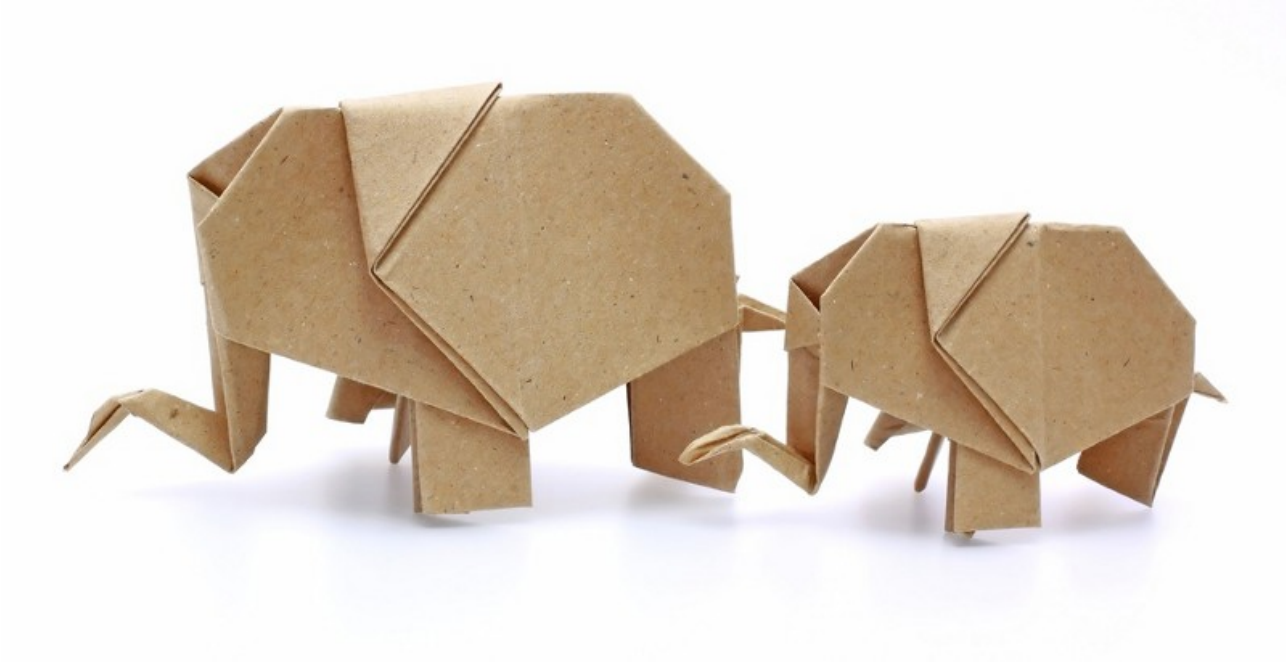
DALIBO

L'expertise PostgreSQL

Réplication logique avec PostgreSQL 9.4

Table des matières

Réplication logique



1 Licence des slides

- Creative Common BY-NC-SA
- Vous êtes libre
  - de partager
  - de modifier
- Sous les conditions suivantes
  - Attribution
  - Non commercial
  - Partage dans les mêmes conditions

2 Auteur

- Thomas Reiss
- Travail
  - Consultant PostgreSQL chez Dalibo
  - email: thomas.reiss@dalibo.com
  - twitter: @frostsct1

3 Historique

- Replication asynchrone en streaming : 9.0
- Replication synchrone : 9.1
- Replication en cascade : 9.2
- Changement de maître en streaming : 9.3

Jusqu'en 9.3, la réplication concerne tous les changements effectués dans la base de données. On réplique alors tous les changements de toutes les tables, dans toutes les bases de données d'une instance. L'esclave doit être une copie physique du maître, qui est mis à niveau par rejeu des journaux de transactions.

Pour ne répliquer que les changements réalisés sur certaines tables ou sur une seule base de données, il fallait utiliser un outil comme Slony ou utiliser des solutions à base de triggers. Ces solutions avaient un impact sur les performances de la base de données PostgreSQL.

3.1 Avant PostgreSQL 9.4

- Réplication de toute l'instance
  - physique: au niveau bloc
  - par rejeu des journaux de transactions

4 Nouveautés de la 9.4

- Réplication logique
- Slots de réplication

5 "Réplication" logique

- Les changements sur une seule base de données
  - toutes les tables
- Uniquement INSERT/UPDATE/DELETE
- Pas les DDL
  - attention à TRUNCATE
- Pas les large objects
  - `pg_largeobject` fait partie du catalogue système
  - les OID ne sont pas transposables.

