Connemara

Comment créer son propre système de réplication logique

people doc
BY ULTIMATE SOFTWARE

Qui suis-je ?

Ronan Dunklau, DBA @ PeopleDoc

- Auteur d'extensions PostgreSQL
- Quelques contributions à PostgreSQL
- DBA @ peopledoc
- PeopleDoc recrute ! https://www.people-doc.com/company/careers

Le problème

Fédération de données

- Différentes applications ont chacune leur BDD
- La BI a besoin de croiser ces informtaions
  - Première solution: FDW + matérialisation
  - ETL ? Capture du changement coûteuse

Solution

Nom de code "connemara" («Datalake»)

Utiliser la réplication logique pour féderer plusieurs bases en une seule, dans son propre schéma

- PostgreSQL gère la partie difficile (changements logiques dans les WALs)
- Les systèmes de réplication existnants n'autorisent pas de déplacer dans un autre schéma
- On a construit le notre !

Cette présentation

Pourquoi ?

- Retour d'expérience sur ce qu'il faut pour concevoir un système de réplication
- Étonnament facile
- Une fois stable et générique, nous voulons le libérer
- Peut servir d'exemple pour vos propres besoins spécifiques

Architecture

Comment fonctionne la réplication logique ?

- Les WALs contiennent des informations sur les opérations logiques (DML)
- Concepts clés:
  - Un «slot de réplication» peut être créé pour diffuser le flux de WAL et le conserver jusqu'à ce qu'il soit consommé
  - Un plugin «decoder» est associé à un slot de réplication pour envoyer les changements dans un format accessible au destinataire
  - Une «replication origin» peut être créee sur le serveur secondaire pour suivre le progrès de la réplication

Outillage PostgreSQL

Streaming logique des WAL

Nécessite un peu de changement de configuration…

- wal_level = 'logical'
- max_replication_slots suffisament élevé
- max_wal_senders suffisament élevé

Outillage PostgreSQL

Slot de réplication

- Conserve les WALs jusqu'à ce qu'il soient consommés
- Supervision de ce qui a été consommé

- Associé à un décodeur

- Créé soit:
  - En SQL: `SELECT pg_create_logical_replication_slot()`
  - Commande du protocole de réplicatrion `CREATE_REPLICATION_SLOT`
- Utilisé avec une connection de réplication:
  - `START_REPLICATION`, qui place la connection en mode `COPY_BOTH`
  - Lecture du flux, écriture du retour (position consommée)

Outillage PostgreSQL

wal2json

- Plugin de décodage logique (packagé dans le repo PGDG debian)
- Beaucoup d'options, parmi lesquelles nous utilisons:
  - include-schemas
  - include-xids
  - include-timestamps
  - write-in-chunks
  - filter-tables

Outillage PostgreSQL

Replication origin

- Permet de suivre l'avancement de la réplication
- Visibles dans la table `pg_replication_origin`
- Mise en place:
  - `pg_replication_origin_create(name)`
  - `pg_replication_origin_advance(initial_position)`
- Utilisation:
  - `pg_replication_origin_session_setup(name)`
  - `pg_replication_origin_progress(name, flushed)`
  - `pg_replication_origin_xact_setup(lsn, timestamp)`

Architecture

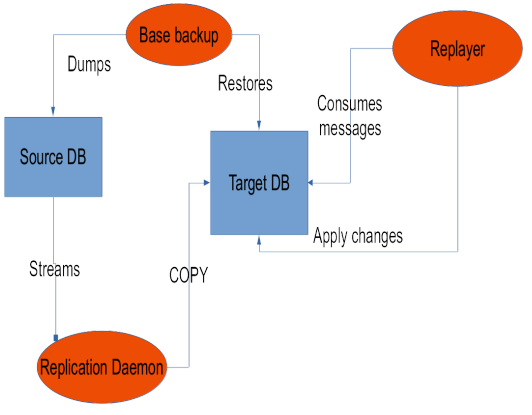
Vue d'ensemble

Différents composants, chacun avec son ensemble de problèmes

- connemara_basebackup: prend un "backup de base", cohérent, servant de base pour appliquer les WAL ensuite
- connemara_replication: Petit démon de réplication, insère les WALs dans une table de spool.
- connemara_replayer: Applique le contenu de la table de spool sur les tables

Architecture

En images...



people doc
BY ULTIMATE SOFTWARE

Premier problème: le backup de base !

En fait, plusieurs problèmes...

- Comment garantir un backup cohérent, sans perdre une transaction ?
- Comment restaurer des tables dans un schéma différent ?
- De manière sélective, on ne veut pas tout ?

Base backup: problème numéro 1

Avoir un backup cohérent: utilisation d'un replication slot

- Initialisation d'une connection de réplication logique
- Création d'un slot en utilisant le protocol `CREATE_REPLICATION_SLOT slot_name LOGICAL <output_plugin> EXPORT SNAPSHOT`
- Retourne un tuple contenant tout ce dont on a besoin !
  - `slot_name`
  - `consistent_point`
  - **`snapshot_name`**
  - `output_plugin`

Base backup: problème numéro 1

Utilisation du snapshot

- Une fois que l'on a le nom du snapshot, utiliser pg_dump pour s'attacher au snapshot et réaliser le dump pour nous (schéma)
- Avec cela, nous savons que nous avons restauré à un point spécifique dans le flux des WAL, à partir duquel on peut repartir et appliquer les transactions suivantes.

Base backup: problème numéro 2

Restaurer dans un schéma différent

- À notre connaissance, pas d'outil capable de sauvegarder / restaurer des objets dans un autre schéma
- Besoin de le faire nous même
- Recréer chaque objet du schema public de db1 dans un schéma db1_public sur le serveur secondaire logique.

