Module W5

Réplication logique

24.04

Table des matières

Sur ce document

Formation	Module W5
Titre	Réplication logique
Révision	24.04
PDF	https://dali.bo/w5_pdf
EPUB	https://dali.bo/w5_epub
HTML	https://dali.bo/w5_html
Slides	https://dali.bo/w5_slides
TP	https://dali.bo/w5_tp
TP (solutions)	https://dali.bo/w5_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

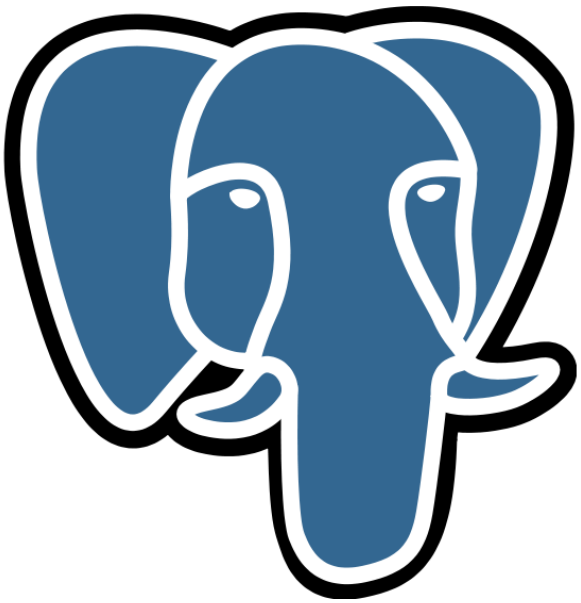
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Réplication logique



1.1 OBJECTIFS

- Réplication logique native
  - connaître les avantages et limites
  - savoir la mettre en place
  - savoir l'administrer et la superviser
- Connaître d'autres outils de réplication logique

Nous verrons ici les principes derrière la réplication logique, les différences avec la réplication physique classique, sa mise en place, son administration et sa supervision.

Historiquement sont apparus des outils de réplication logique externes à PostgreSQL, qui peuvent encore rendre des services.

1.1.1 Au menu

- Principes
- Mise en place
- Exemple
- Administration
- Supervision
- Migration majeure avec la réplication logique
- Limitations
- Autres outils de réplication logique

1.2 PRINCIPES DE LA RÉPLICATION LOGIQUE NATIVE

- Réplication logique
  - résout certaines des limitations de la réplication physique
  - native depuis la version 10
  - avant v10 : solutions externes
  - préférer tout de même PostgreSQL >= 14

La réplication physique, qui existe dans PostgreSQL depuis la version 9.0, fonctionne par application de bloc d'octets ou de delta de bloc. Elle a beaucoup évolué mais possède quelques limitations difficilement contournables directement.

La réplication logique apporte des réponses à ces limitations depuis PostgreSQL 10. Seules des solutions tierces apportaient ce type de réplication à PostgreSQL auparavant. Il est préférable d'utiliser une version récente de PostgreSQL (14 au moins) pour profiter des nombreuses améliorations et optimisations.

1.2.1 Réplication physique vs. logique

Physique	Logique
Instance complète	Tables aux choix
Par bloc	Par ligne/colonnes
Asymétrique (1 principal)	Asymétrique / croisée
Depuis primaire ou secondaire	Depuis primaire ou secondaire (v16)
Toutes opérations	Opération au choix
Réplica identique	Destination modifiable
Même architecture	-
Mêmes versions majeures	-
Synchrone/Asynchrone	Synchrone/Asynchrone

Principe & limites de la réplication physique :

La réplication physique est une réplication au niveau bloc. Le serveur primaire envoie au secondaire les octets à ajouter/remplacer dans des fichiers. Le serveur secondaire n'a aucune information sur les objets logiques (tables, index, vues matérialisées, bases de données). Il n'y a donc pas de granularité

possible, c'est forcément l'instance complète qui est répliquée. Cette réplication est par défaut en asynchrone mais il est possible de la configurer en synchrone suivant différents modes.

Malgré ses nombreux avantages, la réplication physique souffre de quelques défauts.

Il est impossible de ne répliquer que certaines bases ou que certaines tables (pour ne pas répliquer des tables de travail par exemple). Il est aussi impossible de créer des index spécifiques ou même des tables de travail, y compris temporaires, sur les serveurs secondaires, vu qu'ils sont strictement en lecture seule.

Un serveur secondaire ne peut se connecter qu'à un serveur primaire de même version majeure. On ne peut donc pas se servir de la réplication physique pour mettre à jour la version majeure du serveur.

La réplication physique peut se faire depuis un serveur secondaire (réplication en cascade).

Enfin, il n'est pas possible de faire de la réplication entre des serveurs d'architectures matérielles ou logicielles différentes (32/64 bits, *little/big endian*, version de bibliothèque C, etc.).

Réplication logique :

La réplication logique propose une solution à tous ces problèmes.

La réplication logique est une réplication du contenu des tables. Plus précisément, elle réplique les résultats des ordres SQL exécutés sur la table publiée et l'applique sur la table cible. Les lignes insérées, modifiées et/supprimées sur le serveur d'origine sont répliquées sur la destination. La table cible peut être modifiée (index notamment), et son contenu différer de la table source.

Elle se paramètre donc table par table, et même opération par opération.

Elle est asymétrique dans le sens où il existe une seule origine des écritures pour une table. Il est possible de réaliser des réplications croisées où un ensemble de tables est répliqué du serveur 1 vers le serveur 2 et un autre ensemble de tables est répliqué du serveur 2 vers le serveur 1, ce qui est une forme limitée de multimaître. En version 16, la possibilité de filtrer l'origine d'une table permet d'éviter les boucles et d'alimenter la même table depuis deux instances différentes. Cette fonctionnalité est encore jeune et peut mener à des problèmes d'intégrité des données (voir notamment cet article de Brian Pace[1]).

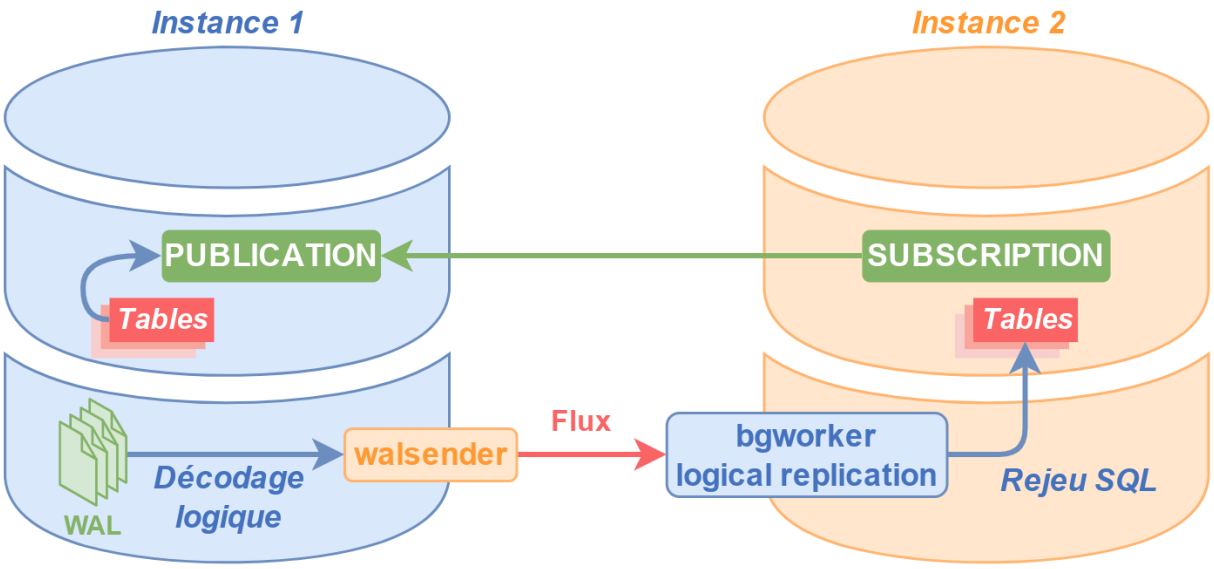
Comme la réplication physique, la réplication logique peut fonctionner en asynchrone ou en synchrone, si l'on accepte l'impact sur les performances.

La réplication logique ne peut se faire depuis un serveur secondaire (en réplication physique) que si l'origine et la destination sont au moins en PostgreSQL 16.

La réplication logique permet de répliquer entre deux serveurs PostgreSQL de versions différentes, et ainsi de procéder à des migrations majeures.

[1] https://www.crunchydata.com/blog/active-active-postgres-16#dont-get-too-carried-away

1.2.2 Schéma de principe de la réplication logique



1.2.3 Quelques termes essentiels

- Serveur origine (publieur/éditeur)
  - publication
- Serveur(s) abonné(s) (*subscriber*)
  - abonnement (*subscription*)

Dans le cadre de la réplication logique, on ne réplique pas une instance vers une autre. On publie les modifications effectuées sur le contenu d'une table à partir d'un serveur. Ce serveur est le serveur origine, ou publieur (*publisher*). Sur ce serveur, on crée des « publications ».

Une publication enregistre un jeu de modifications que d'autres serveurs pourront récupérer en s'abonnant (*subscription*).

De ceci, il découle que :

- le serveur origine est le serveur où les écritures sur une table sont enregistrées pour publication vers d'autres serveurs ;
- les serveurs intéressés par ces enregistrements sont les serveurs destinations ;
- un serveur origine doit proposer une publication des modifications ;
- les serveurs destinations intéressés doivent s'abonner à une publication.

Dans un cluster de réplication, un serveur peut avoir un rôle de serveur origine ou de serveur destination. Il peut aussi avoir les deux rôles. Dans ce cas, il sera origine pour certaines tables et destinations pour d'autres. Il ne peut pas être à la fois origine et destination pour la même table.

NB : dans le texte qui suit, peuvent être utilisés indifféremment les termes publieur/éditeur/origine d'une part, et abonné/souscripteur/destination et abonnement/souscription d'autre part, éventuellement en anglais.

1.2.4 Réplication logique et streaming

La réplication logique utilise le *streaming* :

- `wal_level` = `logical`
- Processus `wal sender`
  - mais pas de `wal receiver`
  - un `logical replication worker` à la place
- Décodage logique des journaux
- Asynchrone / synchrone
- Slots de réplication

La réplication logique utilise le même canal d'informations que la réplication physique : les enregistrements des journaux de transactions. Le transfert se fait par une connexion en *streaming* (ce n'est pas possible en *log shipping*). Pour que les journaux disposent de suffisamment d'informations, le paramètre `wal_level` doit être configuré avec la valeur `logical`.

Une fois cette configuration effectuée et PostgreSQL redémarré sur le serveur origine, le serveur destination peut se connecter au serveur origine dans le cadre de la réplication. Lorsque cette connexion est faite, un processus `wal sender` apparaît sur le serveur origine. Ce processus est en communication avec un processus `logical replication worker` sur le serveur destination.

Comme la réplication physique, la réplication logique peut être configurée en asynchrone comme en synchrone, suivant le même paramétrage (`synchronous_commit`, `synchronous_standby_names`).

Chaque abonné maintient un slot de réplication sur l'instance de l'éditeur. Par défaut, il est créé et supprimé automatiquement avec la souscription. La copie initiale des données crée également des slots de réplication temporaires.

Contrairement à ce qui se passe en réplication physique, l'intégralité des journaux n'est pas transmise. Le `walsender` procède à un « décodage logique » des journaux, que la documentation définit ainsi : Le décodage logique correspond au processus d'extraction de tous les changements persistants sur des tables d'une base de données dans un format cohérent et simple à comprendre, qui peut être interprété sans une connaissance détaillée de l'état interne de la base de données.

Dans PostgreSQL, le décodage logique est implémenté en décodant le contenu des journaux de transaction (WAL), qui décrivent les changements au niveau stockage, dans un format spécifique tel qu'un flux de lignes ou des ordres SQL.

Le décodage logique permet de n'envoyer aux abonnés que les informations qui concernent la table qu'ils ont demandée. Cela permet aussi de n'envoyer que les transactions validées lors du `COMMIT` (du moins dans les cas simples). Cette conversion est hélas un peu gourmande en processeur.

En face, sur une instance PostgreSQL avec des souscriptions, le rôle du `logical replication worker` est de retransmettre les modifications reçues dans les tables concernées.

Noter que le décodage logique permet d'écrire assez facilement des plugins de sortie alternatifs sans outil supplémentaire sur le serveur. Par exemple pg_wal2json[2] permet de s'abonner à une table et de restituer ses modifications sous forme d'objets JSON.

1.2.5 Granularité de la réplication logique

- Par table
  - toutes les tables d'une base
  - toutes les tables d'un schéma (v15+)
  - quelques tables spécifiques
- Granularité d'une table
  - table complète
  - même partitionnée (v13+)
  - uniquement certaines lignes/colonnes (v15+)
- Par opération
  - `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`

La granularité de la réplication physique est simple : c'est l'intégralité de l'instance ou rien.

À l'inverse, la réplication logique propose une granularité à la table près, voire même un niveau en dessous. Une publication se crée en indiquant la table pour laquelle on souhaite publier les modifications. On peut en indiquer plusieurs. On peut en ajouter après en modifiant la publication. Cependant, une nouvelle table ne sera pas ajoutée automatiquement à la publication, sauf dans deux cas précis : la publication a été créée en demandant la publication de toutes les tables (clause `FOR ALL TABLES`) ou, à partir de la version 15, la publication a été créée en demandant la publication de toutes les tables d'un schéma (clause `FOR TABLES IN SCHEMA`).

[2] https://github.com/eulerto/wal2json

À partir de la version 13, il est possible d'ajouter une table partitionnée à une publication. Cette fonctionnalité permet de laisser à PostgreSQL le soin d'ajouter et maintenir à jour la liste des partitions dans la publication. Il est également possible de faire en sorte que les modifications soient publiées avec le nom de la partition finale ou celui de la table partitionnée. Cela permet plus de flexibilité en autorisant de répliquer des données entre des tables avec des structures hétérogènes (partitionnées ou non). Dans le cas d'une réplication depuis une table partitionnée vers une autre table partitionnée, l'agencement des partitions peut être différent.

À partir de la version 15, la granularité est encore plus basse : il est possible de ne filtrer que certaines colonnes et que certaines lignes.

La granularité peut aussi se voir au niveau des opérations de modification réalisées. On peut très bien ne publier que les opérations d'insertion, de modification ou de suppression. Par défaut, tout est publié.

1.2.6 Possibilités sur les tables répliquées

- Possibilités
  - index supplémentaires
  - modification des valeurs
  - colonnes supplémentaires
  - triggers également activables sur la table répliquée
- **Attention à la cohérence des modèles**
- **Attention à ne pas bloquer la réplication logique !**
  - aller au plus simple

La réplication logique permet plusieurs choses impensables en réplication physique. Les cas d'utilisation sont en fait très différents.

On peut rajouter ou supprimer des index sur la table répliquée.

Il est possible de modifier des valeurs dans la table répliquée. Ces modifications sont susceptibles d'être écrasées par des modifications de la table source sur les mêmes lignes. Il est aussi possible de perdre la synchronisation entre les tables, notamment si on modifie la clé primaire.

