Formation DBA3

PostgreSQL Réplication

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Réplication

Formation DBA3

TITRE : PostgreSQL Réplication
SOUS-TITRE : Formation DBA3

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-02-9

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, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale: Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note: Ceci est un résumé de la licence.

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 ARCHITECTURES DE HAUTE-DISPONIBILITÉ

FIGURE 1: POSTGRESQL

2.1 PRÉAMBULE

- Attention au vocabulaire !
- Identifier le besoin
- Keep It Simple...

La réplication est le processus de partage d'informations permettant de garantir la sécurité et la disponibilité des données entre plusieurs serveurs et plusieurs applications. Chaque SGBD dispose de différentes solutions pour cela et introduit sa propre terminologie. Les expressions telles que "`Cluster`" , "`Actif/Passif`" ou "`Primaire/Secondaire`" peuvent avoir un sens différent selon le SGBD choisi. Dès lors, il devient difficile de comparer et de savoir ce que désignent réellement ces termes. C'est pourquoi nous débuterons ce module par un rappel théorique et conceptuel. Nous nous attacherons ensuite à citer les outils de réplication, internes et externes.

2.1.1 AU MENU

- Rappels théoriques
- Réplication interne
  - réplication physique
  - réplication logique
- 5 logiciels externes de réplication
- Alternatives

Dans cette présentation, nous reviendrons rapidement sur la classification des solutions de réplication, qui sont souvent utilisés dans un but de haute disponibilité, mais pas uniquement.

PostgreSQL dispose d'une réplication physique basée sur le rejeu des journaux de transactions par un serveur dit « en Standby ». Nous présenterons ainsi les techniques dites de « Warm Standby » et de « Hot Standby ».

Il dispose aussi depuis la version 10 d'une réplication logique. Elle sera aussi présentée.

Nous détaillerons ensuite les projets de réplication autour de PostgreSQL les plus en vue actuellement.

2.1.2 OBJECTIFS

- Connaître les principaux outils de réplication
- Identifier les différences entre les solutions proposées
- Choisir le système le mieux adapté à votre besoin

La communauté PostgreSQL propose plusieurs réponses aux problématiques de réplication. Le but de cette présentation est de vous apporter les connaissances nécessaires pour comparer chaque solution et comprendre les différences fondamentales qui les séparent.

À l'issue de cette présentation, vous serez capable de choisir le système de réplication qui correspond le mieux à vos besoins et aux contraintes de votre environnement de production.

2.2 RAPPELS THÉORIQUES

- Cluster
- Réplication
  - synchrone / asynchrone
  - symétrique / asymétrique
  - diffusion des modifications

Le domaine de la haute-disponibilité est couvert par un bon nombre de termes qu 'il est préférable de définir avant de continuer.

2.2.1 CLUSTER

- Dans la terminologie PostgreSQL
  - groupe autonome de bases de données
  - i.e. une instance
- Dans la terminologie haute-disponibilité et/ou réplication
  - groupe de serveurs

Toute la documentation (anglophone) de PostgreSQL parle de cluster dans le contexte d'un serveur PostgreSQL seul. Dans ce contexte, le cluster est un groupe de bases de données, groupe étant la traduction directe de cluster.

Dans le domaine de la haute-disponibilité et de la réplication, un cluster désigne un groupe de serveurs. Par exemple, un groupe d'un serveur maître et de ses deux serveurs esclaves compose un cluster de réplication.

2.2.2 RÉPLICATION ASYNCHRONE ASYMÉTRIQUE

- Asymétrique
  - écritures sur un serveur maître unique
  - lectures sur le maître et/ou les esclaves
- Asynchrone
  - les écritures sur les esclaves sont différées
  - perte de données possible en cas de crash du maître
- Quelques exemples
  - streaming replication, Slony, Londiste, Bucardo

Dans la réplication asymétrique, seul le maître accepte des écritures, et les esclaves ne sont accessibles qu'en lecture.

