à une autre base de données et un ensemble de publications (une ou plus) auxquelles l'abonné veut souscrire.

FONCTIONNEMENT

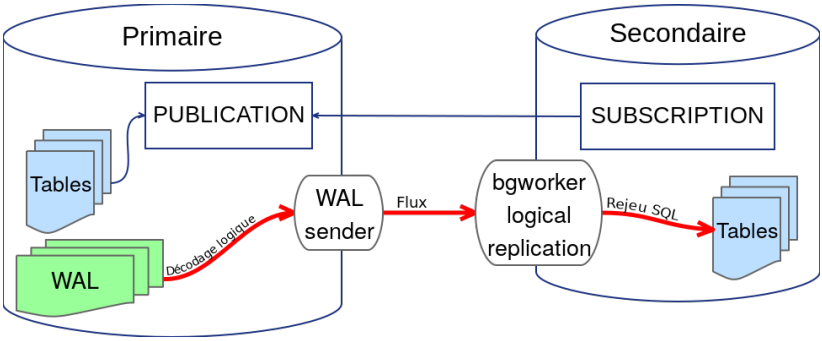

Figure 1: Schéma du fonctionnement de la réplication logique

Schéma obtenu sur blog.anayrat.info.

Source : Adrien Nayrat - Série sur la réplication logique

- Une publication est créée sur le serveur éditeur
- L'abonné souscrit à cette publication, c'est un « souscripteur »
- Un processus spécial est lancé : le « bgworker logical replication ». Il va se connecter à un slot de réplication sur le serveur éditeur
- Le serveur éditeur va procéder à un décodage logique des journaux de transaction pour extraire les résultats des ordres SQL
- Le flux logique est transmis à l'abonné qui l'applique sur les tables

---

LIMITATIONS

- Non répliqué :
  - Schéma
  - Séquences
  - *Large objects*
- Pas de publication des tables parents du partitionnement
- Ne convient pas comme fail-over
- Contrainte d'unicité nécessaire pour `UPDATE` et `DELETE`

Le schéma de la base de données ainsi que les commandes `DDL` ne sont pas répliquées, y compris l'ordre `TRUNCATE`. Le schéma initial peut être créé en utilisant par exemple

`pg_dump --schema-only`. Il faudra dès lors répliquer manuellement les changements de structure.

Il n'est pas obligatoire de conserver strictement la même structure des deux côtés. Afin de conserver sa cohérence, la réplication s'arrêtera en cas de conflit.

Il est d'ailleurs nécessaire d'avoir des contraintes de type `PRIMARY KEY` ou `UNIQUE` et `NOT NULL` pour permettre la propagation des ordres `UPDATE` et `DELETE`.

Les triggers des tables abonnées ne seront pas déclenchés par les modifications reçues via la réplication.

En cas d'utilisation du partitionnement, il n'est pas possible d'ajouter des tables parents dans la publication.

Les séquences et *large objects* ne sont pas répliqués.

De manière générale, il serait possible d'utiliser la réplication logique en cas de fail-over en propageant manuellement les mises à jour de séquences et de schéma. La réplication physique est cependant plus appropriée pour cela.

La réplication logique vise d'autres objectifs, tels que la génération de rapports ou la mise à jour de version majeure de PostgreSQL.

DÉMO RÉPLICATION LOGIQUE : PUBLICATION

Nous allons créer une base de donnée `souscription` et y répliquer de façon logique la table partitionnée `meteo` créée précédemment.

Tout d'abord, nous devons nous assurer que notre instance est configurée pour permettre la réplication logique. Le paramètre `wal_level` doit être fixé à `logical` dans le fichier `postgresql.conf`. Ce paramètre a un impact sur les informations stockées dans les fichiers WAL, un redémarrage de l'instance est donc nécessaire en cas de changement.

Ensuite, créons la base de donnée `souscription` dans notre instance 10 :

```
psql -c "CREATE DATABASE souscription"
```

Dans la base de données `pg10`, nous allons tenter de créer la publication sur la table partitionnée :

```
pg10=# CREATE PUBLICATION local_publication FOR TABLE meteo;
ERROR:  "meteo" is a partitioned table
DÉTAIL : Adding partitioned tables to publications is not supported.
ASTUCE : You can add the table partitions individually.
```

Comme précisé dans le cours, il est impossible de publier les tables parents. Nous allons devoir publier chaque partition. Nous partons du principe que seul le mois de septembre nous intéresse :

```
CREATE PUBLICATION local_publication FOR TABLE
  meteo_lyon_201709, meteo_nantes_201709, meteo_paris_201709;
SELECT * FROM pg_create_logical_replication_slot('local_souscription','pgoutput');
```

Comme nous travaillons en local, il est nécessaire de créer le slot de réplication manuellement. Il faudra créer la souscription de manière à ce qu'elle utilise le slot de réplication que nous venons de créer. Si ce n'est pas fait, nous nous exposons à un blocage de l'ordre de création de souscription. Ce problème n'arrive pas lorsque l'on travaille sur deux instances séparées.