Les triggers ne se déclenchent par défaut que sur la base d'origine. On peut activer ainsi un trigger sur la table répliquée :

```
ALTER TABLE matable ENABLE REPLICA TRIGGER nom_trigger ;
```

Tout cela est parfois très pratique mais peut poser de sérieux problème de cohérence de données entre les deux instances si l'on ne fait pas attention. On vérifiera régulièrement les erreurs dans les

traces.

Il est dangereux d'ajouter sur la destination des contraintes qui n'existent pas sur les tables d'origine ! Elles ne sont pas forcément contrôlées à l'arrivée (clés étrangères, vérification par triggers…) Et si elles le sont, elles risquent de bloquer la réplication logique. De même, sur la destination, ajouter ou modifier des lignes soumises à des contraintes d'unicité peut empêcher l'insertion de lignes provenant de la source.

En cas de blocage, à cause d'une colonne absente, d'un doublon, d'une autre contrainte sur la cible ou pour une autre raison, il faut corriger sur la destination, puis laisser le stock de données bloquées s'insérer avant de pouvoir faire autre chose. L'alternative est de désactiver ou reconstruire la réplication, ce qui peut poser des problèmes de réconciliation de données.

Il existe quelques cas surprenants. Par exemple, une colonne remplie grâce à une valeur `DEFAULT` sur l'origine sera répliquée à l'identique sur la destination ; mais une colonne calculée (clause `GENERATED` avec expression) sera calculée sur l'origine et sur la destination, éventuellement différemment.

Il est possible de créer une publication sur une table elle-même répliquée. La sécurité pour éviter des boucles n'a été ajoutée qu'avec PostgreSQL 16[3].

Pour que la réplication logique fonctionne sans souci, il faut viser au plus simple, avec un modèle de données sur la destination aussi proche que possible de la source, soigneusement maintenu à jour à l'identique. Éviter de modifier les données répliquées. Au plus, se contenter d'ajouter sur la destination des index non uniques ou des colonnes calculées.
Prévoir dès le début le cas où cette réplication devra être arrêtée et reprise de zéro.

[3] https://amitkapila16.blogspot.com/2023/09/evolution-of-logical-replication.html

1.2.7 Limitations de la réplication logique

- Pas de réplication des requêtes DDL
  - à refaire manuellement
  - être rigoureux et surveiller les traces !
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Problèmes avec les tables partitionnées (< v13)
- PK/UK conseillée pour les `UPDATE` / `DELETE`
- Coût CPU, disque, RAM
- Réplication déclenchée uniquement lors du `COMMIT` (< v14)
- Attention en cas de bascule/restauration !

La réplication logique n'a pas que des atouts, elle a aussi ses propres limitations.

La première, et plus importante, est qu'elle ne réplique que les changements de données des tables (commandes DML), et pas de la définition des objets de la base de données (commandes DDL). Une exception a été faite à partir de la version 11 pour répliquer les ordres `TRUNCATE` car, même s'il s'agit d'un ordre DDL d'après le standard, cet ordre modifie les données d'une table.

L'ajout (ou la suppression) d'une colonne ne sera pas répliqué, causant de ce fait un problème de réplication quand l'utilisateur y ajoutera des données. La mise à jour sera bloquée jusqu'à ce que les tables abonnées soient aussi mises à jour. Pour éviter le blocage, il est préférable de commencer une opération d'ajout de colonne sur l'abonné, et une opération de suppression de colonne sur le publieur.

D'autres opérations moins évidentes peuvent aussi poser problème, comme une contrainte ou un index supprimé sur l'origine mais pas la cible ; ou un index fonctionnel dont la fonction n'est corrigée que sur la source.

Il faut être rigoureux et surveiller les erreurs dans les traces.

Une table nouvellement créée ne sera pas non plus automatiquement répliquée.

Les tables partitionnées, sur la source ou l'origine, ne sont bien gérées qu'à partir de PostgreSQL 13.

Il n'y a pas de réplication des valeurs des séquences. Les valeurs des séquences sur les serveurs de destination seront donc obsolètes.

Les Large Objects (pour le stockage de gros binaires) étant stockés dans une table système, ils ne sont pas pris en compte par la réplication logique. Il est préférable de passer par le type `bytea` pour les

données binaires.

Il faut que PostgreSQL sache opérer une correspondance entre les lignes des deux instances pour gérer correctement les opérations `UPDATE` et `DELETE` Pour cela, **il est chaudement conseillé qu'il y ait une clé primaire sur chaque table répliquée.** Sinon, il faut définir une clause `REPLICA IDENTITY` sur la table origine. Utiliser un index unique peut convenir :

```
ALTER TABLE nomtable REPLICA IDENTITY USING INDEX nomtable_col_idx ;
```

Si vraiment on n'a pas le choix, on peut définir que l'ensemble des champs de la ligne servira à la correspondance :

```
ALTER TABLE nomtable REPLICA IDENTITY FULL ;
```

Faute d'index, les mises à jour effectuent des *Seq Scan* sur la table de destination, ce qui généralement catastrophique pour les performances. Toutefois, avec PostgreSQL 16, un simple index B-tree ou hash sur la table destination peut permettre d'éviter ces *Seq Scan*[4].

La réplication logique a un coût en CPU (sur les deux instances concernées) relativement important : attention aux petites configurations. Il y a également un coût en RAM et disque (voir plus bas).

La réplication n'est par défaut déclenchée que lors du `COMMIT` sur le primaire, nous verrons que cela peut être optimisé à partir de PostgreSQL 14.

La situation peut devenir compliquée lors d'une restauration ou bascule d'un des serveurs impliqués (voir plus bas).

[4] https://amitkapila16.blogspot.com/2023/09/evolution-of-logical-replication.html

1.3 MISE EN PLACE

Étapes :

- Configuration du serveur origine
- Configuration du serveur destination
- Création d'une publication
- Ajout d'une souscription

Nous allons voir les étapes de configuration dans le cas simple d'une publication origine alimentant un abonnement destination.

1.3.1 Configurer le serveur origine : utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION ;
GRANT SELECT ON ALL TABLES IN SCHEMA monschema TO logrepli ;

# pg_hba.conf
host base_publication  logrepli XXX.XXX.XXX.XXX/XX scram-sha-256
```

Dans le cadre de la réplication avec PostgreSQL, c'est toujours le serveur destination qui se connecte au serveur origine. Pour la réplication physique, on utilise plutôt les termes de serveur primaire et de serveur secondaire mais c'est toujours du secondaire vers le primaire, de l'abonné vers l'éditeur.

Tout comme pour la réplication physique, il est nécessaire de disposer d'un utilisateur PostgreSQL capable de se connecter au serveur origine et capable d'initier une connexion de réplication. Voici donc la requête pour créer ce rôle :

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

Cet utilisateur doit pouvoir lire le contenu des tables répliquées. Il lui faut donc le droit `SELECT` sur ces objets, souvent simplement ceci :

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

Enfin, la connexion du serveur destination doit être possible sur le serveur origine. Il est donc nécessaire d'avoir une ligne du style :

```
host base_publication   logrepli XXX.XXX.XXX.XXX/XX scram-sha-256
```

en remplaçant `XXX.XXX.XXX.XXX/XX` par l'adresse CIDR du serveur destination. La méthode d'authentification peut aussi être changée suivant la politique interne. Suivant la méthode

d'authentification, il sera nécessaire ou pas de configurer un mot de passe pour cet utilisateur. Ne pas oublier de recharger la configuration.

1.3.2 Configurer le serveur origine : postgresql.conf

- `wal_level = logical`
  - rédémarrage
- `logical_decoding_work_mem = 64MB` (v13+)
  - en-deça : RAM jusque `COMMIT`
  - puis disque
  - ou transmission immédiate (v14+)

Les journaux de transactions doivent disposer de suffisamment d'informations pour que le `wal sender` puisse envoyer les bonnes informations au `logical replication worker`. Le fichier `postgresql.conf` doit donc contenir :

```
wal_level = logical
```

Le défaut est `replica`, et il faudra donc sans doute redémarrer l'instance.

Le paramètre `logical_decoding_work_mem` contrôle la quantité de mémoire allouée à un processus `walsender` pour conserver les modifications en mémoire avant de les stocker sur le disque (et parfois de les envoyer tout de suite au client, voir plus bas).

En effet, la réplication logique, contrairement à la réplication physique, n'est déclenchée que lors d'un `COMMIT` (voir cet article[5]). Par défaut, il n'y a pas d'envoi des données tant que la transaction est en cours, ce qui peut ajouter beaucoup de délai de réplication pour les transactions longues.

`logical_decoding_work_mem` vaut par défaut 64 Mo. Il peut donc être réduit pour baisser l'utilisation de la mémoire des `walsender`, ou augmenté pour réduire les écritures sur le disque.

Avant PostgreSQL 13 et l'apparition de ce paramètre, les modifications d'une transaction étaient stockées en mémoire jusqu'à ce que la transaction soit validée par un `COMMIT`. En conséquence, si cette transaction possédait de nombreuses sous-transactions, chaque `walsender` pouvait allouer énormément de mémoire, menant parfois à un dépassement de mémoire.

[5] http://amitkapila16.blogspot.com/2021/07/logical-replication-of-in-progress.html

1.3.3 Configuration du serveur destination

- Création, si nécessaire, des tables répliquées

```
pg_dump -h origine -s -t la_table la_base | psql la_base
```

Sur le serveur destination, il n'y a pas de configuration à réaliser dans les fichiers `postgresql.conf` et `pg_hba.conf`.

Ensuite, il faut récupérer la définition des objets répliqués pour les créer sur le serveur de destination. Un moyen simple est d'utiliser `pg_dump` et d'envoyer le résultat directement à `psql` pour restaurer immédiatement les objets. Cela se fait ainsi :

```
pg_dump -h origine --schema-only base | psql base
```

Il est aussi possible de sauvegarder la définition d'une seule table en ajoutant l'option `-t` suivi du nom de la table pour avoir son script.

Il est conseillé de déclarer l'objet sur la destination avec la même définition que sur l'origine, mais ce n'est pas obligatoire tant que les mises à jour arrivent à se faire. Les index, notamment, peuvent différer, des types être plus laxistes, des colonnes supplémentaires ajoutées.

1.3.4 Créer une publication

```
CREATE PUBLICATION pub_t1     FOR TABLE t1 ;
CREATE PUBLICATION pub_t1part FOR TABLE t1 (c1, c3);  -- v15
CREATE PUBLICATION pub_tout   FOR ALL TABLES ;
CREATE PUBLICATION pub_public FOR TABLES IN SCHEMA public ; -- v15

CREATE PUBLICATION pub_filtree
FOR TABLE employes  WHERE ( ville = 'Brest' ) ; --v15

… WITH ( publish = 'update, delete, insert, truncate')  -- défaut

… WITH (publish_via_partition_root = false)  -- défaut, v13
```

Une fois que les tables sont définies des deux côtés (origine et destination), il faut créer une publication sur le serveur origine. Cette publication indiquera à PostgreSQL les tables répliquées et les opérations concernées.

La clause `FOR ALL TABLES` permet de répliquer toutes les tables de la base, sans avoir à les nommer spécifiquement. De plus, toute nouvelle table sera répliquée automatiquement dès sa création.

À partir de la version 15, la clause `FOR TABLES IN SCHEMA` permet de répliquer toutes les tables du schéma indiqué sans avoir à nommer les tables spécifiquement. De plus, toute nouvelle table de ce schéma sera répliquée automatiquement dès sa création. (Il faudra tout de même rafraîchir l'abonnement sur le destinataire).

Si on ne souhaite répliquer qu'un sous-ensemble, il faut spécifier toutes les tables à répliquer en utilisant la clause `FOR TABLE` et en séparant les noms des tables par des virgules.

Depuis la version 15, il est possible de ne répliquer que certaines colonnes d'une table, par exemple ainsi : exemple :

```
CREATE PUBLICATION pub1
  FOR TABLE t1 (c1, c3);
```

Toujours depuis cette version, il est possible de ne répliquer que les lignes validant une certaine expression. Par exemple :

```
CREATE PUBLICATION pub_brest
  FOR TABLE employes WHERE (ville='Brest');
```

Par défaut, une table est répliquée intégralement, donc toutes les colonnes et toutes les lignes.

La clause `FOR TABLES` n'est pas obligatoire, la publication peut être vide au départ.

Cette publication est concernée par défaut par toutes les opérations d'écriture (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`). Cependant, il est possible de préciser les opérations si on ne les souhaite pas toutes. Pour cela, il faut utiliser le paramètre de publication `publish` en utilisant les valeurs `insert`, `update`, `delete` et/ou `truncate` et en les séparant par des virgules si on en indique plusieurs.

Lorsque l'on publie les modifications sur une table partitionnée, PostgreSQL utilise par défaut le nom de la partition finale. Il est possible de lui demander d'utiliser le nom de la table partitionnée grâce à l'option `publish_via_partition_root = true`. Cela permet de répliquer d'une table partitionnée vers une table normale ou une table partitionnée avec un agencement de partitions différent.

1.3.5 Souscrire à une publication

```
CREATE SUBSCRIPTION nom
    CONNECTION 'infos_connexion'
    PUBLICATION nom_publication [, ...]
    [ WITH ( parametre_souscription [= value] [, ... ] ) ]
```

- `infos_connexion` : chaîne de connexion habituelle
- Par : superutilisateur ou `pg_create_subscription`

Une fois la publication créée, le serveur destination doit s'y abonner. Il doit pour cela indiquer sur quel serveur se connecter et à quelle publication souscrire.

Chaîne de connexion :

Le serveur s'indique avec la chaîne `infos_connexion`, dont la syntaxe est la syntaxe habituelle des chaînes de connexion avec `host`, `port`, `user`, `password`, `dbname`, etc.

Droits :

Pour créer ou modifier la souscription, il faut être superutilisateur, ou posséder le rôle `pg_create_subscription` (à partir de PostgreSQL 16). Dans ce dernier cas, il faut aussi le droit `CREATE` sur la base.

Il y a une subtilité pour le mot de passe : il doit être présent dans la chaîne de connexion (`password = motdepasse`), ce qui impose une méthode de connexion avec mot de passe (par exemple `scram-sha-256` mais pas `peer` ou `ldap`). Le superutilisateur n'a pas cette contrainte.

Une alternative est de créer la souscription en tant que superutilisateur en désactivant le mot de passe avant de changer le propriétaire :