6 Notions

- LSN, **L**ogical **S**equence **N**umber
  - pointeur vers un emplacement dans les journaux de transactions
  - i.e. où l'on écrit dans le flux des WAL
  - c'est un nouveau type de données en 9.4 (`pg_lsn`)
  - -> opérateurs particuliers

Documentation du type pg_lsn.

7 Slots de réplication

- Slot :
  - PostgreSQL conserve les WAL utiles à un esclave
  - chaque esclave peut avoir le sien
  - réplication physique et logique

7.1 Avantages / Inconvénients

- Plusieurs esclaves indépendants
  - conserve l'état d'un slot
- Un seul consommateur par slot
- Rétention des WAL utiles
  - Attention à leur accumulation !

Les esclaves en réplication logique reçoivent le flux de modification indépendamment des autres.

Conserve l'état d'un slot en cas de rupture de la communication entre le maître et l'esclave : dernière position lue.

Comme le slot connait la dernière position lue dans le flux des WAL (LSN, logical sequence number), PostgreSQL sait déterminer quels sont les WAL qui ne doivent pas être supprimés. Si un client reste déconnecté trop longtemps, les WAL vont s'accumuler dans le répertoire `$PGDATA/pg_xlog`. Il faut être attentif à la gestion de ces slots de réplication, sinon la base risque de s'arrêter brutalement par manque d'espace disque.

7.2 Configuration

- `max_replication_slots`
  - max number of replication slots.
  - (change requires restart)

Le paramètre est modifié dans le `postgresql.conf` :

```
max_replication_slots = 4
```

Le changement de ce paramètre nécessite un redémarrage de PostgreSQL :

```
pg_ctl restart
```

7.3 Créer un slot

- Appel de fonctions
  - `pg_create_physical_replication_slot(`*nom*`)`
  - `pg_create_logical_replication_slot(`*nom*`, `*plugin*`)`
- En ligne de commande
  - `pg_recvlogical … -S` *nom* `–create`

Les slots de réplication ne sont pas créés automatiquement. Il faut les créer manuellement à l'aide de l'une des deux fonctions proposées, en fonction de l'usage.

La création d'un slot de réplication physique ne nécessite que de spécifier le nom du slot, qui doit être unique.

Création d'un slot de réplication logique nommé `json1` et utilisant le plugin wal2json :

```
postgres=# SELECT * FROM pg_create_logical_replication_slot('json1', 'wal2json');
 slotname | xlog_position
----------+---------------
 json1    | 0/3AAB79F0
(1 row)
```

Un message apparait dans les traces de PostgreSQL :

```
STATEMENT:  SELECT pg_create_logical_replication_slot('json1', 'wal2json');
LOG:  logical decoding found consistent point at 0/3AAB79F0
DETAIL:  running xacts with xcnt == 0
```

La vue `pg_replication_slots` montre une ligne indiquant l'état du slot :

```
postgres=# SELECT * FROM pg_replication_slots ;
-[ RECORD 1 ]+--------------
 slot_name    | json1
 plugin       | wal2json
 slot_type    | logical
 datoid       | 12175
 database     | postgres
 active       | f
 xmin         |
 catalog_xmin | 954
 restart_lsn  | 0/3AAB79F0
```

Le slot de réplication logique est créé sur la base postgres.

7.4 Supprimer un slot

- pg_drop_replication_slot(*nom*)

La fonction ne retourne aucun résultat :

```
postgres=# SELECT pg_drop_replication_slot('json1');
-[ RECORD 1 ]------------+-
 pg_drop_replication_slot |
```

Mais le slot n'est plus enregistré :

```
postgres=# SELECT * FROM pg_replication_slots ;
(No rows)
```

7.5 Supervision

- pg_replication_slots
  - slot_name
  - plugin
  - slot_type : physical ou logical
  - datoid/database
  - active
  - xmin
  - catalog_xmin
  - **restart_lsn** (pg_xlog_location_diff)

```
postgres=# SELECT * FROM pg_replication_slots ;
-[ RECORD 1 ]+--------------
 slot_name     | json1
 plugin        | wal2json
 slot_type     | logical
 datoid        | 12175
 database      | postgres
 active        | f
 xmin          |
```

```
 catalog_xmin | 954
 restart_lsn  | 0/3AAB79F0
```