Dans la réplication asynchrone, les écritures sont faites sur le maître et, avant qu'elles ne soient poussées vers l'esclave, le client a un retour lui indiquant que l'écriture s'est bien passée. La mise à jour des tables répliquées **est différée** (asynchrone). Elle est réalisée par un programmateur de tâches, possédant une horloge. Des points de synchronisation sont utilisés pour propager les changements.

L'inconvénient de ce système est que, si un crash intervient sur le maître après que le client ait eu la réponse du succès de l'écriture mais avant que les données ne soient poussées sur l'esclave, certaines données validées sur le maître ne seront pas disponibles sur l'esclave. Autrement dit, il existe une fenêtre, plus ou moins importante, de perte de données.

2.2.3 RÉPLICATION ASYNCHRONE SYMÉTRIQUE

- Symétrique
  - écritures sur les différents maîtres
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents maîtres
- Asynchrone
  - les écritures sur les esclaves sont différées
  - perte de données possible en cas de crash du maître

Dans la réplication symétrique, tous les serveurs sont accessibles aux utilisateurs, aussi bien en lecture qu'en écriture. La réplication asynchrone, comme indiquée précédemment, met en attente l'envoi des modifications sur les esclaves, donc il y a toujours un risque de perte de données si le maître tombe sans avoir eu le temps d'envoyer les données à au moins un esclave

Ce mode de réplication ne respecte généralement pas les propriétés `ACID` (atomicité, cohérence, isolation, durabilité) car si une copie échoue sur l'autre maître alors que la transaction a déjà été validée, on peut alors arriver dans une situation où les données sont incohérentes entre les serveurs.

Généralement, ce type de système doit proposer un gestionnaire de conflits, de préférence personnalisable.

2.2.4 RÉPLICATION SYNCHRONE ASYMÉTRIQUE

- Asymétrique
  - écritures sur un serveur maître unique
  - lectures sur le maître et/ou les esclaves
- Synchrone
  - les écritures sur les esclaves sont immédiates
  - le client sait si sa commande a réussi sur l'esclave

Dans la réplication asymétrique, seul le maître accepte des écritures, et les esclaves ne sont accessibles qu'en lecture.

Dans la réplication synchrone, le client envoie sa requête en écriture sur le maître, le maître l'écrit sur son disque, il envoie les données à l' esclave, attend que ce dernier l'écrive sur son disque. Si tout ce processus s' est bien passé, le client est averti que l'écriture a été réalisée avec succès. On utilise généralement un mécanisme dit de **Two Phase Commit** ou "Validation en deux phases", qui assure qu'une transaction est validée sur tous les nœuds dans la même transaction. Les propriétés `ACID` sont dans ce cas respectées.

Le gros avantage de ce système est qu'il n'y a pas de risque de perte de données quand le maître s'arrête brutalement et qu'on doive repartir sur l' esclave. L'inconvénient majeur est que cela ralentit fortement les écritures.

Ce type de réplication garantit que l'esclave a bien écrit la transaction dans ses journaux et qu'elle a été synchronisé sur disque (fsync). En revanche elle ne garantit pas que l'esclave a bien rejoué la transaction. Il peut se passer un laps de temps très court où une lecture sur l'esclave serait différente du maître (le temps que l'esclave rejoue la transaction).

2.2.5 RÉPLICATION SYNCHRONE SYMÉTRIQUE

- Symétrique
  - écritures sur les différents maîtres
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents maîtres
- Synchrone
  - les écritures sur les esclaves sont immédiates
  - le client sait si sa commande a réussi sur l'esclave
  - risque important de lenteur

Ce système est le plus intéressant en théorie. L'utilisateur peut se connecter à n'importe quel serveur pour des lectures et des écritures. Il n'y a pas de risques de perte de don-

nées vu que la commande ne réussit que si les données sont bien enregistrées sur tous les serveurs. Autrement dit, c'est le meilleur système de réplication et de répartition de charge.