```
ALTER SUBSCRIPTION abonnement SET (password_required = false) ;
ALTER SUBSCRIPTION abonnement OWNER TO erpadmin ;
```

Sans cela, PostgreSQL n'empêchera pas de transférer la propriété d'une souscription à un non-superutilisateur ; mais l'obligation du mot de passe risque de poser divers problèmes lors des modifications de la souscription.

Bien sûr, les accès aux tables répliquées ne nécessite aucun droit sur les souscriptions même.

Autres paramètres :

Le champ `nom_publication` doit être remplacé par le nom de la publication créée précédemment sur le serveur origine.

Les paramètres de souscription sont détaillés ci-dessous.

1.3.6 Options de la souscription (1/2)

Par défaut :

- `connect = true`
  - connexion immédiate
- `copy_data = true`
  - copie initiale des données
- `create_slot = true`
  - création du slot de réplication
- `enabled = true`
  - activation immédiate de la souscription
- `slot_name = <nom de la souscription>`
  - nom du slot de réplication

Les options de souscription sont assez nombreuses et permettent de créer une souscription pour des cas particuliers. (Pour les détails, voir la documentation officielle[6].)

Par exemple, si le serveur destination possède déjà les données du serveur origine, il faut placer le paramètre `copy_data` à la valeur `false` dans la clause `WITH` de `CREATE SUBSCRIPTION`.

`enabled = false` permet de mettre en place la souscription sans la démarrer.

[6] https://docs.postgresql.fr/current/sql-createsubscription.html

1.3.7 Options de la souscription (2/2)

Par défaut :

- `streaming = off`
  - `true` pour envoyer les modifications avant `COMMIT` (v14+)
  - évite de gros fichiers sur le primaire
  - `parallel` : plusieurs workers (v16+)
- `binary = off` (v14+)
  - pour envoyer les données sous un format binaire
- `disable_on_error = false`
  - désactivation de la souscription en cas d'erreurs détectées
- `synchronous_commit = off`
  - surcharge `synchronous_commit` pour les `wal sender`

streaming :

Ce paramètre est très important pour les performances.

Par défaut, le `walsender` de l'origine attend le `COMMIT`, et aussi d'avoir décodé toute la transaction, avant de l'envoyer aux abonnés. De grosses transactions peuvent alors entraîner de la consommation mémoire (jusque `logical_decoding_work_mem`), puis l'apparition d'énormes fichiers temporaires dans le répertoire `pg_replslot` du serveur d'origine.

Depuis la version 14, le client peut demander l'envoi des données au fil de l'eau, sans attendre le `COMMIT`, dès que `logical_decoding_work_mem` est atteint. Le serveur destination stocke alors lui-même les données dans un fichier temporaire, et ne les rejoue qu'à réception du `COMMIT`.

Cette fonctionnalité doit s'activer explicitement avec le paramètre `streaming` au niveau de la souscription depuis le client :

```
CREATE SUBSCRIPTION sub_stream
  CONNECTION 'connection string'
  PUBLICATION pub WITH (streaming = on);
```

ou :

```
ALTER SUBSCRIPTION sub_stream SET (streaming = on);
```

Depuis PostgreSQL 16, on peut même paralléliser :

```
ALTER SUBSCRIPTION sub_stream SET (streaming = parallel);
```

Dans ce cas plusieurs workers apparaissent pour appliquer les modifications en parallèle. S'il n'est pas possible de créer les workers car il y a déjà trop de *background workers*, on revient en pratique au fonctionnement de `streaming = on`.)

binary

Activer le mode binaire est potentiellement plus rapide mais dépend beaucoup des types employés qui ne peuvent pas tous être convertis.

1.4 MISE EN PLACE : EXEMPLE

- Réplication complète d'une base
- Réplication partielle d'une base
- Réplication croisée

Pour rendre la mise en place plus concrète, voici trois exemples de mise en place de la réplication logique. On commence par une réplication complète d'une base, qui permettrait notamment de faire une montée de version. On continue avec une réplication partielle, ne prenant en compte que 2 des 3 tables de la base. Et on finit par une réplication croisée sur la table partitionnée.

1.4.1 Serveurs et schéma

- 4 serveurs
  - **s1**, 192.168.10.1 : origine de toutes les réplications, et destination de la réplication croisée
  - **s2**, 192.168.10.2 : destination de la réplication complète
  - **s3**, 192.168.10.3 : destination de la réplication partielle
  - **s4**, 192.168.10.4 : origine et destination de la réplication croisée
- Schéma
  - 2 tables ordinaires
  - 1 table partitionnée, avec trois partitions

Voici le schéma de la base d'exemple, `b1` :

```
CREATE TABLE t1 (id_t1 serial, label_t1 text);
CREATE TABLE t2 (id_t2 serial, label_t2 text);

CREATE TABLE t3 (id_t3 serial, label_t3 text, clepartition_t3 integer)
  PARTITION BY LIST (clepartition_t3);

CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES IN (1);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES IN (2);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES IN (3);

INSERT INTO t1 SELECT i, 't1, ligne '||i FROM generate_series(1, 100) i;
```

```
INSERT INTO t2 SELECT i, 't2, ligne '||i FROM generate_series(1, 1000) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 1 FROM generate_series(  1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 3 FROM generate_series(301, 600) i;

ALTER TABLE t1 ADD PRIMARY KEY(id_t1);
ALTER TABLE t2 ADD PRIMARY KEY(id_t2);
ALTER TABLE t3 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

1.4.2 Réplication complète

- Configuration du serveur origine
- Configuration du serveur destination
- Création de la publication
- Ajout de la souscription

Pour ce premier exemple, nous allons détailler les quatre étapes nécessaires.

1.4.3 Configuration du serveur origine (1/2)

- Création et configuration de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

- Fichier `postgresql.conf`

```
wal_level = logical
```

La configuration du serveur d'origine commence par la création du rôle de réplication. On lui donne ensuite les droits sur toutes les tables. Ici, la commande ne s'occupe que des tables du schéma `public`, étant donné que nous n'avons que ce schéma. Dans le cas où la base dispose d'autres schémas, il serait nécessaire d'ajouter les ordres SQL pour ces schémas.

Les fichiers `postgresql.conf` et `pg_hba.conf` sont modifiés pour y ajouter la configuration nécessaire.

1.4.4 Configuration du serveur origine (2/2)

- Fichier `pg_hba.conf`

```
host b1 logrepli 192.168.10.0/24 trust
```

- Redémarrer le serveur origine
- Attention, dans la vraie vie, ne pas utiliser `trust`
  - et utiliser le fichier `.pgpass`

Comme dit précédemment, les fichiers `postgresql.conf` et `pg_hba.conf` sont modifiés pour y ajouter la configuration nécessaire. Le serveur PostgreSQL du serveur d'origine est alors redémarré pour qu'il prenne en compte cette nouvelle configuration.

Il est important de répéter que la méthode d'authentification `trust` ne devrait jamais être utilisée en production. Elle n'est utilisée ici que pour se faciliter la vie.

1.4.5 Configuration des 4 serveurs destinations

- Création de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

- Création des tables répliquées (sans contenu)

```
createdb -h s2 b1
pg_dump -h s1 -s b1 | psql -h s2 b1
```

Pour cet exemple, nous ne devrions configurer que le serveur **s2** mais tant qu'à y être, autant le faire pour les quatre serveurs destinations.

La configuration consiste en la création de l'utilisateur de réplication. Puis, nous utilisons `pg_dump` pour récupérer la définition de tous les objets grâce à l'option `-s` (ou `--schema-only`). Ces ordres SQL sont passés à `psql` pour qu'il les intègre dans la base **b1** du serveur **s2**.

1.4.6 Créer une publication complète

- Création d'une publication de toutes les tables de la base **b1** sur le serveur origine **s1**

```
CREATE PUBLICATION publi_complete
  FOR ALL TABLES;
```

On utilise la clause `ALL TABLES` pour une réplication complète d'une base.

1.4.7 Souscrire à la publication

- Souscrire sur **s2** à la publication de **s1**

```
CREATE SUBSCRIPTION subscr_complete
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_complete;
```

- Un slot de réplication est créé sur l'origine
- Les données initiales sont immédiatement transférées

Maintenant que le serveur **s1** est capable de publier les informations de réplication, le serveur intéressé doit s'y abonner. Lors de la création de la souscription, il doit préciser comment se connecter au serveur origine et le nom de la publication.

La création de la souscription ajoute immédiatement un slot de réplication sur le serveur origine.

Par défaut, les données initiales de la table `t1` sont immédiatement envoyées du serveur **s1** vers le serveur **s2**.

1.4.8 Tests de la réplication complète

- Insertion, modification, suppression sur les différentes tables de **s1**
- Vérifications sur **s2**
  - toutes doivent avoir les mêmes données entre **s1** et **s2**

Toute opération d'écriture sur la table `t1` du serveur **s1** doit être répliquée sur le serveur **s2**.

Sur le serveur **s1** :

```
INSERT INTO t1 VALUES (101, 't1, ligne 101');
UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;
DELETE FROM t1 WHERE id_t1=11;
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
```

```
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

Sur le serveur **s2** :

```
SELECT count(*) FROM t1;
```

```
 count
-------
   100
```

```
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
```

```
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

1.4.9 Réplication partielle

- Identique à la réplication complète, à une exception…
- Créer la publication partielle

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2 ;
```

- Souscrire sur **s3** à cette nouvelle publication de **s1**

```
CREATE SUBSCRIPTION subscr_partielle
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_partielle;
```

La mise en place d'une réplication partielle est identique à la mise en place d'une réplication complète à une exception près : la publication doit mentionner la liste des tables à répliquer. Chaque nom de table est séparé par une virgule.

Mise en place :

Cela donne donc dans notre exemple :

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2;
```

Il ne reste plus qu'à souscrire à cette publication à partir du serveur **s3** avec la requête indiquée.

Vérification :

Sur **s3**, nous n'avons que les données des deux tables répliquées :

```
SELECT count(*) FROM t1;

 count
-------
   100

SELECT count(*) FROM t2;

 count
-------
  1000

SELECT count(*) FROM t3;

 count
-------
     0
```

À noter que nous avons déjà les données précédemment modifiées :

```
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);

 id_t1 |   label_t1
-------+--------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
```

Maintenant, ajoutons une ligne dans chaque table de **s1** :

```
INSERT INTO t1 VALUES (102, 't1, ligne 102');
INSERT INTO t2 VALUES (1001, 't2, ligne 1002');
INSERT INTO t3 VALUES (-1, 't3, cle 1, ligne -1', 1);
```

Et vérifions qu'elles apparaissent bien sur **s3** pour `t1` et `t2`, mais pas pour `t3` :

```
SELECT * FROM t1 WHERE id_t1=102;

 id_t1 |   label_t1
-------+--------------
   102 | t1, ligne 102

SELECT * FROM t2 WHERE id_t2=1001;

 id_t2 |    label_t2
-------+---------------
  1001 | t2, ligne 1002

SELECT * FROM t3 WHERE id_t3 < 0;

 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)
```

1.4.10 Réplication croisée

- Écrire sur une table sur **s1**
  - et répliquer sur **s4**
- Écrire sur une (autre) table sur **s4**
  - et répliquer sur **s1**
- Pour compliquer :
  - on utilisera la table partitionnée

La réplication logique ne permet pas pour l'instant de faire du multi-maîtres pour une même table. Cependant, il est tout à fait possible de croiser les réplications, c'est-à-dire de répliquer un ensemble

de tables de serveur **s1** (origine) vers **s4** (destination), de répliquer un autre ensemble en sens inverse, du serveur **s4** vers **s1**.

Pour rendre cela encore plus intéressant, nous allons utiliser la table `t3` et ses partitions. Le but est de pouvoir écrire dans la partition `t3_1` sur **s1** et dans la partition `t3_2` sur **s4**, simulant ainsi une table où il sera possible d'écrire sur les deux serveurs à condition de respecter la clé de partitionnement.

Pour le mettre en place, nous allons travailler en deux temps :

- nous allons commencer par mettre en réplication `t3_1` ;
- et nous finirons en mettant en réplication `t3_2`.

1.4.11 Réplication de t3_1 de s1 vers s4

- Créer la publication partielle sur **s1**

```
CREATE PUBLICATION publi_t3_1
  FOR TABLE t3_1;
```

- Y souscrire sur **s4**

```
CREATE SUBSCRIPTION subscr_t3_1
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_t3_1;
```

- Configurer **s4** comme serveur origine
  - `wal_level`, `pg_hba.conf`

Rien de bien nouveau ici, il s'agit d'une réplication partielle. On commence par créer la publication sur le serveur **s1** et on souscrit à cette publication sur le serveur **s4**.

Cependant, le serveur **s4** n'est plus seulement un serveur destination, il devient aussi un serveur origine. Il est donc nécessaire de le configurer pour ce nouveau rôle. Cela passe par une configuration similaire et symétrique à celle vue pour **s1** :

- Fichier `postgresql.conf` :

```
wal_level = logical
```

(Si ce n'était pas déjà fait, il faudra redémarrer l'instance PostgreSQL sur **s4**).

- Fichier `pg_hba.conf` :

```
host all logrepli 192.168.10.0/24 trust
```

(Ne pas oublier de recharger la configuration.)

1.4.12 Réplication de t3_2 de s4 vers s1

- Créer la publication partielle sur **s4**

```
CREATE PUBLICATION publi_t3_2
  FOR TABLE t3_2;
```

- Y souscrire sur **s1**

```
CREATE SUBSCRIPTION subscr_t3_2
  CONNECTION 'host=192.168.10.4 user=logrepli dbname=b1'
  PUBLICATION publi_t3_2;
```

Là-aussi, rien de bien nouveau. On crée la publication sur le serveur **s4** et on souscrit à cette publication sur le serveur **s1**.

1.4.13 Tests de la réplication croisée

- Insertion, modification, suppression sur `t3` (partition 1) sur **s1**
  - Vérifications sur **s4** : les nouvelles données doivent être présentes
- Insertion, modification, suppression sur `t3` (partition 2) sur **s4**
  - Vérifications sur **s1** : les nouvelles données doivent être présentes

Sur **s1** :

```
SELECT * FROM t3 WHERE id_t3 > 999;

 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)

INSERT INTO t3 VALUES (1001, 't3, ligne 1001', 1);
SELECT * FROM t3 WHERE id_t3>999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

Sur **s4** :

```
SELECT * FROM t3 WHERE id_t3 > 999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

```
INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 2);
SELECT * FROM t3 WHERE id_t3 > 999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               2
```

Sur **s1** :

```
SELECT * FROM t3 WHERE id_t3>999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               2
(2 rows)
```

1.5 ADMINISTRATION

- Processus
- Fichiers
- Procédures
  - Empêcher les écritures sur un serveur destination
  - Que faire pour les DDL ?
  - Gérer les opérations de maintenance
  - Gérer les sauvegardes

Dans cette partie, nous allons tout d'abord voir les changements de la réplication logique au niveau du système d'exploitation, et tout particulièrement au niveau des processus et des fichiers.

Ensuite, nous regarderons quelques procédures importantes d'administration et de maintenance.

1.5.1 Processus

- Serveur origine
  - `wal sender`
- Serveur destination
  - `logical replication launcher`
  - `logical replication worker`

Tout comme il existe un processus `wal sender` communiquant avec un processus `wal receiver` dans le cadre de la réplication physique, il y a aussi deux processus discutant ensemble dans le cadre de la réplication logique.

Le `logical replication launcher` est toujours exécuté. Ce processus a pour but de demander le lancement d'un `logical replication apply worker` lors de la création d'une souscription. Ce worker se connecte au serveur origine et applique toutes les modifications dont **s1** lui fait part. Si la connexion se passe bien, un processus `wal sender` est ajouté sur le serveur origine pour communiquer avec le *worker* sur le serveur destination.