La requête suivante permet d'afficher une information sur le delta entre la position courante du slot et la position courante de PostgreSQL dans les WAL. Ici, PostgreSQL a écrit 56 octets dans les WAL depuis le dernier accès au slot `sql1` :

```
postgres=# SELECT slot_name, slot_type, active, pg_current_xlog_location() - restart_lsn AS
retention FROM pg_replication_slots ;
 slot_name | slot_type | active | retention
-----------+-----------+--------+-----------
 sql1      | logical   | t      |        56
 json1     | logical   | f      |     15752
(2 rows)
```

8 Réplication logique

- Fonctionnalité majeure
- Uniquement de l'infrastructure
  - décodage des WAL
  - sous-ensemble de BiDirectional Replication
- Uniquement les changements sur les données
  - pas les DDL, ni TRUNCATE

8.1 Configuration

- `wal_level = logical`
- `max_wal_senders`
- `max_replication_slots`
- `pg_hba.conf`

Un nouveau niveau `wal_level` est arrivé: `logical`. Il permet d'inscrire des informations supplémentaires dans les WAL. Ces informations vont permettre de décoder les informations contenues dans ces fichiers par PostgreSQL. Cependant, le fait d'écrire ces informations supplémentaires aura un impact sur le volume de données écrit dans les WAL.

La modification de ce paramètre est suffisant pour utiliser l'interface SQL. En revanche, il faut revoir les paramètres `max_wal_senders` et `max_replication_slots` pour pouvoir bénéficier de la réplication logique via le protocole de réplication en streaming. Le fichier

`pg_hba.conf` devra également être ajusté en conséquence.

8.2 Module de sortie

- Module chargé par `postmaster`
  - décode le flux de données
  - sortie dans un format particulier, en texte
  - utilise une API spécifique
- Plusieurs modules disponibles
  - test_decoding
  - wal2json
  - decoder_raw

Michael Paquier est à l'origine de décodeur `decoder_raw`. Il permet de recréer les ordres SQL, ce qui permet de ré-appliquer les changements sur une autre base de données PostgreSQL. Attention toutefois, il ne s'agit pas d'une solution de réplication mature. Elle est le fruit d'un travail de seulement 2 heures. Michael décrit son plugin et l'API de PostgreSQL sur le billet de blog suivant : http://michael.otacoo.com/postgresql-2/postgres-9-4-feature-highlight-output-plugin-logical-replication/.

En complément, la documentation officielle de PostgreSQL décrit cette API créée pour l'occasion : Logical Decoding Output Plugins.

8.3 Capture des modifications

- Fonctions SQL
- Protocole de réplication
  - client `pg_recvlogical`

Plusieurs fonctions accessibles en SQL sont proposées : Fonction de réplication. Ces fonctions permettent de récupérer les changements selon le format spécifique au module de sortie.

En complément de ces fonctions, le protocole de réplication de PostgreSQL a été enrichie pour permettre à des clients de recevoir le flux de modification en streaming.

8.4 Fonctions SQL

- Sortie texte
  - `pg_logical_slot_peek_changes`
  - `pg_logical_slot_get_changes`
- Sortie binaire (bytea)
  - `pg_logical_slot_peek_binary_changes`
  - `pg_logical_slot_get_binary_changes`

Il est également possible de passer des options particulières au plugin :

```
postgres=# SELECT pg_create_logical_replication_slot('sql1', 'decoder_raw');
 pg_create_logical_replication_slot
------------------------------------
 (sql1,0/3AAD30C0)
(1 row)

postgres=# TRUNCATE t1;
TRUNCATE TABLE

postgres=# INSERT INTO t1 SELECT generate_series(1, 10);
INSERT 0 10
```