DÉMO RÉPLICATION LOGIQUE : SOUSCRIPTION

Après avoir géré la partie publication, passons à la partie souscription.

Nous allons maintenant créer un utilisateur spécifique qui assurera la réplication logique :

```
$ createuser --replication replilogique
```

Lui donner un mot de passe et lui permettre de visualiser les données dans la base `pg10` :

```
pg10=# ALTER ROLE replilogique PASSWORD 'pwd';
ALTER ROLE
pg10=# GRANT SELECT ON ALL TABLES IN SCHEMA public TO replilogique;
GRANT
```

Nous devons également lui autoriser l'accès dans le fichier `pg_hba.conf` de l'instance :

```
host    all             replilogique    127.0.0.1/32            md5
```

Sans oublier de recharger la configuration :

```
pg10=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 ligne)
```

Dans la base de données `souscription`, créer les tables à répliquer :

```
CREATE TABLE meteo (
   t_id integer GENERATED BY DEFAULT AS IDENTITY,
   lieu text NOT NULL,
   heure_mesure timestamp DEFAULT now(),
   temperature real NOT NULL
 ) PARTITION BY RANGE (lieu, heure_mesure);
CREATE TABLE meteo_lyon_201709 PARTITION of meteo FOR VALUES
   FROM ('Lyon', '2017-09-01 00:00:00') TO ('Lyon', '2017-10-01 00:00:00');
CREATE TABLE meteo_nantes_201709 PARTITION of meteo FOR VALUES
   FROM ('Nantes', '2017-09-01 00:00:00') TO ('Nantes', '2017-10-01 00:00:00');
CREATE TABLE meteo_paris_201709 PARTITION of meteo FOR VALUES
   FROM ('Paris', '2017-09-01 00:00:00') TO ('Paris', '2017-10-01 00:00:00');
```

Nous pouvons maintenant créer la souscription à partir de la base de donnée `souscription` :

```
souscription=# CREATE SUBSCRIPTION souscription
 CONNECTION 'host=127.0.0.1 port=5432 user=replilogique dbname=pg10 password=pwd'
 PUBLICATION local_publication with (create_slot=false,slot_name='local_souscription');
CREATE SUBSCRIPTION
```

Vérifier que les données ont bien été répliquées sur la base `souscription`.

N'hésitez pas à vérifier dans les logs dans le cas où une opération ne semble pas fonctionner.

DÉMO RÉPLICATION LOGIQUE : MODIFICATION DES DONNÉES

Maintenant que la réplication logique est établie, nous allons étudier les possibilités offertes par cette dernière.

Contrairement à la réplication physique, il est possible de modifier les données de l'instance en souscription :

```
souscription=# SELECT * FROM meteo WHERE t_id=1;
 t_id | lieu |    heure_mesure     | temperature
------+------+---------------------+-------------
    1 | Lyon | 2017-09-24 04:10:59 |       18.59
(1 ligne)

souscription=# DELETE FROM meteo WHERE t_id=1;
```

```
DELETE 1
souscription=# SELECT * FROM meteo WHERE t_id=1;
 t_id | lieu | heure_mesure | temperature
------+------+--------------+-------------
(0 ligne)
```

Cette suppression n'a pas eu d'impact sur l'instance principale :

```
pg10=# SELECT * FROM meteo WHERE t_id=1;
 t_id | lieu |    heure_mesure     | temperature
------+------+---------------------+-------------
    1 | Lyon | 2017-09-24 04:10:59 |       18.59
(1 ligne)
```

Essayons maintenant de supprimer ou modifier des données de l'instance principale :

```
pg10=# UPDATE meteo SET temperature=25 WHERE temperature<15;
ERROR:  cannot update table "meteo_lyon_201709" because it does not have
        replica identity and publishes updates
ASTUCE : To enable updating the table, set REPLICA IDENTITY using ALTER TABLE.

pg10=# DELETE FROM meteo WHERE temperature < 15;
ERROR:  cannot delete from table "meteo_lyon_201709" because it does not have
        replica identity and publishes deletes
ASTUCE : To enable deleting from the table, set REPLICA IDENTITY using ALTER TABLE.
```