Sur notre serveur **s2**, destinataire pour la publication complète du serveur **s1**, nous avons les processus suivant :

```
postmaster -D /opt/postgresql/datas/s2
postgres: checkpointer process
postgres: writer process
postgres: wal writer process
postgres: autovacuum launcher process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication apply worker for subscription 16445
```

Le serveur **s1** est origine de trois publications (d'où les 3 `wal sender`) et destinataire d'une souscription (d'où le seul `logical replication apply worker`). Il a donc les processus suivants :

```
postmaster -D /opt/postgresql/datas/s1
postgres: checkpointer process
postgres: writer process
postgres: wal writer process
postgres: autovacuum launcher process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication apply worker for subscription 16573
postgres: wal sender process logrepli [local] idle
postgres: wal sender process logrepli [local] idle
postgres: wal sender process logrepli [local] idle
```

1.5.2 Synthèse des paramètres sur le serveur origine

Paramètre	Valeur
`wal_level`	`logical`
`logical_decoding_work_mem`	`64MB` ou plus
`max_slot_wal_keep_size`	0 (à ajuster)
`wal_sender_timeout`	1 min
`max_wal_senders`	10 (parfois à ajuster)
`max_replication_slots`	10 (parfois à ajuster

À part les deux premiers, ces paramètres ont la même utilité que pour une réplication physique. Les valeurs par défaut sont généralement suffisantes, mais doivent parfois être augmentées.

Exception : `max_slot_wal_keep_size` doit être mis en place à une valeur élevée pour qu'un slot de réplication très en retard ou oublié ne sature le répertoire `pg_wal` du serveur origine.

1.5.3 Synthèse des paramètres sur le serveur destination

Paramètre	Valeur
`max_worker_processes`	8 (parfois à ajuster)
`max_logical_replication_workers`	4 (parfois à ajuster)

`max_logical_replication_workers` spécifie le nombre maximal de *workers* de réplication logique (*leader*, *parallel*, de synchronisation). Ils sont pris dans la réserve définie par `max_worker_processes`.

Si les valeurs sont trop basses, les réplications seront bloquées. Il faudra augmenter ces valeurs et redémarrer le serveur.

1.5.4 Fichiers (serveur origine)

- 2 répertoires importants
- `pg_replslot`
  - slots de réplication
  - 1 répertoire par slot (+ slots temporaires)
  - 1 fichier `state` dans le répertoire
  - fichiers `.snap` (volumétrie !)
- `pg_logical`
  - métadonnées
  - snapshots

La réplication logique maintient des données dans deux répertoires : `pg_replslot` et `pg_logical`.

`pg_replslot` contient un répertoire par slot de réplication physique ou logique. On y trouvera aussi des slots temporaires lors de l'initialisation de la réplication logique.

`pg_replslot` contient aussi les snapshots des transactions en cours (fichiers `.snap`). Il peut donc atteindre une taille importante si le serveur exécute beaucoup de transactions longues avec du volume en écriture, ou si l'abonné met du temps à répliquer les données. Il est donc important de surveiller la place prise par ce répertoire.

`pg_logical` contient des métadonnées et une volumétrie beaucoup plus faible.

À cela s'ajoutent les journaux de transaction conservés dans `pg_wal/` en fonction de l'avancement des slots de réplication.

1.5.5 Empêcher les écritures sur un serveur destination

- Par défaut, toutes les écritures sont autorisées sur le serveur destination
  - y compris écrire dans une table répliquée avec un autre serveur comme origine
- Problèmes
  - serveurs non synchronisés
  - blocage de la réplication en cas de conflit sur la clé primaire
- Solution
  - révoquer le droit d'écriture sur le serveur destination
  - mais ne pas révoquer ce droit pour le rôle de réplication !

Sur **s2**, nous allons créer un utilisateur applicatif en lui donnant tous les droits sur les tables répliquées, entre autres :

```
CREATE ROLE u1 LOGIN;
GRANT ALL ON ALL TABLES IN SCHEMA public TO u1;
```

Maintenant, nous nous connectons avec cet utilisateur et vérifions s'il peut écrire dans la table répliquée :

```
\c b1 u1
INSERT INTO t1 VALUES (103, 't1 sur s2, ligne 103');
```

C'est bien le cas, contrairement à ce que l'on aurait pu croire instinctivement. Le seul moyen d'empêcher ce comportement par défaut est de lui supprimer les droits d'écriture :

```
\c b1 postgres
REVOKE INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM u1;
\c b1 u1
INSERT INTO t1 VALUES (104);

ERROR:  permission denied for relation t1
```

L'utilisateur **u1** ne peut plus écrire dans les tables répliquées.

Sans cette interdiction, on peut arriver à des problèmes très gênants. Par exemple, nous avons inséré dans la table `t1` de **s2** la valeur 103 :

```
SELECT * FROM t1 WHERE id_t1=103;

 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
```

Cette ligne n'apparaît pas sur **s1** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 | label_t1
-------+----------
(0 rows)
```

De ce fait, on peut l'insérer sur la table `t1` de **s1** :

```
INSERT INTO t1 VALUES (103, 't1 sur s1, ligne 103');
```

Et maintenant, on se trouve avec deux serveurs désynchronisés :

- sur **s1** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

- sur **s2** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
(1 row)
```

Notez que le contenu de la colonne `label_t1` n'est pas identique sur les deux serveurs.

Ce n'est pas le seul problème : cette valeur insérée sur **s1** va devoir être répliquée. Le processus de réplication logique n'arrive alors plus à appliquer les données sur **s2**, avec ces messages dans les traces :

```
LOG:  logical replication apply worker for subscription "subscr_complete" has started
ERROR:  duplicate key value violates unique constraint "t1_pkey"
DETAIL:  Key (id_t1)=(103) already exists.
LOG:  worker process: logical replication worker for subscription 16445 (PID 31113)
 ↪  exited with exit code 1
```

Il faut corriger manuellement la situation, par exemple en supprimant la ligne de `t1` sur le serveur **s2** :

```
DELETE FROM t1 WHERE id_t1=103;
```

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 | label_t1
-------+----------
(0 rows)
```

Au bout d'un certain temps, le *worker* est relancé, et la nouvelle ligne est finalement disponible :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

Dans des cas plus complexes et avec plus de données, la réconciliation des données peut devenir très complexe et chronophage.

1.5.6 Que faire pour les DDL ?

- Les opérations DDL ne sont pas répliquées
- De nouveaux objets ?
  - les déclarer sur tous les serveurs du cluster de réplication
  - tout du moins, ceux intéressés par ces objets
- Changement de définition des objets ?
  - à réaliser sur chaque serveur

Seules les opérations DML sont répliquées pour les tables ciblées par une publication. Toutes les opérations DDL sont ignorées, que ce soit l'ajout, la modification ou la suppression d'un objet, y compris si cet objet fait partie d'une publication.

Il est donc important que toute modification de schéma soit effectuée sur toutes les instances d'un cluster de réplication. Ce n'est cependant pas requis. Il est tout à fait possible d'ajouter un index sur un serveur sans vouloir l'ajouter sur d'autres. C'est d'ailleurs une des raisons de passer à la réplication logique.

Par contre, dans le cas du changement de définition d'une table répliquée (ajout ou suppression d'une colonne, par exemple), il est nettement préférable de réaliser cette opération sur tous les serveurs intégrés dans cette réplication.

1.5.7 Que faire pour les nouvelles tables ?

- Créer la table sur origine **et** destination
- Publication `FOR ALL TABLES` / `FOR TABLES IN SCHEMA`
  - prise en compte automatique ajouter la table aux souscriptions concernées :

```
-- origine
ALTER PUBLICATION … ADD TABLE …, TABLE … ;
ALTER SUBSCRIPTION … REFRESH PUBLICATION ;
```

La création d'une table est une opération DDL. Elle est donc ignorée dans le contexte de la réplication logique. Si l'on veut la répliquer, il faut d'abord créer la table manuellement dans la base destinataire. Puis, plusieurs cas se présentent :

- si la publication a été définie `FOR ALL TABLES`, la nouvelle table sera prise en compte immédiatement ;
- si la publication a été définie `FOR ALL TABLES IN SCHEMA`, et que la nouvelle table est dans le bon schéma, elle sera aussi prise en compte ;
- si la publication liste les tables une à une, il va falloir l'ajouter manuellement à la publication ainsi :

```
ALTER PUBLICATION … ADD TABLE … ;
```

Dans les deux cas, sur les serveurs destinataires, il va falloir rafraîchir les souscriptions :

```
ALTER SUBSCRIPTION … REFRESH PUBLICATION ;
```

Si l'on a oublié de créer la table sur le destinataire, cela provoquera une erreur :

```
ERROR:  relation "public.t4" does not exist
```

Si la publication contient des tables partitionnées, la même commande doit être exécutée lorsque l'on ajoute ou retire des partitions à une de ces tables partitionnées.

Il est possible d'ajouter une table à une publication définie sur un schéma différent avec `FOR ALL TABLES IN SCHEMA`.

Exemple :

Sur le serveur origine **s1**, on crée la table `t4`, on lui donne les bons droits, et on insère des données :

```
CREATE TABLE t4 (id_t4 integer, PRIMARY KEY (id_t4));
GRANT SELECT ON TABLE t4 TO logrepli;
INSERT INTO t4 VALUES (1);
-- optionnel pour les publications table à table
ALTER PUBLICATION publi_partielle ADD TABLE t4 ;
```

Sur le serveur **s2**, on regarde le contenu de la table `t4` :

```
SELECT * FROM t4;

ERROR:  relation "t4" does not exist
LINE 1: SELECT * FROM t4;
                      ^
```

La table n'existe pas. En effet, la réplication logique ne s'occupe que des modifications de contenu des tables, pas des changements de définition. Il est donc nécessaire de créer la table sur le serveur destination, ici **s2** :

```
CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
SELECT * FROM t4;

 id_t4
-------
(0 rows)
```

Elle ne contient toujours rien. Ceci est dû au fait que la souscription n'a pas connaissance de la réplication de cette nouvelle table. Il faut donc rafraîchir les informations de souscription :

```
ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;

SELECT * FROM t4;

 id_t4
-------
     1
```

1.5.8 Comment ajouter une nouvelle colonne ?

- 1. Ajouter la colonne sur l'abonné
- 2. Puis ajouter la colonne sur le publieur
- Si le contraire : pas grave, la réplication reprendra une fois les colonnes ajoutées

Un publieur qui a une table avec plus de colonnes qu'un abonné posera problème à la première insertion de ligne ou modification de la colonne, et ce message apparaîtra dans les traces :

```
ERROR:  logical replication target relation "public.t4" is missing replicated
↪  column: "c9"
```

Le contraire n'est pas vrai : un abonné peut avoir une table ayant plus de colonnes que la même table sur le publieur. C'est un des intérêt de la réplication logique. Les colonnes n'ont pas non plus besoin d'être dans le même ordre sur les deux instances.

Il est donc conseillé d'ajouter la nouvelle colonne sur l'abonné en premier lieu, puis de faire la même opération sur le publieur.

Si jamais vous faites l'opération dans le sens inverse et qu'une ligne est insérée avant avoir terminé l'opération, la réplication sera en erreur jusqu'à ce que l'opération soit terminée.

1.5.9 Comment supprimer une colonne ?

- 1. Supprimer la colonne sur le publieur
- 2. Supprimer la colonne sur l'abonné
- Si le contraire : pas grave, la réplication reprendra une fois les colonnes supprimées

Comme indiqué ci-dessus, une table peut avoir plus de colonnes sur l'abonné mais pas sur le publieur. De ce fait, pour supprimer une colonne, il convient de commencer par la supprimer sur le publieur, puis de la supprimer sur l'abonné.

Si jamais vous faites l'opération dans le sens inverse et qu'une ligne est insérée avant la fin de l'opération, la réplication sera en erreur jusqu'à ce que l'opération soit terminée.

1.5.10 Comment ajouter une nouvelle contrainte ?

- 1. Ajouter la contrainte sur le publieur
- 2. Ajouter la contrainte sur l'abonné
- Si incohérence : bloquage de la réplication

L'ajout d'une contrainte identique sur les deux machines doit se faire d'abord sur le primaire. Sans cela, il y a une fenêtre pour que de nouvelles données violant cette contrainte soient insérées dans l'origine et bloquent la réplication.

Ajoutons que les contraintes ne sont pas obligatoirement les mêmes sur l'origine et la destination. Pour faciliter l'administration, c'est tout de même conseillé. Rappelons qu'une clé primaire ou unique est nécessaire pour repérer plus efficacement les lignes.

En cas de différence, il vaut donc mieux que les contraintes les plus strictes soient posées sur le publieur. Une ligne insérée sans problème sur l'origine et violant une contrainte sur la destination bloquera la réplication.

1.5.11 Comment corriger une erreur de réplication ?

- Si les données diffèrent entre les serveurs, il faut corriger manuellement les données
- Si blocage
  - publication arrêtée
  - pas de recyclage des journaux → accumulation → danger !
- Puis, avancer le pointeur du slot de réplication
  - fonction `pg_replication_slot_advance()`
  - outil `pg_waldump` ou extension `pg_walinspect`

Voici un exemple complet de correction d'une erreur de réplication.

Commençons par mettre en place une réplication logique entre deux serveurs **s1** (port 5432) et **s2** (port 5433). Cette réplication prend en compte la seule table de la base tests1.

Dans la base **tests1** sur le publieur (**s1**) :

```
CREATE TABLE t1(c1 integer, c2 integer);
ALTER TABLE t1 ADD PRIMARY KEY(c1);
CREATE PUBLICATION pub1 FOR ALL TABLES;
```

Dans la base **tests1** sur l'abonné (**s2**) :

```
CREATE TABLE t1(c1 integer, c2 integer);
ALTER TABLE t1 ADD PRIMARY KEY(c1);
CREATE SUBSCRIPTION sub1 CONNECTION 'port=5432 dbname=tests1' PUBLICATION pub1;
```

À partir de maintenant, toute écriture sur **s1** sera lisible aussi sur **s2** :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (1,1), (2,2);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
```

```
(2 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
(2 rows)
```

Ajoutons maintenant une contrainte sur l'abonné :

```
tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres".
tests1=# ALTER TABLE t1 ADD CHECK (c2<10);
ALTER TABLE
```

Tout ajout se passera bien, sur **s1** et **s2**, tant que la contrainte est respectée :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (3,3), (4,4);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Par contre, si la contrainte n'est pas respectée, l'ajout se fera uniquement sur **s1** (qui n'a pas la contrainte) :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (11,11);
INSERT 0 1
tests1=# TABLE t1;
```

```
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
 11 | 11
(5 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Les traces du serveur **s2** nous expliquent pourquoi :

```
LOG:  logical replication apply worker for subscription "sub1" has started
ERROR:  new row for relation "t1" violates check constraint "t1_c2_check"
DETAIL:  Failing row contains (11, 11).
CONTEXT:  processing remote data for replication origin "pg_16390" during message
 ↪  type "INSERT" for replication target relation "public.t1" in transaction 748,
 ↪  finished at 0/1C442C8
LOG:  background worker "logical replication worker" (PID 194674) exited with exit
 ↪  code 1
```

Ce message sera répété tant que l'erreur ne sera pas corrigée.

De plus, aucune autre donnée de réplication ne passera par ce slot de réplication tant que l'erreur n'est pas corrigée.