Récupération des changements sans option :

```
postgres=# SELECT * FROM pg_logical_slot_peek_changes('sql1', NULL, NULL);
  location   | xid |                  data
-------------+-----+----------------------------------------
 0/3AAD6A28 | 995 |
 0/3AAD9A30 | 995 |
 0/3AAD9A30 | 996 |
 0/3AAD9A30 | 996 | INSERT INTO public.t1 (i) VALUES (1);
 0/3AAD9A70 | 996 | INSERT INTO public.t1 (i) VALUES (2);
 0/3AAD9AB0 | 996 | INSERT INTO public.t1 (i) VALUES (3);
 0/3AAD9AF0 | 996 | INSERT INTO public.t1 (i) VALUES (4);
 0/3AAD9B30 | 996 | INSERT INTO public.t1 (i) VALUES (5);
 0/3AAD9B70 | 996 | INSERT INTO public.t1 (i) VALUES (6);
 0/3AAD9BB0 | 996 | INSERT INTO public.t1 (i) VALUES (7);
 0/3AAD9BF0 | 996 | INSERT INTO public.t1 (i) VALUES (8);
 0/3AAD9C30 | 996 | INSERT INTO public.t1 (i) VALUES (9);
 0/3AAD9C70 | 996 | INSERT INTO public.t1 (i) VALUES (10);
 0/3AAD9CF0 | 996 |
(14 rows)
```

Récupération des changements avec l'option `include_transaction` à on :

```
postgres=# SELECT * FROM pg_logical_slot_peek_changes('sql1', NULL, NULL, 'include_transaction',
'on');
  location   | xid |                  data
-------------+-----+----------------------------------------
 0/3AAD6A28 | 995 | BEGIN;
 0/3AAD9A30 | 995 | COMMIT;
```

```
 0/3AAD9A30 | 996 | BEGIN;
 0/3AAD9A30 | 996 | INSERT INTO public.t1 (i) VALUES (1);
 0/3AAD9A70 | 996 | INSERT INTO public.t1 (i) VALUES (2);
 0/3AAD9AB0 | 996 | INSERT INTO public.t1 (i) VALUES (3);
 0/3AAD9AF0 | 996 | INSERT INTO public.t1 (i) VALUES (4);
 0/3AAD9B30 | 996 | INSERT INTO public.t1 (i) VALUES (5);
 0/3AAD9B70 | 996 | INSERT INTO public.t1 (i) VALUES (6);
 0/3AAD9BB0 | 996 | INSERT INTO public.t1 (i) VALUES (7);
 0/3AAD9BF0 | 996 | INSERT INTO public.t1 (i) VALUES (8);
 0/3AAD9C30 | 996 | INSERT INTO public.t1 (i) VALUES (9);
 0/3AAD9C70 | 996 | INSERT INTO public.t1 (i) VALUES (10);
 0/3AAD9CF0 | 996 | COMMIT;
(14 rows)
```

8.5 Streaming

- Extension du protocole de réplication
  - capture des changements en streaming
  - client `pg_recvlogical`

`pg_recvlogical` peut créer un slot de réplication :

```
pg_recvlogical -P wal2json -S demo1 -d tpc --create
```

Le message suivant apparaît dans les traces :

```
LOG:  logical decoding found consistent point at 0/1B45A608
DETAIL:  running xacts with xcnt == 0
LOG:  exported logical decoding snapshot: "00000392-1" with 0 xids
```

La vue `pg_replication_slots` montre bien le nouveau slot :

```
tpc=# SELECT * FROM pg_replication_slots ;
-[ RECORD 1 ]+-----------
 slot_name     | demo1
 plugin        | wal2json
 slot_type     | logical
 datoid        | 16384
 database      | tpc
 active        | f
 xmin          |
 catalog_xmin  | 914
 restart_lsn   | 0/1B45A608
```

Dans un autre terminal, lancer `pg_recvlogical` en lui indiquant d'écrire sur la sortie standard :

```
pg_recvlogical -S demo1 -d tpc -f - --start
```

Création d'une table :

```
tpc=# CREATE TABLE t4 (i integer);
CREATE TABLE
```

Il y a bien une transaction, mais elle ne fournit pas de données :

```
{
        "xid": 915,
        "change": [
        ]
}
```

Seul un trigger sur évènement permettra de consigner ce changement.

Un `INSERT` montre que les changements sont décodés et restitués :

```
tpc=# INSERT INTO t4 VALUES (1);
INSERT 0 1
```

La sortie de `pg_recvlogical` montre les lignes suivantes, au format JSON du fait de l'utilisation du décodeur `wal2json` :

```
{
        "xid": 916,
        "change": [
                {
                        "kind": "insert",
                        "schema": "public",
                        "table": "t4",
                        "columnnames": ["i"],
                        "columntypes": ["int4"],
                        "columnvalues": [1]
                }
        ]
}
```

8.6 Et les UPDATE et DELETE ?

- Le décodeur permet d'obtenir les `INSERT`
  - qu'en est-il des UPDATE et DELETE ?
- Comment déterminer la clé de mise à jour ?