Il nous faut créer un index unique sur les tables répliquées puis déclarer cet index comme `REPLICA IDENTITY` dans la base de donnée `pg10` :

```
CREATE UNIQUE INDEX meteo_lyon_201709_pkey ON meteo_lyon_201709 (t_id);
CREATE UNIQUE INDEX meteo_nantes_201709_pkey ON meteo_nantes_201709 (t_id);
CREATE UNIQUE INDEX meteo_paris_201709_pkey ON meteo_paris_201709 (t_id);
ALTER TABLE meteo_lyon_201709 REPLICA IDENTITY USING INDEX meteo_lyon_201709_pkey;
ALTER TABLE meteo_nantes_201709 REPLICA IDENTITY USING INDEX meteo_nantes_201709_pkey;
ALTER TABLE meteo_paris_201709 REPLICA IDENTITY USING INDEX meteo_paris_201709_pkey;
```

Vérifions l'effet de nos modifications :

```
pg10=# UPDATE meteo SET temperature=25 WHERE temperature<15;
UPDATE 150310
pg10=# SELECT count(*) FROM meteo WHERE temperature<15;
 count
-------
```

```
     0
(1 ligne)
```

La mise à jour a été possible sur la base de données principale. Quel effet cela a-t-il produit sur la base de données répliquée :

```
souscription=# SELECT count(*) FROM meteo WHERE temperature<15;
 count
-------
 75291
(1 ligne)
```

La mise à jour ne semble pas s'être réalisée. Vérifions dans les logs applicatifs :

```
LOG:  logical replication apply worker for subscription "souscription"
      has started
LOG:  starting logical decoding for slot "local_souscription"
DETAIL:  streaming transactions committing after 0/F4FFF450,
         reading WAL from 0/F33E5B18
LOG:  logical decoding found consistent point at 0/F33E5B18
DETAIL:  There are no running transactions.
ERROR:  logical replication target relation "public.meteo_lyon_201709" has
        neither REPLICA IDENTITY index nor PRIMARY KEY and published relation
    does not have REPLICA IDENTITY FULL
LOG:  could not send data to client: Connection reset by peer
CONTEXT:  slot "local_souscription", output plugin "pgoutput", in the change
          callback, associated LSN 0/F33EC9B0
LOG:  worker process: logical replication worker for subscription 17685
      (PID 3743) exited with exit code 1
```

Les ordres DDL ne sont pas transmis avec la réplication logique. Nous devons toujours penser à appliquer tous les changements effectués sur l'instance principale sur l'instance en réplication.

```
souscription=# CREATE UNIQUE INDEX meteo_lyon_201709_pkey
   ON meteo_lyon_201709 (t_id);
CREATE INDEX
souscription=# CREATE UNIQUE INDEX meteo_nantes_201709_pkey
   ON meteo_nantes_201709 (t_id);
CREATE INDEX
souscription=# CREATE UNIQUE INDEX meteo_paris_201709_pkey
   ON meteo_paris_201709 (t_id);
CREATE INDEX
```

```
souscription=# ALTER TABLE meteo_lyon_201709 REPLICA IDENTITY
   USING INDEX meteo_lyon_201709_pkey;
ALTER TABLE
souscription=# ALTER TABLE meteo_nantes_201709 REPLICA IDENTITY
   USING INDEX meteo_nantes_201709_pkey;
ALTER TABLE
souscription=# ALTER TABLE meteo_paris_201709 REPLICA IDENTITY
   USING INDEX meteo_paris_201709_pkey;
ALTER TABLE
```

La réplication logique est de nouveau fonctionnelle. Cependant les modifications effectuées sur la base principale sont dorénavant perdues :

```
souscription=# SELECT count(*) FROM meteo WHERE temperature<15;
 count
-------
 75291
(1 ligne)
```

Réappliquons la modification sur la base pg10 :

```
pg10=# UPDATE meteo SET temperature=25 WHERE temperature<15;
UPDATE 0
```

Vérifions l'effet sur la base de donnée répliquée :

```
souscription=# SELECT count(*) FROM meteo WHERE temperature<15;
 count
-------
     0
(1 ligne)

souscription=# SELECT count(*) FROM meteo WHERE temperature=25;
 count
-------
 75291
(1 ligne)
```

DES QUESTIONS ?

Your browser does not support the video tag.