Par exemple, ces cinq lignes sont bien insérées dans la table `t1` du serveur **s1** mais pas dans celle du serveur **s2** :

```
postgres=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres".
tests1=# INSERT INTO t1 (c1) SELECT generate_series(5, 9);
INSERT 0 5
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
 11 | 11
```

```
  5 |
  6 |
  7 |
  8 |
  9 |
(10 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Ceci est très problématique car les journaux de transactions ne pourront pas être recyclés sur le serveur **s1** tant que le problème n'est pas réglé. Pour éviter une perte du service sur **s1**, il est donc essentiel de corriger le problème le plus rapidement possible.

Supprimer la contrainte résoudra facilement et rapidement le problème… si la contrainte n'avait pas lieu d'être. Si, au contraire, cette contrainte est nécessaire, et si nous avons seulement oublié de l'ajouter sur le serveur **s1**, il faut pouvoir supprimer la ligne 11 sur **s1**, ajouter la contrainte sur **s1** et reprendre la réplication sur **s2**.

Voyons comment faire cela. La suppression de la ligne 11 est simple. Profitons-en en plus pour récupérer l'identifiant de transaction qui a créé la ligne :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# DELETE FROM t1 WHERE c1=11 RETURNING xmin;
 xmin
------
  748
(1 row)

DELETE 1
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
```

```
  8 |
  9 |
(9 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

La ligne 11 est bien supprimée du serveur **s1** mais ça n'a pas débloqué pour autant la situation sur le serveur **s2**. C'est normal. L'information d'ajout de la ligne est dans les journaux disponibles sur le serveur **s2**. Supprimer la ligne sur **s1** ne supprime pas l'enregistrement de l'insertion préalable de cette ligne des journaux de transactions.

En attendant, ajoutons la contrainte sur **s1** pour ne plus avoir de « mauvaises » données insérées puis répliquées :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres".
tests1=# ALTER TABLE t1 ADD CHECK (c2<10);
ALTER TABLE
```

Rétablissons maintenant la réplication sur le serveur **s2**. Nous ne voulons pas appliquer l'enregistrement qui insère la ligne 11 sur le serveur **s2**. (Sinon il serait possible de lever temporairement la contrainte, et laisser la ligne 11 s'insérer ; puis le `DELETE` ci-dessus s'appliquerait, et on pourrait remettre la contrainte en place sur **s2**.)

Pour cela, il faut pouvoir avancer le pointeur du prochain enregistrement à rejouer pour notre slot de réplication. Il existe une fonction dédiée : `pg_replication_slot_advance()`. Cette fonction prend en premier argument le nom du slot de réplication à modifier, et en deuxième argument la nouvelle position dans les journaux de transactions. Il va donc falloir trouver l'emplacement suivant dans la transaction qui a insérée cette ligne 11.

Nous savons à quel emplacement le slot est bloqué grâce à la vue `pg_replication_slots` et son champ `confirmed_flush_lsn`, qui indique le dernier enregistrement reçu, mais pas forcément appliqué :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT confirmed_flush_lsn FROM pg_replication_slots WHERE slot_name='sub1';
 confirmed_flush_lsn
---------------------
 0/1C44248
(1 row)
```

Maintenant, il faut décoder les enregistrements après cet emplacement là. Auparavant, il fallait utiliser l'outil `pg_walinspect`. Voici ce que nous donne la fonction `pg_get_wal_records_info()` de cette

extension pour les enregistrements allant de `0/1C40ED0` à la dernière position :

```
tests1=# \c tests1 - - 5432
tests1=# CREATE EXTENSION pg_walinspect;
CREATE EXTENSION
tests1=# SELECT start_lsn, xid, resource_manager, record_type, block_ref
  FROM pg_get_wal_records_info('0/1C44248', pg_current_wal_lsn()) \gx

-[ RECORD 1 ]----+-----------------------------------------------
start_lsn        | 0/1C44248
xid              | 748
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 2 ]----+-----------------------------------------------
start_lsn        | 0/1C44288
xid              | 748
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 3 ]----+-----------------------------------------------
start_lsn        | 0/1C442C8
xid              | 748
resource_manager | Transaction
record_type      | COMMIT
block_ref        |
-[ RECORD 4 ]----+-----------------------------------------------
start_lsn        | 0/1C442F8
xid              | 0
resource_manager | Standby
record_type      | RUNNING_XACTS
block_ref        |
-[ RECORD 5 ]----+-----------------------------------------------
start_lsn        | 0/1C44330
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 6 ]----+-----------------------------------------------
start_lsn        | 0/1C44370
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 7 ]----+-----------------------------------------------
start_lsn        | 0/1C443B0
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 8 ]----+-----------------------------------------------
start_lsn        | 0/1C443F0
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
```

```
-[ RECORD 9 ]----+-------------------------------------------------
start_lsn        | 0/1C44430
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 10 ]---+-------------------------------------------------
start_lsn        | 0/1C44470
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 11 ]---+-------------------------------------------------
start_lsn        | 0/1C444B0
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 12 ]---+-------------------------------------------------
start_lsn        | 0/1C444F0
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 13 ]---+-------------------------------------------------
start_lsn        | 0/1C44530
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
[...]
```

Les trois premiers enregistrements concernent la transaction 748 (colonne `xid`). C'est bien cette transaction qui a ajouté la ligne 11, comme nous l'indiquait le résultat de la requête `DELETE` ainsi que le message dans les traces de PostgreSQL indiqué plus haut.

Le premier enregistrement indique une insertion (`record_type` à `INSERT` sur la table référencée 1663/16384/16389 (colonne `block_ref`). Le premier numéro est l'OID du tablespace, le deuxième numéro est l'OID de la base de données et le dernier numéro est le `refilenode` de la table. Il se trouve que la table `t1` a comme `relfilenode` 16389 :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT relfilenode FROM pg_class WHERE relname='t1';
 relfilenode
-------------
       16389
(1 row)
```

Le deuxième enregistrement indique une écriture dans un index B-tree. Il s'agit de l'index lié à la clé primaire sur la table `t1`.

Enfin, le troisième enregistrement concerne la validation de la transaction. La transaction 748 s'arrête à l'emplacement `0/1C442F8`. Nous devons donc avancer le slot de réplication `sub1` à cet emplacement :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT pg_replication_slot_advance('sub1', '0/1C442F8');
 pg_replication_slot_advance
-----------------------------
 (sub1,0/1C442F8)
(1 row)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
(9 rows)
```

Nous pouvons voir que la réplication a repris immédiatement et que les deux tables contiennent les mêmes données. Et on peut de nouveau ajouter des données :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (-1), (-2);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
 -1 |
 -2 |
(11 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
```

```
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
 -1 |
 -2 |
(11 rows)
```

1.5.12 Gérer les opérations de maintenance

- À faire séparément sur tous les serveurs
- `VACUUM`, `ANALYZE`, `REINDEX`

Dans la réplication physique, les opérations de maintenance ne sont réalisables que sur le serveur primaire, qui va envoyer le résultat de ces opérations aux serveurs secondaires.

Ce n'est pas le cas dans la réplication logique. Il faut bien voir les serveurs d'une réplication logique comme étant des serveurs indépendants les uns des autres.

Donc il faut configurer leur maintenance, avec les opérations `VACUUM`, `ANALYZE`, `REINDEX`, comme pour n'importe quel serveur PostgreSQL.

1.5.13 Gérer les sauvegardes & restaurations logiques

- `pg_dumpall` et `pg_dump`
  - sauvegardent publications et souscriptions
  - options `--no-publications` et `--no-subscriptions`
- Restauration d'une publication :
  - nouveau slot de réplication !
  - réconciliation de données à prévoir
- Restauration d'un abonnement :
  - `ENABLE` et `REFRESH PUBLICATION`
  - reprendre à zéro la copie… ou copier manuellement ?

Les sauvegardes logiques incluent les publications et souscriptions. Deux options (`--no-publications` et `--no-subscriptions`) permettent de les exclure.

Par contre, les slots de réplication liées aux publications, et leur position dans le flux de transaction, ne sont pas sauvegardés. Cela peut poser problème pour une restauration sans perte.

Après une restauration, il faudra soigneusement vérifier dans les traces que les réplications logiques ont repris leur fonctionnement, et qu'il n'y a pas de perte dans les données transmises.

Restauration d'une publication

Voici l'ordre SQL exécuté pour la restauration d'une publication complète :

```
CREATE PUBLICATION publi_complete FOR ALL TABLES
  WITH (publish = 'insert, update, delete');
```

Et ceux correspondant à la restauration d'une publication partielle :

```
CREATE PUBLICATION publi_partielle
  WITH (publish = 'insert, update, delete');
ALTER PUBLICATION publi_partielle ADD TABLE ONLY t1;
```

La publication sera fonctionnelle, mais il peut être délicat d'y raccrocher les abonnements existants. Selon ce qui s'est passé, le slot de réplication a souvent disparu, et il peut être plus simple de recréer ces abonnements.

Si le slot manque, le recréer sur l'instance d'origine est possible :

```
SELECT pg_create_logical_replication_slot ('abonnement','pgoutput') ;
```

Ces opérations sont obligatoirement manuelles. De toute façon, il faudra se poser la question de la resynchronisation des données. Généralement, l'origine aura été restaurée dans un état antérieur à celui déjà répliqué : les données répliquées à présent absentes de l'origine sont-elles toutes à conserver ? Comment gérer les clés primaires qui vont souvent entrer en conflit ?

Restauration d'une souscription

Pour une souscription, l'ordre SQL dans la sauvegarde est :

```
CREATE SUBSCRIPTION subscr_t3_2
  CONNECTION 'port=5444 user=logrepli dbname=b1'
  PUBLICATION publi_t3_2
  WITH (connect = false, slot_name = 'subscr_t3_2');
```

Contrairement à l'ordre exécuté manuellement à la création, celui-ci précise le nom du slot de réplication (au cas où il aurait été personnalisé) et désactive la connexion immédiate. Cette désactivation a pour effet de désactiver la souscription, de ne pas créer le slot de réplication et d'empêcher la copie initiale des données (dont nous n'avons pas besoin étant donné que nous les avons dans la sauvegarde, au moins en partie).

Une réplication restaurée est donc par défaut inactive.

Une fois la sauvegarde restaurée et les vérifications nécessaires effectuées, il est possible d'activer la souscription et de la rafraîchir :

```
ALTER SUBSCRIPTION subscr_complete ENABLE ;
ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION ;
```

Ces opérations sont obligatoirement manuelles.

La restauration logique d'un abonnement revient à en créer un nouveau, et ne permet pas de savoir où la copie s'était arrêtée auparavant dans le flux des transactions : la copie des données sera intégralement relancée.
Sans autre opération, et si le contenu des tables répliquées a été restauré, le contenu déjà présent bloquera la réplication (s'il y a une clé primaire) ou de se retrouver en double (sans clé primaire). Il peut être plus simple de ne pas restaurer les données sur la destination, ou de tronquer les tables avant le `ENABLE`, pour reprendre la copie à zéro.
Une alternative est de ne pas effectuer la copie initiale :

```
ALTER SUBSCRIPTION nom_abonnement REFRESH PUBLICATION
WITH (copy_data = false) ;
```

auquel cas on risque d'avoir un « trou » entre les données restaurées et celles qui vont apparaître sur le publieur ; qu'il faudra corriger à la main dans les nombreux cas où cela est important.

1.5.14 Gérer les bascules & les restaurations physiques

Comme pour la réplication physique :

- Sauvegarde PITR
  - publications et souscriptions
  - slots ?
- Slots perdus et « trous » dans la réplication si :
  - bascule origine
  - restauration origine
  - restauration destination
- Contrôle délicat !
  - interdire les écritures à ces moments ?
- Bascule de la destination
  - si propre, devrait mieux se passer

Pendant ces opérations, il est fortement conseillé d'interdire les écritures dans les tables répliquées pour avoir une vision claire de ce qui a été répliqué et ne l'a pas été. Les slots doivent souvent être reconstruits, et il faut éviter que les tables soient modifiées entre-temps.

Restauration de l'instance d'origine :

Cela dépend de la méthode de sauvegarde/restauration utilisée, mais la restauration du serveur origine ne conserve généralement pas les slots de réplication (qui sont périmés de toute façon).

Il faudra recréer les slots, peut-être recréer les souscriptions, et pendant ce temps des trous dans les données répliquées peuvent apparaître, qu'il faudra vérifier ou corriger manuellement.

Bascule de l'instance d'origine :

Ici, l'instance d'origine est arrêtée et un de ses secondaires est promu comme nouveau serveur principal. Les slots de réplication étant propres à une instance, il ne seront pas disponibles immédiatement sur la nouvelle origine. Il faudra aussi reparamétrer la connexion des abonnements.

Il y a donc à nouveau un risque sérieux de perdre au moins quelques données répliquées.

Restauration de l'instance de destination :

Un slot de réplication sur l'origine garantit seulement que les journaux seront toujours disponibles pendant une indisponibilité du souscripteur. Ils ne permettent pas de revenir sur des données déjà répliquées.

En redémarrant, les abonnements vont tenter de se raccrocher au slot de réplication de l'origine, ce qui fonctionnera, mais ils ne recevront que des données jamais répliquées. Là encore des « trous »

dans les données répliquées peuvent apparaître si l'instance destination n'a pas été restaurée dans un état suffisamment récent !

Bascule de l'instance destination :

C'est le cas le plus favorable. Si la bascule s'est faite proprement sans perte entre l'ancienne destination et la nouvelle, il ne devrait pas y avoir de perte de données répliquées. Cela devra tout de même faire partie des contrôles.

1.5.15 Réplication logique depuis un secondaire comme origine

- Depuis PostgreSQL 16
- `wal_level = logical` sur le secondaire/origine
- Création de la publication toujours sur le primaire
- Le secondaire porte le slot et décode
- Latence supplémentaire

La réplication logique ne peut se faire depuis un serveur secondaire (lui-même en réplication physique) que si l'origine de la réplication logique (secondaires et primaire) est au moins sous PostgreSQL 16, mais pas forcément le destinataire.

La situation devient plus complexe car on a deux modes de réplication (physique et logique), et il faut bien distinguer les trois instances primaire, secondaire/origine et destination.

Rappelons que les slots de réplication sont propres à une instance, qu'elle soit secondaire ou pas. Le slot de réplication logique et le `walsender` associé seront donc créés sur le serveur secondaire, qui procédera au décodage logique, stockera les journaux au besoin, etc. et enverra les informations à l'instance destinataire.

Comme le secondaire est en lecture, il faudra continuer à créer et détruire les publications sur le primaire.

Latence :

Évidemment, la réplication logique sera tributaire des délais (voire pause) dans le rejeu des journaux sur le secondaire, et la latence en souffrira. En cas de complète inactivité, cette fonction, exécutée sur le primaire, permet d'envoyer dans les journaux le nécessaire pour une synchronisation des réplications logiques :

```
SELECT pg_log_standby_snapshot() ;
```

Promotion :

Si le serveur secondaire origine est promu et devient un primaire, la réplication logique qui y est attachée fonctionne toujours.

1.5.16 Combien de réplications logiques ?

- 1 publication logique = 1 walsender+1 slot par abonné
- Chaque worker doit décoder les WAL
  - Attention au CPU et à la RAM !
- Risques de slots bloqués
- Contournements :
  - regrouper les réplications
  - réplication depuis un secondaire
  - `streaming = on`

Il est possible de monter à plusieurs dizaines, voire une centaine, le nombre de réplications logiques depuis un même serveur origine.

Chaque publication nécessite donc un `walsender` et un slot par abonné sur la source. Sur la cible apparaît un `logical replication apply worker` pour chaque abonnement. D'autres processus peuvent aussi apparaître pendant la synchronisation (`table synchronization worker`) ou en cas d'application en parallèle des transactions (`parallel apply worker`).

On évitera donc de multiplier les réplications inutiles (par exemple en répliquant un schéma entier plutôt que chaque table séparément).