Un UPDATE cette fois :

```
tpc=# UPDATE t4 SET i=-i WHERE i = 1;
UPDATE 1
```

Sortie de `pg_recvlogical` :

```
{
        "xid": 917,
        "change": [
WARNING:  table "t4" without primary key or replica identity is nothing
CONTEXTE : slot "demo1", output plugin "wal2json", in the change callback, associated LSN
0/1B46FAC0
        ]
}
```

Aucune clé primaire n'a été déclarée sur la table `t4`. De ce fait, PostgreSQL refuse de décoder ce changement car il ne dispose pas de moyen d'identifier la clé de mise à jour.

La table `t4` ne possède pas de clé primaire pour identifier les lignes mises à jour :

```
postgres=# \d t4
      Table "public.t4"
 Column |  Type   | Modifiers
--------+---------+-----------
 i      | integer |
```

Comment déterminer la clé de mise à jour ? Bien que PostgreSQL soit capable de relire les données issues des WAL, le fait de permettre de relire de tels changements est dangereux pour le système cible car la mise à jour ne devra toucher que la ligne concernée, pas les autres. Sans clé, c'est impossible.

9 Clés de mises à jour

- Identifier de manière sûr les clés de mises à jour :
  - clé primaire
  - autre clé ?
- REPLICA IDENTITY
  - *Identity : The characteristics determining who or what a person or thing is*

9.1 REPLICA IDENTITY

- DEFAULT
- INDEX
- FULL
- NOTHING

La table `pg_class` a été modifiée pour tenir compte de cette nouvelle fonctionnalité. La colonne `relreplident` indique le type d'identité de la table. Il est aussi positionné sur les index composants une identité.

Dans le cas où l'identité utilise un index, la colonne de type booléen `indisreplident` de la table `pg_index` indique que cet index sert à l'identification des valeurs des clés.

La requête suivante permet de connaître les identités de réplication positionnés dans une base de données :

```
SELECT n.nspname AS schemaname, c.relname AS tablename,
       CASE c.relreplident
         WHEN 'i' THEN 'USING INDEX'
         WHEN 'n' THEN 'NOTHING'
         WHEN 'd' THEN 'DEFAULT'
         WHEN 'f' THEN 'FULL'
       END AS identity,
       ii.relname AS indexname,
       string_agg(a.attname::text, ', ' ORDER BY attnum) AS cols
  FROM pg_class c
  LEFT JOIN pg_index i
    ON (c.oid = i.indrelid AND
           (   (i.indisreplident AND c.relreplident = 'i')
            OR (i.indisprimary AND c.relreplident = 'd')))
  LEFT JOIN pg_class ii
    ON (ii.oid = i.indexrelid)
  LEFT JOIN pg_attribute a
    ON (c.oid = a.attrelid AND a.attnum = ANY (i.indkey))
  LEFT JOIN pg_namespace n
    ON (c.relnamespace = n.oid)
 WHERE n.nspname NOT IN ('pg_catalog', 'pg_toast', 'information_schema')
   AND c.relkind = 'r'
 GROUP BY n.nspname, c.relname, identity, indexname
 ORDER BY c.relname
```

Elle permet d'obtenir un résultat de la forme suivante :

```
schemaname | tablename |  identity   | indexname | cols
------------+-----------+-------------+-----------+------
 public     | t1        | FULL        |           |
 public     | t4        | USING INDEX | t4_uniq_i | i
 public     | t5        | DEFAULT     |           |
 public     | t6        | NOTHING     |           |
 public     | t7        | DEFAULT     | t7_pkey   | i
(5 rows)
```

9.2 Clé primaire

- REPLICA IDENTITY **DEFAULT**
  - utilise la clé primaire
  - enregistre l'ancienne valeur de la PK
  - équivalent à NOTHING si pas de clé primaire

9.3 Un index

- REPLICA IDENTITY **USING INDEX** *nom_index*
  - enregistre l'ancienne valeur de l'index
  - colonne NOT NULL
  - index UNIQUE

Si la table ne possède pas de clé primaire, il est possible d'utiliser un index.