Dans les inconvénients, il faut gérer les éventuels conflits qui peuvent survenir quand deux transactions concurrentes opèrent sur le même ensemble de lignes. On résout ces cas particuliers avec des algorithmes plus ou moins complexes. Il faut aussi accepter la perte de performance en écriture induite par le côté synchrone du système.

Postgres-X2 (anciennement appelé "Postgres-XC") n'est pas un choix pérenne, il s'agit d'un fork de PostgreSQL 9.3 ce qui signifie qu'à l'horizon 2018, sa base de code sera périmée. Par ailleurs, c'est un projet qui nécessite un investissement très important en terme de matériel et maintenance opérationnelle.

Pgpool semblait prometteur, mais certaines fonctions (notamment le load-balancing et la réplication) se révèlent souvent complexes à mettre en œuvre, difficile à stabiliser et limitées à des cas d'utilisation très spécifiques. Malgré son ancienneté il y a encore beaucoup de corrections de bugs à chaque mise à jour.

2.2.6 DIFFUSION DES MODIFICATIONS

- 4 types de récupération des informations de réplication
- Par requêtes
  - diffusion de la requête
- Par triggers
  - diffusion des données résultant de l'opération
- Par journaux, physique
  - diffusion des blocs disques modifiés
- Par journaux, logique
  - extraction et diffusion des données résultant de l'opération depuis les journaux

La récupération des données de réplication se fait de différentes façons suivant l'outil utilisé.

La diffusion de l'opération de mise à jour (donc **le SQL lui-même**) est très flexible et compatible avec toutes les versions. Cependant, cela pose la problématique des opérations dites non déterministes. L'insertion de la valeur `now()` exécutée sur différents serveurs fera que les données seront différentes, généralement très légèrement différentes, mais différentes malgré tout. L'outil pgPool, qui implémente cette méthode de réplication, est capable de récupérer l'appel à la fonction `now()` pour la remplacer par la date et l'heure. Il est capable de le faire car il connaît les différentes fonctions de date et heure proposées

en standard par PostgreSQL. Cependant, il ne connaît pas les fonctions utilisateurs qui pourraient faire de même. Il est donc préférable de renvoyer les données, plutôt que les requêtes.

Le renvoi du résultat peut se faire de deux façons : soit en récupérant les nouvelles données avec un trigger, soit en récupérant les nouvelles données dans les journaux de transactions.

Cette première solution est utilisée par un certain nombre d'outils externes de réplication, comme Slony, Londiste ou Bucardo. Les fonctions triggers étant écrites en C, cela assure de bonnes performances. Cependant, seules les modifications des données sont attrapables avec des triggers. Les modifications de la structure de la base ne le sont pas (l'ajout des triggers sur événement en 9.3 est une avancée intéressante pour permettre ce genre de fonctionnalités dans le futur). Autrement dit, l'ajout d'une table, l'ajout d' une colonne demande une administration plus poussée, non automatisables.

La deuxième solution (par journaux de transactions) est bien plus intéressante car les journaux contiennent toutes les modifications, données comme structures. De ce fait, une fois mise en place, elle ne demande pas une grosse administration.

Depuis PostgreSQL 9.4, un nouveau niveau `logical` a été ajouté dans le paramétrage des journaux de transactions (paramètre `wal_level`). Couplé à l'utilisation des slots de réplication (nouveau paramètre `max_replication_slots`), il permet le décodage logique des modifications de données correspondant aux blocs modifiés dans les journaux de transactions. L' objectif était de permettre la reconstitution d'un ordre SQL permettant d'obtenir le même résultat, ce qui permettrait la mise en place d'une réplication logique des résultats entièrement intégrée, donc sans triggers. Ceci est disponible depuis la version 10 de PostgreSQL.