Il faudra parfois monter `max_wal_senders` et `max_replication_slots` sur le publieur, mais il n'y a pas besoin de monter `max_connections`. Sur la cible, il faudra vérifier `max_replication_slots`, `max_logical_replication_workers`, voire `max_worker_processes`. `max_sync_workers_per_subscription` ou `max_parallel_apply_workers_per_subscription` (initialisation et parallélisation) peuvent consommer encore d'autres workers. Prévoir donc de la marge.

Et il faut être conscient que chaque worker doit décoder le flux de journaux communs, ne serait-ce que pour chercher ce qui l'intéresse. Il y a donc un coût en CPU et en RAM, voire en disque lors de transactions longues. Dans ce dernier cas, il faudra arbitrer entre l'impact sur la RAM et la création de fichiers temporaires sur disque avec le paramètre `logical_decoding_work_mem`. et penser à activer l'option `streaming = on`.

S'il n'y a pas d'abonnement actif sur les tables répliquées, la consommation de ressources sera faible. Par contre, la présence de nombreux serveurs abonnés augmente le risque que certains slots bloquent le recyclage des journaux (penser à `max_slot_wal_keep_size`).

Depuis PostgreSQL 16, l'utilisation d'un serveur secondaire dédié est une option intéressante pour ce cas d'usage.

Voir cette discussion sur plgsql-general[7].

[7] https://www.postgresql.org/message-id/flat/CAAkB0aDof-atNom4qO_RGefgPDib3ukEzX1B9Tva11nusWMriA%40mail.gmail.com

1.6 SUPERVISION

- Méta-données
- Statistiques
- Outils

1.6.1 Catalogues systèmes - méta-données

- `pg_publication`
  - définition des publications
  - `\dRp` sous psql
- `pg_publication_tables`
  - tables ciblées par chaque publication
- `pg_subscription`
  - définition des souscriptions
  - `\dRs` sous psql

Dans la base origine :

Le catalogue système `pg_publication` contient la liste des publications, avec leur méta-données :

```
TABLE pg_publication ;
```

```
     pubname     | pubowner | puballtables | pubinsert | pubupdate | pubdelete
-----------------+----------+--------------+-----------+-----------+-----------
 publi_complete  |       10 | t            | t         | t         | t
 publi_partielle |       10 | f            | t         | t         | t
 publi_t3_1      |       10 | f            | t         | t         | t
```

Le catalogue système `pg_publication_tables` contient une ligne par table par publication :

```
TABLE pg_publication_tables ;
```

```
     pubname     | schemaname | tablename
-----------------+------------+-----------
 publi_complete  | public     | t1
```

```
 publi_complete  | public     | t3_1
 publi_complete  | public     | t3_2
 publi_complete  | public     | t2
 publi_complete  | public     | t3_3
 publi_complete  | public     | t4
 publi_partielle | public     | t1
 publi_partielle | public     | t2
 publi_t3_1      | public     | t3_1
```

On peut en déduire deux versions abrégées :

- la liste des tables par publication :

```
SELECT pubname, array_agg(tablename ORDER BY tablename) AS tables_list
FROM pg_publication_tables
GROUP BY pubname  ORDER BY pubname ;
```

```
     pubname     |         tables_list
-----------------+-----------------------------
 publi_complete  | {t1,t2,t3_1,t3_2,t3_3,t4,t5}
 publi_partielle | {t1,t2}
 publi_t3_1      | {t3_1}
```

- la liste des publications par table :

```
SELECT tablename, array_agg(pubname ORDER BY pubname) AS publications_list
FROM pg_publication_tables
GROUP BY tablename
ORDER BY tablename ;
```

```
 tablename |        publicationss_list
-----------+----------------------------------
 t1        | {publi_complete,publi_partielle}
 t2        | {publi_complete,publi_partielle}
 t3_1      | {publi_complete,publi_t3_1}
 t3_2      | {publi_complete}
 t3_3      | {publi_complete}
 t4        | {publi_complete}
 t5        | {publi_complete}
```

Dans la base destinataire :

Enfin, il y a aussi un catalogue système contenant la liste des souscriptions :

```
\x
Expanded display is on.
SELECT * FROM pg_subscription;
-[ RECORD 1 ]---+----------------------------------
subdbid         | 16443
subname         | subscr_t3_2
subowner        | 10
subenabled      | t
subconninfo     | port=5444 user=logrepli dbname=b1
subslotname     | subscr_t3_2
subsynccommit   | off
subpublications | {publi_t3_2}
```

1.6.2 Vues statistiques

- `pg_stat_replication`
  - statut de réplication
- `pg_replication_slots`
  - slots de réplication : statut
- `pg_stat_replication_slots` (v14)
  - volumes écrits/envoyés en *streaming* via les slots de réplication logique
- `pg_stat_subscription`
  - état des souscriptions
- `pg_replication_origin_status`
  - statut des origines de réplication
- `pg_stat_database_conflicts` (si origine est un secondaire)

Statut de la réplication :

Comme pour la réplication physique, le retard de réplication est visible ou calculable en utilisant les informations de la vue `pg_stat_replication` sur le serveur origine :

```
SELECT * FROM pg_stat_replication ;
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 18200
usesysid         | 16442
usename          | logrepli
application_name | subscr_t3_1
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 10:31:01.13489+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
```

```
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 26606
usesysid         | 16442
usename          | logrepli
application_name | subscr_partielle
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 10:02:28.196654+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 3 ]----+------------------------------
pid              | 15127
usesysid         | 16442
usename          | logrepli
application_name | subscr_complete
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 11:44:04.267249+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

La vue `pg_replication_slots` est complémentaire de `pg_stat_replication` car elle contient des statuts :

```
SELECT * FROM pg_replication_slots ;

-[ RECORD 1 ]-------+--------------------------
slot_name           | abonnement_16001
plugin              | pgoutput
slot_type           | logical
datoid              | 16388
database            | editeur
temporary           | f
active              | t
active_pid          | 1711279
```

```
xmin                |
catalog_xmin        | 2388
restart_lsn         | 1/FF7E0670
confirmed_flush_lsn | 1/FF7E06A8
wal_status          | reserved
safe_wal_size       |
two_phase           | f
conflicting         | f
```

Le dernier champ (apparu en version 16) indique une invalidation à cause d'un conflit de réplication.

Depuis la version 14, une autre vue, `pg_stat_replication_slots` (description complète dans la documentation[8], permet de suivre les volumétries (octets, nombre de transactions) écrites sur disque (*spilled*) ou envoyées en *streaming* :

```
SELECT * FROM pg_stat_replication_slots \gx

-[ RECORD 1 ]+-----------
slot_name    | abonnement
spill_txns   | 3
spill_count  | 7
spill_bytes  | 412435584
stream_txns  | 0
stream_count | 0
stream_bytes | 0
total_txns   | 30467
total_bytes  | 161694536
stats_reset  |
```

Souscriptions :

L'état des souscriptions est disponible sur les serveurs destination à partir de la vue `pg_stat_subscription` :

```
SELECT * FROM pg_stat_subscription ;

-[ RECORD 1 ]---------+------------------------------
subid                 | 16573
subname               | subscr_t3_2
pid                   | 18893
relid                 |
received_lsn          | 0/168A748
last_msg_send_time    | 2017-12-20 10:36:13.315798+01
last_msg_receipt_time | 2017-12-20 10:36:13.315849+01
latest_end_lsn        | 0/168A748
latest_end_time       | 2017-12-20 10:36:13.315798+01
```

Conflits de réplication :

Depuis PostgreSQL 16, un secondaire peut être origine d'une réplication logique. Comme dans une réplication physique classique, il est possible d'avoir des conflits de réplication (le primaire envoie des modifications sur des lignes que le secondaire aurait voulu garder pour ses abonnés).

[8] https://docs.postgresql.fr/current/monitoring-stats.html#MONITORING-PG-STAT-REPLICATION-SLOTS-VIEW

Dans une réplication physique classique, le conflit entraîne juste l'arrêt de requêtes sur le secondaire. Mais si le secondaire est origine d'une réplication logique, celle-ci peut décrocher. Le problème apparaît surtout lors d'une modification dans le schéma de données. Le message suivant apparaît dans les traces de la destination si un slot de réplication a été invalidé suite à ce conflit :

```
LOG:  logical replication apply worker for subscription "abonnement_decompte" has
↪  started
ERROR:  could not start WAL streaming: ERROR:  can no longer get changes from
↪  replication slot "decompte_abonnement_16001"
DETAIL:  This slot has been invalidated because it was conflicting with recovery.
```

L'option `disable_on_error` sur la souscription permet d'éviter qu'elle ne tente de se reconnecter en boucle. Le plus propre est de sécuriser la réplication entre primaire et secondaire en passant `hot_standby_feedback` à `on` sur le secondaire (ce qui doit toujours se sécuriser sur le primaire en mettant un seuil dans `max_slot_wal_keep_size`).

1.6.3 Outils de supervision

- `check_pgactivity`
  - `replication_slots`
- `check_postgres`
  - `same_schema`

Il est possible de surveiller le retard de réplication via l'état des slots de réplication, comme le propose l'outil `check_pgactivity` (disponible sur github[9] ou les paquets des dépôts). Ici, il n'y a pas de retard sur la réplication, pour les trois slots :

```
$ ./check_pgactivity -s replication_slots -p 5441 -F human

Service         : POSTGRES_REPLICATION_SLOTS
Returns         : 0 (OK)
Message         : Replication slots OK
Perfdata        : subscr_complete_wal=0File
Perfdata        : subscr_complete_spilled=0File
Perfdata        : subscr_t3_1_wal=0File
Perfdata        : subscr_t3_1_spilled=0File
Perfdata        : subscr_partielle_wal=0File
Perfdata        : subscr_partielle_spilled=0File
```

Faisons quelques insertions après l'arrêt de s3 (qui correspond à la souscription pour la réplication partielle) :

[9] https://github.com/OPMDG/check_pgactivity

```
INSERT INTO t1 SELECT generate_series(1000000, 2000000);
```

L'outil détecte bien que le slot `subscr_partielle` a un retard conséquent (8 journaux de transactions) et affiche le nombre de fichiers de débordement créés :