On crée un index sur la colonne `i` :

```
CREATE INDEX idx_t4_i ON t4 (i);
```

On crée l'identité, mais elle retourne une erreur :

```
=# ALTER TABLE t4 REPLICA IDENTITY USING INDEX idx_t4_i;
ERROR:  cannot use non-unique index "idx_t4_i" as replica identity
```

Il faut utiliser un index assurant l'unicité des valeurs. On ajoute une contrainte unique :

```
ALTER TABLE t4 ADD CONSTRAINT t4_uniq_i UNIQUE (i);
```

On retente la création de l'identité, mais elle échoue encore une fois :

```
=# ALTER TABLE t4 REPLICA IDENTITY USING INDEX t4_uniq_i;
ERROR:  index "t4_uniq_i" cannot be used as replica identity because column "i" is nullable
```

La colonne d'IDENTITY doit avoir les mêmes propriétés qu'une clé primaire: être unique et non-nulle :

```
ALTER TABLE t4 ALTER COLUMN i SET NOT NULL;
ALTER TABLE t4 REPLICA IDENTITY USING INDEX t4_uniq_i ;
```

Cette fois ça fonctionne :

```
=# UPDATE t4 SET i = -1 WHERE i = 1;
UPDATE 1
```

Le terminal exécutant la commande `pg_recvlogical` montre les détails de la transaction :

```
{
        "xid": 1016,
        "change": [
                {
                        "kind": "update",
                        "schema": "public",
                        "table": "t4",
                        "columnnames": ["i"],
                        "columntypes": ["int4"],
                        "columnvalues": [-1],
                        "oldkeys": {
                                "keynames": ["i"],
                                "keytypes": ["int4"],
                                "keyvalues": [1]
                        }
                }
        ]
}
```

Une autre commande `pg_recvlogical` branché sur un slot `sql1` utilisant le décodeur `decoder_raw` recrée un ordre UPDATE équivalent :

```
UPDATE public.t4 SET i = -1 WHERE i = 1 ;
```

Attention toutefois, PostgreSQL est incapable de retrouver l'ordre SQL réellement exécuté. Par exemple, comme la table ne fait qu'une ligne :

```
=# UPDATE t4 SET i = round(random()*100);
UPDATE 1
```

`pg_recvlogical` montre alors la sortie suivante :

```
UPDATE public.t4 SET i = 21 WHERE i = 1 ;
```

9.4 FULL

- REPLICA IDENTITY **FULL**
  - enregistre toutes les anciennes valeurs de toutes les colonnes
  - overhead potentiellement important
  - seule solution si la table n'a pas de clé

On crée une nouvelle table t5 et on insère 5 lignes :

```
CREATE TABLE t5 (i integer, t text);
INSERT INTO t5 SELECT i, md5(random()::text) FROM generate_series(1, 5) i;
```

decoder_raw reconstitue les ordres SQL suivants :

```
INSERT INTO public.t5 (i, t) VALUES (1, '99c390ce364c35b3dec8768dc8147bbc');
INSERT INTO public.t5 (i, t) VALUES (2, '85c20658ebd94eff0eb2d6127ea52717');
INSERT INTO public.t5 (i, t) VALUES (3, '27d5937e3651677256b5b536a303e800');
INSERT INTO public.t5 (i, t) VALUES (4, '4c86855142501660bc9f41e2059ab862');
INSERT INTO public.t5 (i, t) VALUES (5, '0b0f78a4b906930431f9687e284c538a');
```

On crée l'identité :

```
ALTER TABLE t5 REPLICA IDENTITY FULL ;
```

On effectue un UPDATE :

```
UPDATE t5 SET i = i + 1 WHERE i BETWEEN 2 AND 4;
```

decoder_raw reconstitue les ordres SQL suivants :

```
UPDATE public.t5 SET i = 3, t = '85c20658ebd94eff0eb2d6127ea52717'
WHERE i = 2 AND t = '85c20658ebd94eff0eb2d6127ea52717' ;
UPDATE public.t5 SET i = 4, t = '27d5937e3651677256b5b536a303e800'
WHERE i = 3 AND t = '27d5937e3651677256b5b536a303e800' ;
UPDATE public.t5 SET i = 5, t = '4c86855142501660bc9f41e2059ab862'
WHERE i = 4 AND t = '4c86855142501660bc9f41e2059ab862' ;
```

La clé de mise à jour est bien constituée de toutes colonnes de la table t5.

Attention, cela provoque des écritures supplémentaires dans les WAL. En plus des données modifiées, PostgreSQL va également écrire l'ensemble des anciennes valeurs des colonnes de la table.