Base backup: problème numéro 2

On a trois problèmes...

- Solution 1: Réécrire le dump avec des regexp !
- Ça marche !
  - En fait non...
- Certains objets ne devraient pas être traduit vers un autre schéma
  - les extensions
  - les références à leur contenu
  - plus de schémas que juste public

Base backup: problèm numéro 2

Faire les choses correctement

- Solution 2: parser le dump SQL, construire un AST, le modifier, et réémettre les requêtes modifiées.
- pg_last à la rescousse (https://github.com/lelit/pglast)
- Nous permets de réécrire les noms des objets dans l'AST
- Noms de fonctions etc...
- La liste des objets à ignorée récupérée dynamiquemnet depuis *pg_depend*

Base backup: problème numéro 2

Exemple pglast

- Requête original: `CREATE TYPE public.t1 AS (id int)`
- Parse tree


```
(
"stmt": {
    "CompositeTypeStmt": {
        "typevar": {
            "RangeVar": {
                "schemaname": "public",
                "relname": "t1",
                "relpersistence": "p",
            }
        },
        "coldeflist": [
            {
                "ColumnDef": {
                    "colname": "id",
                    "typeName": {
                        "TypeName": {
                            "names": [
                                {"String": {"str": "pg_catalog"}},
                                {"String": {"str": "int4"}},
                            ],
                            "typemod": -1,
                        }
                    },
                    "is_local": True,
                }
            }
        ],
    }
}
)
```

Base backup

Résumé

- Création d'un slot de réplication, récupération du snapshot associé
- Réalisation d'un pg_dump du schéma uniquement avec ce snapshot
- Parsing et réécriture du SQL pour émettre les requêtes de création d'objet dans un schéma diférent, filtrant ce dont on ne veut pas au passag (triggers, indexes...)
- Démarrage de plusieurs processus `COPY` en parallèle depuis la base source vers la base cible
- Création d'une repliction origin sur la base cible pour garder trace de la position dans les WALs

Streaming replication

Implémentation

- Petit démon en C
- Utilise le plugin de décodage wal2json
- Bête et méchant: emphase sur le débit pour consommer les WALs le plus rapidement possible.
- Environ 70k changements consommables par seconde

Streaming replication

Comment ça marche

- Ouverture d'une connection de réplication (libpq)
- Connection à la base cible pour obtenir le statut de la replication origin (dernier LSN consommé)
- Démarre le plugin de réplication à la connection `START_REPLICATION SLOT <slot_nae> LOGICAL <wal_position> (<plugin_options>)`
- Chaque transaction sur la base source résulte en une transaction sur la cible
- Association du LSN concerné à la transation (sur la cible) `SELECT pg_replication_origin_xact_setup('%X/%X', '%s')`
- Diffusion des changements depuis wal2json (protocole copy) et copie des messages dans une table de spool
- Un démon pour chaque couple (source, cible)

Streaming replication

Structure des messages

- wal2json envoie un gros json par transaction
- option "write-in-chunks" obligatoire pour les grosses transactions
- stockage tel quel dans la table de spool

```
insert_timestamp | 2019-05-13 14:43:12.698439+00
database         | database1
lsn_start        | 326D/8254F780
xid              | 1087436211
payload          | {"kind": "update", "table": "table", "schema": "public",
                 |  "oldkeys": {"keynames": ["id"], "keyvalues": [1]},
                 |  "columnnames": ["id", "label"],
                 |  "columnvalues": [1, "label 1"]}
xid_timestamp    | 2019-05-13 14:43:12.069522+00
```

people doc
BY ULTIMATE SOFTWARE

Replayer

Comment appliquer les changements ?

- Un process de rejeu par base de données cible
- Utilise la base source pour déterminer le search_path
- Reconstruit les ordres INSERT / UPDATE / DELETE depuis le payload
- Multi-threadé
- Peut rejouer ~15k changements par seconde

Replayer

Mais aussi...

- Crée les indexes manquants sur les colonnes participants aux FK
- Propage les DDL:
  - Un event trigger stocke les ordres DDL dans une table sur la base source
  - Cas spécial pour cette table: rejoue la requête dans le bon search_path plutôt que de l'insérer
  - Nécessite de la vigilance lors de l'écriture des migrations: les noms des objets ne doivent pas être qualifiés par le schéma pour laisser le search_path les résoudre.

Supervision

Comment détecter les incidents ?

- `pg_replication_slots` permet de suivre le retard de réplication sur le primaire
- La taille de la table de spool `raw_messages` reflète le retard de rejeu des WALs
- En cours: métrique métier "retard de réplication (en s)"

Et après ?

Futur du projet

- Déjà en production
- Plus d'intelligence du côté du rejeu des DDL
- Libération et publication du code
- Remplacement de wal2json par un format plus facile à parser

Les problèmes principaux

Qu'est-ce qui était plus difficile que prévu ?

- Déplacer les objets d'un schéma à un autre est plus compliqué qu'il ne semble (valeurs par défaut, surtout sur les séquences)
- Réécrire le SQL à la volée expose beaucoup de cas particuliers
- Attention aux performances

Les problèmes principaux

Quelques soucis avec PostgreSQL

- Les event triggers ne sont pas la panacée
  - Souvent, pas accès au vrai DDL exécuté
- Certains changement sont curieux
  - Les requêtes réécrivant une table génèrent des changements dans une table `pg_temp_*` (corrigé en 11)
  - `CREATE TABLE AS` génère des changements (INSERT) avant que l'event trigger ne soit déclenché

Conclusion

Questions ?

- Ce projet nous a beaucoup appris sur le fonctionnement de la réplication logique
- Nous espérons que ça pourra vous être utile !