```
$ ./check_pgactivity -s replication_slots -p 5441 -F human

Service        : POSTGRES_REPLICATION_SLOTS
Returns        : 0 (OK)
Message        : Replication slots OK
Perfdata       : subscr_t3_1_wal=8File
Perfdata       : subscr_t3_1_spilled=0File
Perfdata       : subscr_partielle_wal=8File
Perfdata       : subscr_partielle_spilled=9File
Perfdata       : subscr_complete_wal=8File
Perfdata       : subscr_complete_spilled=9File
```

Il est aussi possible d'utiliser l'action `same_schema` avec l'outil `check_postgres` (disponible aussi sur github[10]) pour détecter des différences de schémas entre deux serveurs (l'origine et une destination).

[10] https://github.com/bucardo/check_postgres/

1.7 MIGRATION MAJEURE PAR RÉPLICATION LOGIQUE

- Possible entre versions 10 et supérieures
- Remplace Slony, Bucardo…
- Bascule très rapide
- Et retour possible
- Des limitations

La réplication logique rend possible une migration entre deux instances de version majeure différente avec une indisponibilité très courte. La base à migrer doit bien sûr être en version 10 ou supérieure. C'était déjà possible avec des outils de réplication par trigger comme Slony ou Bucardo. Ces outils externes ne sont maintenant plus nécessaires. (Noter que Slony en particulier reste parfaitement utilisable et recommandable, et sert encore pour nombre de migrations).

Le principe est de répliquer une base à l'identique alors que la production tourne. Lors de la bascule, il suffit d'attendre que les dernières données soient répliquées, ce qui peut être très rapide, et de connecter les applications au nouveau serveur. La réplication peut alors être inversée pour garder l'ancienne production synchrone, permettant de rebasculer dessus en cas de problème sans perdre les données modifiées depuis la bascule.

Les étapes sont :

- copie des structures et des objets globaux concernés ;
- mise en place d'une publication sur la source et d'un abonnement sur la cible ;
- suivi de la réplication (*lag* entre les serveurs) ;
- arrêt des connexions applicatives ;
- attente de la fin de la réplication logique ;
- isolation de la base source des connexions applicatives ;
- synchronisation manuelle des valeurs des séquences (non répliquées) ;
- suppression de la publication et de l'abonnement ;
- éventuellement création d'un abonnement et d'une publication en sens inverse ;
- ouverture de la base cible aux applications.

Les restrictions liées à la réplication logique subsistent :

- les modifications de schéma effectuées pendant la synchronisation ne sont pas répliquées (cela est problématique si l'application elle-même effectue du DDL sur des tables non temporaires) ;
- les `TRUNCATE` depuis une base v10 ne sont pas répliqués ;
- les *Large objects* et les séquences ne sont pas répliqués ;
- il est fortement conseillé que toutes les tables aient des clés primaires ;
- la réplication fonctionnant uniquement pour les tables « de base », les vues matérialisées sont à reconstruire sur la cible ;
- jusqu'en version 13, le partitionnement doit être identique des deux côtés.

Cette méthode reste donc plus complexe et fastidieuse qu'une migration par `pg_dump` / `pg_restore` ou `pg_upgrade`.

1.8 RAPPEL DES LIMITATIONS DE LA RÉPLICATION LOGIQUE NATIVE

- Pas de réplication : DDL, LO, valeurs de séquence
- Pas de réplication des tables partitionnées (< v13)
  - mais réplication possible des partitions
- Pas de réplication vers une table partitionnée (< v13)
- Contraintes d'unicité obligatoires pour les `UPDATE` / `DELETE`
- Coût CPU, disque, RAM
- Réplication déclenchée uniquement lors du `COMMIT` (< v14)
- Que faire lors des restaurations/bascules ?

Rappelons que la réplication logique native ne réplique pas les ordres DDL. Elle se base uniquement au niveau des données (donc les ordres DML, et `TRUNCATE`). Les valeurs des séquences et les Larges Objects ne sont pas répliqués.

Avant la version 13, il n'était pas possible d'ajouter une table partitionnée à une publication pour qu'elle et ses partitions soient répliquées. Il fallait ajouter chaque partition individuellement. Cette limitation a été supprimée en version 13. Toujours avant la version 13, il n'était pas possible d'envoyer des données vers une table partitionnée.

Pour les versions inférieures à 14, la réplication logique n'est déclenchée que lors d'un `COMMIT`, avec un délai de réplication pour les transactions longues. Pensez à `streaming=on`.

Enfin, la réplication logique doit tenir compte des cas de restauration, ou bascule, d'une des instances impliquées. Le concept de flux unique de transaction unique ne s'applique plus ici, et il n'est pas prévu de moyen pour garantir que la réplication se fera sans aucune perte ou risque de doublon. La mise en place de la réplication logique doit toujours prévoir ce qu'il faudra faire dans ce cas.

Certaines applications supporteront cette limite. Dans d'autres, il sera plus ou moins facile de reprendre la réplication à zéro. Parfois, une réconciliation manuelle sera nécessaire (la présence de clés primaires peut grandement aider). Dans certains cas, ce problème peut devenir bloquant ou réclamer des développements.

1.9 OUTILS DE RÉPLICATION LOGIQUE EXTERNE

- Conseillé : Slony
- Non conseillés: Londiste, Bucardo

Slony est un outil que nous utilisons régulièrement pour des montées de versions majeures.

Nous n'avons pas rencontré Londiste et Bucardo en production depuis plusieurs années. Ils semblent encore maintenus mais le développement s'est depuis longtemps figé, et nous ne conseillons pas leur utilisation en production.

1.9.1 Slony : Carte d'identité

- Projet libre (`BSD`)
- Asynchrone / Asymétrique
- Diffusion des résultats (triggers)

Slony[11] est un très ancien projet libre de réplication pour PostgreSQL. C'était l'outil de choix avant l'arrivée de la réplication native dans PostgreSQL.

1.9.2 Slony : Fonctionnalités

- Réplication de tables sélectionnées
- Procédures de bascule
  - *switchover* / *switchback*
  - *failover* / *failback*

Slony permet de choisir les tables à répliquer. Il faudra ajouter à la réplication toute nouvelle table qui serait créée après sa mise en place.

[11] http://slony.info/

Les procédures de bascule chez Slony sont très simples. Il est ainsi possible de basculer un serveur primaire et son serveur secondaire autant de fois qu'on le souhaite, très rapidement, sans avoir à reconstruire quoi que ce soit.

1.9.3 Slony : Technique

- Réplication basée sur des triggers
- Démons externes, écrits en C
- Le primaire est un **provider**
- Les secondaires sont des **subscribers**

Slony est un système de réplication asynchrone/asymétrique, donc un seul primaire et un ou plusieurs serveurs secondaires mis à jour à intervalle régulier. La récupération des données modifiées se fait par des triggers, qui stockent les modifications dans des tables propres à Slony avant leur transfert vers les secondaires. Un système de démon récupère les données pour les envoyer sur les secondaires et les applique.

Les démons et les triggers sont écrits en C, ce qui permet à Slony d'être très performant.

Au niveau du vocabulaire utilisé, le primaire est souvent appelé un « provider » (il fournit les données aux serveurs secondaires) et les secondaires sont souvent des « subscribers » (ils s'abonnent au flux de réplication pour récupérer les données modifiées).

1.9.4 Slony : Points forts

- Choix des tables et séquences à répliquer
- Indépendance des versions de PostgreSQL
- Technique de propagation des DDL
- Robustesse

Slony dispose de nombreux points forts, parfois liés au simple fait qu'il s'agit d'une réplication logique.

Il permet de ne répliquer qu'un sous-ensemble des objets d'une instance : pas forcément toutes les bases, pas forcément toutes les tables d'une base particulière, etc.

Le serveur primaire et les serveurs secondaires n'ont pas besoin d'utiliser la même version majeure de PostgreSQL. Il est donc possible de mettre à jour en plusieurs étapes (plutôt que tous les serveurs à la fois). Cela facilite aussi le passage à une version majeure ultérieure.

Même si la réplication des DDL est impossible, leur envoi aux différents serveurs est possible grâce à un outil fourni. Tous les systèmes de réplication par triggers ne peuvent pas en dire autant.

1.9.5 Slony : Limites

- Le réseau doit être fiable : peu de *lag*, pas ou peu de coupures
- Supervision délicate
- Modifications de schémas complexes

Slony peut survivre avec un réseau coupé. Cependant, il n'aime pas quand le réseau passe son temps à être disponible puis indisponible. Les démons slon ont tendance à croire qu'ils sont toujours connectés alors que ce n'est plus le cas.

Superviser Slony n'est possible que via une table statistique appelée `sl_status`. Elle fournit principalement deux informations : le retard en nombre d'événements de synchronisation et la date de la dernière synchronisation.

Enfin, la modification de la structure d'une base, même si elle est simplifiée avec le script fourni, n'est pas simple, en tout cas beaucoup moins simple que d'exécuter une requête DDL seule.

1.9.6 Slony : Utilisations

- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc.)
- Migrations de versions majeures avec indisponibilité réduite

Bien que la réplication logique soit arrivée avec PostgreSQL 10, Slony garde son utilité pour les nombreuses instances des versions précédentes.

Slony peut se révéler intéressant car il est possible d'avoir des tables de travail en écriture sur le secondaire avec Slony. Il est aussi possible d'ajouter des index sur le secondaire qui ne seront pas présents sur le serveur primaire (on évite donc la charge de maintenance des index par le serveur primaire, tout en permettant de bonnes performances pour la création des rapports).

Il est encore fréquent d'utiliser Slony pour des migrations entre deux versions majeures avec une indisponibilité réduite, voire avec un retour en arrière possible.

Pour plus d'informations sur Slony, n'hésitez pas à lire un de nos articles disponibles sur notre site[12]. Le thème des réplications complexes a aussi été abordé lors du PostgreSQL Sessions 2012[13].

[12] https://www.dalibo.org/hs44_slony_la_replication_des_donnees_par_trigger
[13] https://www.postgresql-sessions.org/assets/archives/pgsessions3_slony.pdf

1.10 CONCLUSION

- Réplication logique simple et pratique
  - …avec ses subtilités

La réplication logique de PostgreSQL apparue en version 10 continue de s'améliorer avec les versions. Elle complète la réplication physique sans la remplacer.

Les cas d'utilisation sont nombreux, mais la supervision est délicate et il faut prévoir les sauvegardes/restaurations et bascules.

1.10.1 Questions

N'hésitez pas, c'est le moment !

1.11 QUIZ

https://dali.bo/w5_quiz

1.12 TRAVAUX PRATIQUES

1.12.1 Pré-requis

En préalable, nettoyer les instances précédemment créés sur le serveur.

Ensuite, afin de réaliser l'ensemble des TP, créer 4 nouvelles instances PostgreSQL « instance[1-4] », en leur attribuant des ports différents :

```
# systemctl stop instance1
# systemctl stop instance2
# systemctl stop instance3
# systemctl stop instance4

# rm -rf /var/lib/pgsql/16/instance1
# rm -rf /var/lib/pgsql/16/instance2
# rm -rf /var/lib/pgsql/16/instance3
# rm -rf /var/lib/pgsql/16/instance4

# export PGSETUP_INITDB_OPTIONS='--data-checksums'
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance1
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance2
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance3
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance4

# sed -i "s/#port = 5432/port = 5433/" /var/lib/pgsql/16/instance2/postgresql.conf
# sed -i "s/#port = 5432/port = 5434/" /var/lib/pgsql/16/instance3/postgresql.conf
# sed -i "s/#port = 5432/port = 5435/" /var/lib/pgsql/16/instance4/postgresql.conf

# systemctl start instance1
# systemctl start instance2
# systemctl start instance3
# systemctl start instance4

$ ps -o pid,cmd fx

  PID CMD
 7077 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance4/
 7079  \_ postgres: logger
 7081  \_ postgres: checkpointer
 7082  \_ postgres: background writer
 7083  \_ postgres: walwriter
 7084  \_ postgres: autovacuum launcher
 7085  \_ postgres: logical replication launcher
 7056 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance3/
 7058  \_ postgres: logger
 7060  \_ postgres: checkpointer
 7061  \_ postgres: background writer
 7062  \_ postgres: walwriter
 7063  \_ postgres: autovacuum launcher
 7064  \_ postgres: logical replication launcher
 7035 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance2/
 7037  \_ postgres: logger
 7039  \_ postgres: checkpointer
 7040  \_ postgres: background writer
```

```
 7041  \_ postgres: walwriter
 7042  \_ postgres: autovacuum launcher
 7043  \_ postgres: logical replication launcher
 7015 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance1/
 7016  \_ postgres: logger
 7018  \_ postgres: checkpointer
 7019  \_ postgres: background writer
 7020  \_ postgres: walwriter
 7021  \_ postgres: autovacuum launcher
 7022  \_ postgres: logical replication launcher
```

Le schéma de la base **b1** de l'instance origine (**instance1**) est le suivant. Noter que la table `t3` est partitionnée.

```
CREATE TABLE t1 (id_t1 serial, label_t1 text);
CREATE TABLE t2 (id_t2 serial, label_t2 text);

CREATE TABLE t3 (id_t3 serial, label_t3 text, clepartition_t3 integer)
  PARTITION BY LIST (clepartition_t3);
CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES IN (1);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES IN (2);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES IN (3);
CREATE TABLE t3_4 PARTITION OF t3 FOR VALUES IN (4);

INSERT INTO t1 SELECT i, 't1, ligne '||i FROM generate_series(1, 100) i;

INSERT INTO t2 SELECT i, 't2, ligne '||i FROM generate_series(1, 1000) i;

INSERT INTO t3 SELECT i, 't3, ligne '||i, 1 FROM generate_series(  1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 3 FROM generate_series(301, 600) i;

ALTER TABLE t1 ADD PRIMARY KEY(id_t1);
ALTER TABLE t2 ADD PRIMARY KEY(id_t2);
ALTER TABLE t3 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

1.12.2 Réplication complète d'une base

But : Mettre en place la réplication complète d'une base avec la réplication logique.

Pour répliquer toute la base **b1** sur le serveur **instance2** :

Sur **instance1**, créer l'utilisateur de réplication **logrepli** et lui donner les droits de lecture sur les tables de la base **b1**.

Sur **instance1**, modifier la configuration du paramètre `wal_level` dans le fichier `postgresql.conf`.

Sur **instance1**, modifier la configuration des connexions dans le fichier `pg_hba.conf`.

Redémarrer **instance1**.

Sur **instance2**, créer l'utilisateur de réplication.

Sur **instance2**, créer la base **b1**.

Sur **instance2**, ajouter dans la base **b1** les tables répliquées (sans contenu).

Sur **instance1**, créer la publication pour toutes les tables.

Sur **instance2**, créer la souscription.

Vérifier sur **instance1**, dans la vue `pg_stat_replication` l'état de la réplication logique.

Sur **instance2**, consulter `pg_stat_subscription`.

Vérifier que les tables ont le même contenu que sur **instance1** et que les modifications sont également répliquées.

1.12.3 Réplication partielle d'une base

But : Mettre en place la réplication partielle d'une base avec la réplication logique.

On veut répliquer uniquement les tables `t1` et `t2` de la base **b1** sur le serveur **instance3**.

Sur **instance1**, créer la publication pour `t1` et `t2`.

Sur **instance3**, créer la base **b1**, les tables à répliquer, puis souscrire à la nouvelle publication de **instance1**.

Vérifier sur **instance1**, dans la vue `pg_stat_replication` l'état de la réplication logique.

Sur **instance3**, consulter `pg_stat_subcription`.

1.12.4 Réplication croisée

But : Mettre en place une réplication croisée avec la réplication logique.

Pour répliquer la partition `t3_1` du serveur **instance1** vers le serveur **instance4**, puis répliquer la partition `t3_2` du serveur **instance4** vers le serveur **instance2** :

Sur **instance1**, créer la publication pour la partition `t3_1`.

Sur **instance4**, créer l'utilisateur de réplication.

Sur **instance4**, souscrire à cette nouvelle publication de **instance1**. Pour créer la table `t3_1`, il faut aussi créer la table mère `t3`.

Sur **instance4**, adapter la valeur du paramètre `wal_level` dans `postgresql.conf`.

Sur **instance4**, adapter les autorisations dans `pg_hba.conf` pour permettre une réplication depuis **instance4**.

Redémarrer **instance4**.

Sur **instance4**, créer la publication pour `t3_4`. Il faudra importer la partition `t3_4` et donner les droits de lecture à **logrepli**.

Sur **instance1**, souscrire à cette nouvelle publication de **instance4**.

Insérer des données dans `t3_4` sur **instance4** et vérifier que la réplication se fait de **instance4** à **instance1**.

1.12.5 Réplication et partitionnement

But : Mettre en évidence des particularités de la réplication logique et du partitionnement.

Voici un exemple de réplication entre des tables qui n'ont pas le même schéma de partitionnement :

- Sur **instance1**, créer une base `bench_part`.

- Sur **instance2**, créer une base `bench_part`.

- Sur **instance1**, utiliser pgbench pour créer la table `pgbench_account`
- avec un partitionnement de type *hash* et cinq partitions.

- Sur **instance2**, utiliser pgbench pour créer la table `pgbench_account`
- avec un partitionnement de type *range* et trois partitions,
- mais sans insérer de données.

- Sur **instance1**, autoriser l'utilisateur de réplication à accéder aux tables.
- Créer une publication pour toutes les tables de la base.

- Sur **instance2**, créer la souscription associée. Que constatez-vous ?

- Sur **instance1**, supprimer la publication et la recréer avec l'option `publish_via_partition_root`.

- Sur **instance2**, recréer la souscription.

- Sur **instance1** et **instance2**, compter les lignes dans chaque partition de `pgbench_accounts`. Qu'observez-vous ?

1.13 TRAVAUX PRATIQUES (SOLUTIONS)

1.13.1 Pré-requis

En préalable, nettoyer les instances précédemment créés sur le serveur.

Ensuite, afin de réaliser l'ensemble des TP, créer 4 nouvelles instances PostgreSQL « instance[1-4] », en leur attribuant des ports différents :

```
# systemctl stop instance1
# systemctl stop instance2
# systemctl stop instance3
# systemctl stop instance4

# rm -rf /var/lib/pgsql/16/instance1
# rm -rf /var/lib/pgsql/16/instance2
# rm -rf /var/lib/pgsql/16/instance3
# rm -rf /var/lib/pgsql/16/instance4

# export PGSETUP_INITDB_OPTIONS='--data-checksums'
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance1
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance2
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance3
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance4

# sed -i "s/#port = 5432/port = 5433/" /var/lib/pgsql/16/instance2/postgresql.conf
# sed -i "s/#port = 5432/port = 5434/" /var/lib/pgsql/16/instance3/postgresql.conf
# sed -i "s/#port = 5432/port = 5435/" /var/lib/pgsql/16/instance4/postgresql.conf

# systemctl start instance1
# systemctl start instance2
# systemctl start instance3
# systemctl start instance4

$ ps -o pid,cmd fx

  PID CMD
 7077 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance4/
 7079  \_ postgres: logger
 7081  \_ postgres: checkpointer
 7082  \_ postgres: background writer
 7083  \_ postgres: walwriter
 7084  \_ postgres: autovacuum launcher
 7085  \_ postgres: logical replication launcher
 7056 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance3/
 7058  \_ postgres: logger
 7060  \_ postgres: checkpointer
 7061  \_ postgres: background writer
 7062  \_ postgres: walwriter
 7063  \_ postgres: autovacuum launcher
 7064  \_ postgres: logical replication launcher
 7035 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance2/
 7037  \_ postgres: logger
 7039  \_ postgres: checkpointer
 7040  \_ postgres: background writer
```

```
 7041  \_ postgres: walwriter
 7042  \_ postgres: autovacuum launcher
 7043  \_ postgres: logical replication launcher
 7015 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance1/
 7016  \_ postgres: logger
 7018  \_ postgres: checkpointer
 7019  \_ postgres: background writer
 7020  \_ postgres: walwriter
 7021  \_ postgres: autovacuum launcher
 7022  \_ postgres: logical replication launcher
```

Le schéma de la base **b1** de l'instance origine (**instance1**) est le suivant. Noter que la table `t3` est partitionnée.

```
CREATE TABLE t1 (id_t1 serial, label_t1 text);
CREATE TABLE t2 (id_t2 serial, label_t2 text);

CREATE TABLE t3 (id_t3 serial, label_t3 text, clepartition_t3 integer)
  PARTITION BY LIST (clepartition_t3);
CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES IN (1);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES IN (2);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES IN (3);
CREATE TABLE t3_4 PARTITION OF t3 FOR VALUES IN (4);

INSERT INTO t1 SELECT i, 't1, ligne '||i FROM generate_series(1, 100) i;

INSERT INTO t2 SELECT i, 't2, ligne '||i FROM generate_series(1, 1000) i;

INSERT INTO t3 SELECT i, 't3, ligne '||i, 1 FROM generate_series(  1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 3 FROM generate_series(301, 600) i;

ALTER TABLE t1 ADD PRIMARY KEY(id_t1);
ALTER TABLE t2 ADD PRIMARY KEY(id_t2);
ALTER TABLE t3 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

1.13.2 Réplication complète d'une base

> Sur **instance1**, créer l'utilisateur de réplication **logrepli** et lui donner les droits de lecture sur les tables de la base **b1**.