La commande pg_xlogdump montre cet overhead pour la simple table t5 :

```
$ pg_xlogdump `psql -At -c "SELECT pg_xlogfile_name(pg_current_xlog_location())"` `psql -At -c
"SELECT pg_xlogfile_name(pg_current_xlog_location())"` -x 1035
rmgr: Heap        len (rec/tot):    126/   158, tx:       1035, lsn: 0/3AB31020, prev 0/3AB30FE8,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/2 xmax 1035 ; new tid 0/6 xmax 0
rmgr: Heap        len (rec/tot):    126/   158, tx:       1035, lsn: 0/3AB310C0, prev 0/3AB31020,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/3 xmax 1035 ; new tid 0/7 xmax 0
rmgr: Heap        len (rec/tot):    126/   158, tx:       1035, lsn: 0/3AB31160, prev 0/3AB310C0,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/4 xmax 1035 ; new tid 0/8 xmax 0
rmgr: Transaction len (rec/tot):     32/    64, tx:       1035, lsn: 0/3AB31200, prev 0/3AB31160,
bkp: 0000, desc: commit: 2014-05-15 10:05:43.750905 CEST
pg_xlogdump: FATAL:  error in WAL record at 0/3AB31318: record with zero length at 0/3AB31350
```

Avec REPLICA IDENTITY NOTHING, pg_xlogdump montre que chaque enregistrement écrit dans les WAL fait 45 octets de moins (158 avec FULL contre 113 avec NOTHING) :

```
$ pg_xlogdump `psql -At -c "SELECT pg_xlogfile_name(pg_current_xlog_location())"` `psql -At -c
"SELECT pg_xlogfile_name(pg_current_xlog_location())"` -x 1037
rmgr: Heap        len (rec/tot):     81/   113, tx:       1037, lsn: 0/3AB36898, prev 0/3AB36628,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/5 xmax 1037 ; new tid 0/9 xmax 0
rmgr: Heap        len (rec/tot):     81/   113, tx:       1037, lsn: 0/3AB36910, prev 0/3AB36898,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/6 xmax 1037 ; new tid 0/10 xmax 0
rmgr: Heap        len (rec/tot):     81/   113, tx:       1037, lsn: 0/3AB36988, prev 0/3AB36910,
bkp: 0000, desc: hot_update: rel 1663/12175/16673; tid 0/7 xmax 1037 ; new tid 0/11 xmax 0
rmgr: Transaction len (rec/tot):     32/    64, tx:       1037, lsn: 0/3AB36A78, prev 0/3AB36A00,
bkp: 0000, desc: commit: 2014-05-15 10:10:12.786923 CEST
pg_xlogdump: FATAL:  error in WAL record at 0/3AB389C0: record with zero length at 0/3AB389F8
```

9.5 NOTHING

- REPLICA IDENTITY **NOTHING**
  - n'enregistre aucune ancienne valeur
  - reconstitution au gré du décodeur
  - mode par défaut sur une table sans clé primaire

Avec `REPLICA IDENTITY NOTHING`, PostgreSQL n'enregistre pas d'information complémentaire sur la clé de mise à jour dans les journaux de transactions. Il ne sera donc pas possible de reconstituer la modification telle quelle a été réalisée.

L'implémentation de ce type d'identité dépend du décodeur employé. Certains autorisent la lecture des INSERT dans la table, mais pas les UPDATE ni les DELETE, d'autres empêchent carrément toute exploitation des changements journalisés dans les WAL.

Par exemple, le décodeur `wal2json` ne permet pas de reconstituer la mise à jour et émettra une erreur :

```
{
       "xid": 1043,
       "change": [
WARNING:  table "t5" without primary key or replica identity is nothing
CONTEXT:  slot "json1", output plugin "wal2json", in the change callback, associated LSN 0/3AB427E0
       ]
}
```

10 Conclusion

- Uniquement de l'infrastructure
- Permet la capture des changements
  - pas de solution de réplication logique “clé en main”
  - peut-être en 9.5 ?
  - Slony ?
  - ETL ?

Voir aussi le projet `receiver_raw` de Michael Paquier.

11 Remerciements

- Andres Freund et Robert Haas
- Michael Paquier pour `decoder_raw` et sa disponibilité

12 Questions ?

- Sinon, je suis joignable sur mon email
  - thomas.reiss@dalibo.com