```
CREATE ROLE logrepli LOGIN REPLICATION;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

> Sur **instance1**, modifier la configuration du paramètre `wal_level` dans le fichier `postgresql.conf`.

```
wal_level = logical
```

> Sur **instance1**, modifier la configuration des connexions dans le fichier `pg_hba.conf`.

```
host b1 logrepli 127.0.0.1/24 trust
```

Redémarrer **instance1**.

Sur **instance2**, créer l'utilisateur de réplication.

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

Sur **instance2**, créer la base **b1**.

```
$ createdb -p 5433 b1
```

Sur **instance2**, ajouter dans la base **b1** les tables répliquées (sans contenu).

```
$ pg_dump -p 5432 -s b1 | psql -p 5433 b1
```

Sur **instance1**, créer la publication pour toutes les tables.

```
CREATE PUBLICATION publi_complete FOR ALL TABLES;
```

Sur **instance2**, créer la souscription.

```
CREATE SUBSCRIPTION subscr_complete
  CONNECTION 'host=127.0.0.1 port=5432 user=logrepli dbname=b1'
  PUBLICATION publi_complete;
```

Vérifier sur **instance1**, dans la vue `pg_stat_replication` l'état de la réplication logique. Sur **instance2**, consulter `pg_stat_subscription`.

Sur **instance1** :

```
b1=# SELECT * FROM pg_stat_replication \gx

-[ RECORD 1 ]----+------------------------------
pid              | 7326
usesysid         | 16451
usename          | logrepli
application_name | subscr_complete
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 48094
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/195BF78
write_lsn        | 0/195BF78
flush_lsn        | 0/195BF78
replay_lsn       | 0/195BF78
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
```

Sur **instance2** :

```
b1=# select * from pg_stat_subscription \gx

-[ RECORD 1 ]---------+------------------------------
subid                 | 16521
subname               | subscr_complete
pid                   | 7325
relid                 |
received_lsn          | 0/195BF78
last_msg_send_time    | ...
last_msg_receipt_time | ...
latest_end_lsn        | 0/195BF78
latest_end_time       | ...
```

> Vérifier que les tables ont le même contenu que sur **instance1** et que les modifications sont également répliquées.

Toute opération d'écriture sur la base **b1** du serveur **instance1** est répliquée sur **instance2**.

Sur **instance1** :

```
b1=# INSERT INTO t1 VALUES (101, 't1, ligne 101');

INSERT 0 1

b1=# UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;

UPDATE 1

b1=# DELETE FROM t1 WHERE id_t1=11;

DELETE 1

b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);

 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

Sur **instance2** :

```
b1=# SELECT count(*) FROM t1;

 count
-------
   100

b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);

 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
```

1.13.3 Réplication partielle d'une base

On veut répliquer uniquement les tables `t1` et `t2` de la base **b1** sur **instance3**.

> Sur **instance1**, créer la publication pour `t1` et `t2`.

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2;
```

> Sur **instance3**, créer la base **b1**, les tables à répliquer, puis souscrire à la nouvelle publication de **instance1**.

```
$ psql -p 5434 -c "CREATE ROLE logrepli LOGIN REPLICATION;"
$ createdb -p 5434 b1
$ pg_dump -p 5432 -s -t t1 -t t2 b1 | psql -p 5434 b1
```

```
CREATE SUBSCRIPTION subscr_partielle
  CONNECTION 'host=127.0.0.1 port=5432 user=logrepli dbname=b1'
  PUBLICATION publi_partielle;
```

> Vérifier sur **instance1**, dans la vue `pg_stat_replication` l'état de la réplication logique.

Sur **instance1** :

```
b1=# SELECT * FROM pg_stat_replication \gx
-[ RECORD 1 ]----+------------------------------
pid              | 7326
usesysid         | 16451
usename          | logrepli
application_name | subscr_complete
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 48094
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/1965548
write_lsn        | 0/1965548
flush_lsn        | 0/1965548
replay_lsn       | 0/1965548
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
-[ RECORD 2 ]----+------------------------------
pid              | 7511
usesysid         | 16451
usename          | logrepli
application_name | subscr_partielle
client_addr      | 127.0.0.1
client_hostname  |
```

```
client_port      | 48124
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/1965548
write_lsn        | 0/1965548
flush_lsn        | 0/1965548
replay_lsn       | 0/1965548
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
```

Sur **instance3**, consulter `pg_stat_subcription`.

Sur **instance3** :

```
b1=# SELECT * FROM pg_stat_subscription \gx
-[ RECORD 1 ]---------+----------------------------
subid                 | 16431
subname               | subscr_partielle
pid                   | 7510
relid                 |
received_lsn          | 0/1965630
last_msg_send_time    | ...
last_msg_receipt_time | ...
latest_end_lsn        | 0/1965630
latest_end_time       | ...
```

1.13.4 Réplication croisée

Sur **instance1**, créer la publication pour la partition `t3_1`.

```
CREATE PUBLICATION publi_t3_1
  FOR TABLE t3_1;
```

Sur **instance4**, créer l'utilisateur de réplication.

```
$ psql -p 5435 -c "CREATE ROLE logrepli LOGIN REPLICATION;"
```

Sur **instance4**, souscrire à cette nouvelle publication de **instance1**. Pour créer la table `t3_1`, il faut aussi créer la table mère `t3`.

```
$ createdb -p 5435 b1
$ pg_dump  -p 5432 -s -t t3 -t t3_1 b1 | psql -p 5435 b1
```

```
CREATE SUBSCRIPTION subscr_t3_1
  CONNECTION 'host=127.0.0.1 port=5432 user=logrepli dbname=b1'
  PUBLICATION publi_t3_1;
```

> Sur **instance4**, adapter la valeur du paramètre `wal_level` dans `postgresql.conf`.

```
wal_level = logical
```

> Sur **instance4**, adapter les autorisations dans `pg_hba.conf` pour permettre une réplication depuis **instance4**.

```
host all logrepli 127.0.0.1/24 trust
```

> Redémarrer **instance4**.

> Sur **instance4**, créer la publication pour `t3_4`. Il faudra importer la partition `t3_4` et donner les droits de lecture à **logrepli**.

```
$ pg_dump -p 5432 -s -t t3_4 b1 | psql -p 5435 b1
```

```
GRANT SELECT ON t3_4 TO logrepli;

CREATE PUBLICATION publi_t3_4
  FOR TABLE t3_4;
```

> Sur **instance1**, souscrire à cette nouvelle publication de **instance4**.

```
CREATE SUBSCRIPTION subscr_t3_4
  CONNECTION 'host=127.0.0.1 port=5435 user=logrepli dbname=b1'
  PUBLICATION publi_t3_4;
```

> Insérer des données dans `t3_4` sur **instance4** et vérifier que la réplication se fait de **instance4** à **instance1**.

Sur **instance1** :

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;

 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)

b1=# INSERT INTO t3 VALUES (1001, 't3, ligne 1001', 1);

INSERT 0 1

b1=# SELECT * FROM t3 WHERE id_t3>999;

 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

Sur **instance4** :

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

```
b1=# INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 4);

INSERT 0 1

b1=# SELECT * FROM t3 WHERE id_t3 > 999;

 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               4
```

Sur **instance1** :

```
b1=# SELECT * FROM t3 WHERE id_t3>999;

 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               4
```

1.13.5 Réplication et partitionnement

- Sur **instance1**, créer une base `bench_part`.

```
createdb --port 5432 bench_part
```

- Sur **instance2**, créer une base `bench_part`.

```
createdb --port 5433 bench_part
```

- Sur **instance1**, utiliser pgbench pour créer la table `pgbench_account`
- avec un partitionnement de type *hash* et cinq partitions.

```
pgbench --initialize \
        --partition-method=hash \
        --partitions=5 \
        --port=5432 bench_part
```

- Sur **instance2**, utiliser pgbench pour créer la table `pgbench_account`
- avec un partitionnement de type *range* et trois partitions,
- mais sans insérer de données.

```
pgbench --initialize \
        --init-steps=dtp \
        --partition-method=range \
        --partitions=3 \
        --port=5433 bench_part
```

- Sur **instance1**, autoriser l'utilisateur de réplication à accéder aux tables.
- Créer une publication pour toutes les tables de la base.

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli ;
CREATE PUBLICATION pub_bench FOR ALL TABLES;
```

- Sur **instance2**, créer la souscription associée. Que constatez-vous ?

```
CREATE SUBSCRIPTION sub_bench
  CONNECTION 'host=127.0.0.1 port=5432 user=logrepli dbname=bench_part'
  PUBLICATION pub_bench;
```

La commande échoue avec le message suivant :

```
ERROR:  relation "public.pgbench_accounts_5" does not exist
```

L'erreur fait référence à une partition qui n'existe pas sur la souscription. C'est normal puisque le schéma de partitionnement est différent. Un autre cas de figure peut se présenter : la partition existe, mais les lignes ne correspondent pas aux contraintes de partitionnement. Dans ce cas la souscription sera créée, mais des erreurs seront présentes dans les traces de PostgreSQL.

- Sur **instance1**, supprimer la publication et la recréer avec l'option `publish_via_partition_root`.

L'option de publication `publish_via_partition_root` permet de répondre à ce problème en publiant les modifications avec le nom de la partition mère.

```
DROP PUBLICATION pub_bench;
CREATE PUBLICATION pub_bench
  FOR ALL TABLES
  WITH ( publish_via_partition_root = true );
```

- Sur **instance2**, recréer la souscription.

```
CREATE SUBSCRIPTION sub_bench
  CONNECTION 'host=127.0.0.1 port=5432 user=logrepli dbname=bench_part'
  PUBLICATION pub_bench;
```

- Sur **instance1** et **instance2**, compter les lignes dans chaque partition de `pgbench_accounts`. Qu'observez-vous ?

Sur **instance1**, la répartition des lignes dans `pgbench_account` est la suivante :

```
bench_part=# SELECT tableoid::regclass, count(*)
bench_part-# FROM pgbench_accounts
bench_part-# GROUP BY ROLLUP (1) ORDER BY 1;

      tableoid      | count
--------------------+-------
 pgbench_accounts_1 | 19851
 pgbench_accounts_2 | 20223
```

```
 pgbench_accounts_3 | 19969
 pgbench_accounts_4 | 19952
 pgbench_accounts_5 | 20005
 ¤                  | 100000
(6 rows)
```

Sur **instance2**, la répartition des lignes est la suivante :

```
bench_part=# SELECT tableoid::regclass, count(*)
bench_part-# FROM pgbench_accounts
bench_part-# GROUP BY ROLLUP (1) ORDER BY 1;

      tableoid      | count
--------------------+-------
 pgbench_accounts_1 | 33334
 pgbench_accounts_2 | 33334
 pgbench_accounts_3 | 33332
 ¤                  | 100000
(4 rows)
```

On constate que toutes les lignes sont répliquées et qu'elles sont ventilées différemment sur les deux serveurs.

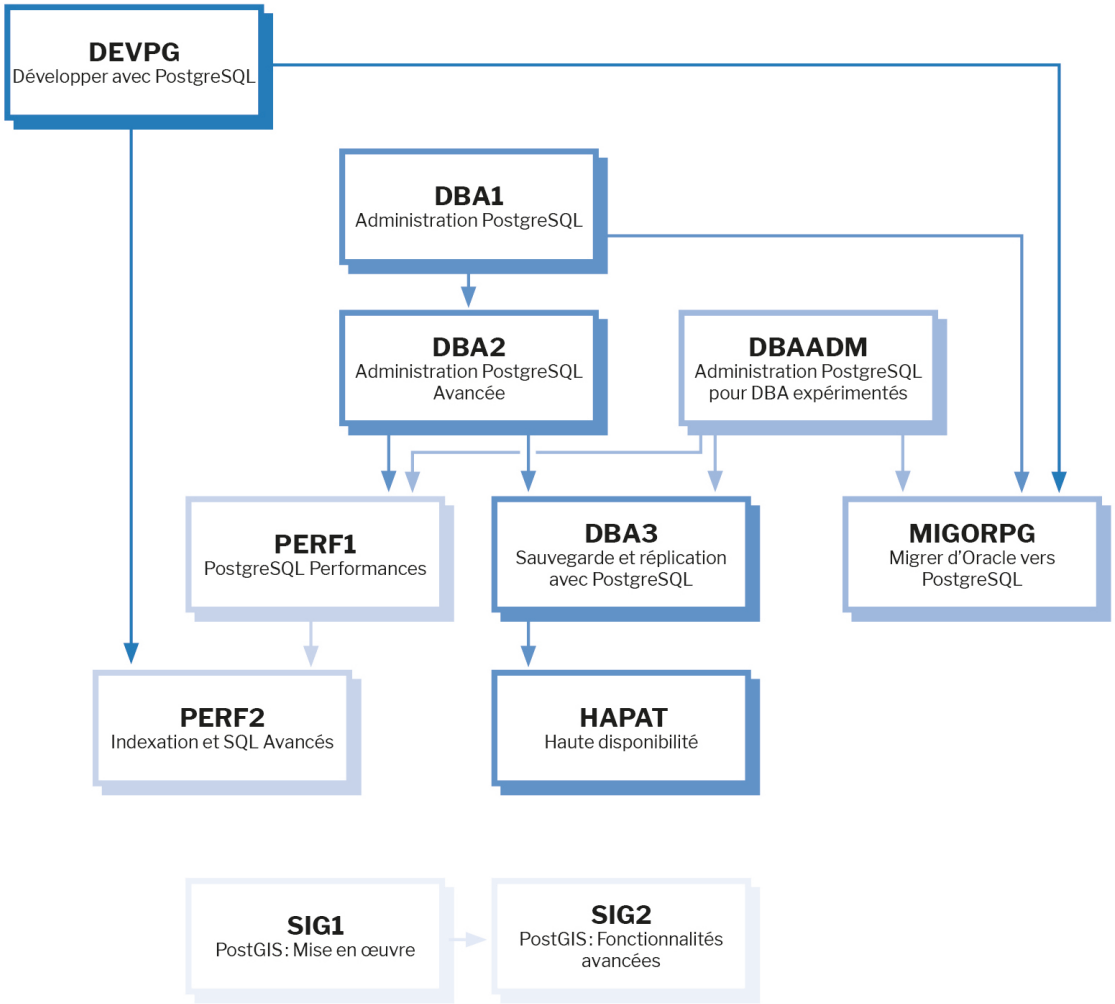
Ce paramétrage peut également être utilisé pour répliquer depuis une table partitionnée vers une table classique.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.