2.3 RÉPLICATION INTERNE PHYSIQUE

- Réplication
  - asymétrique
  - asynchrone ou synchrone
- Esclaves
  - non disponibles (Warm Standby)
  - disponibles en lecture seule (Hot Standby, à partir de la 9.0)
  - cascade possible à partir de la 9.2
  - retard possible à partir de la 9.4

Ce mode de réplication est par défaut **asynchrone** et **asymétrique**. Le mode synchrone est disponible à partir de la version 9.1. Il est même possible de sélectionner le mode synchrone/asynchrone pour chaque esclave individuellement.

Il fonctionne par l'envoi des enregistrements des journaux de transactions, soit par envoi de fichiers complets (on parle de `Log Shipping`), soit par envoi de groupes d'enregistrements en flux (on parle là de `Streaming Replication`), puisqu'il s'agit d'une réplication par diffusion de journaux.

La différence entre `Warm Standby` et `Hot Standby` est très simple :

- un serveur esclave en Warm Standby est un serveur de secours sur lequel il n'est pas possible de se connecter ;
- un serveur esclave en Hot Standby accepte les connexions et permet l'exécution de requêtes en lecture seule.

À partir de la version 9.2, un esclave peut récupérer les informations de réplication d'un autre esclave. À partir de la 9.4, il peut appliquer les informations de réplication après un délai configurable.

2.3.1 LOG SHIPPING

- But
  - envoyer les journaux de transactions à un esclave
- Première solution disponible (dès 2006)
- Gros inconvénient
  - il est possible de perdre un journal de transactions entier

Le Log Shipping permet d'envoyer les journaux de transactions terminés sur un autre serveur. Ce dernier peut être un serveur esclave, en Warm Standby ou en Hot Standby, prêt à les rejouer.

Cependant, son gros inconvénient vient du fait qu'il faut attendre qu'un journal soit complètement écrit pour qu'il soit propagé vers l'esclave. Autrement dit, il est possible de perdre les transactions contenues dans le journal de transactions en cours en cas de failover. Sans même ce problème, cela veut aussi dire que le retard de l'esclave sur le maître peut être assez important, ce qui est gênant dans le cas d'un Hot Standby qu'on peut utiliser en lecture seule, par exemple dans le cadre d'une répartition de la charge de lecture.

2.3.2 STREAMING REPLICATION

- But
  - avoir un retard moins important sur le maître
- Rejouer **les enregistrements de transactions** du maître par **paquets**
  - paquets plus petits qu'un journal de transactions
- Solution idéalement couplée au `Hot Standby`

L'objectif du mécanisme de `Streaming Replication` est d'avoir un esclave qui accuse moins de retard. En effet, dans le cas du `Log Shipping`, il faut attendre qu'un journal soit complètement rempli avant qu'il ne soit envoyé à l'esclave. Un journal peut contenir plusieurs centaines de transactions, ce qui veut dire qu'en cas de crash du maître, si ce dernier n'a pas eu le temps de transférer le dernier journal, on peut avoir perdu plusieurs centaines de transactions. Le `Streaming Replication` diminue ce retard en envoyant les enregistrements des journaux de transactions par groupe bien inférieur à un journal complet. Il introduit aussi deux processus gérant le transfert entre le maître et l'esclave. Ainsi, en cas de perte du maître, la perte de données est très faible.

Les délais de réplication entre le maître et l'esclave sont très courts. Couplée au `Hot Standby`, cette technologie a rendu obsolète certains systèmes de réplication, utilisés bien souvent avec deux nœuds (un maître et un esclave) : une modification sur le maître sera en effet très rapidement visible sur un esclave, en lecture seule. Néanmoins, cette solution a ses propres inconvénients : réplication de l'instance complète, architecture forcément identique entre les serveurs du cluster, etc.

2.3.3 WARM STANDBY

- Intégré à PostgreSQL depuis 2006
- Serveur de secours en cas de panne
- L'esclave est identique au maitre
  - à quelques transactions près

Le Warm Standby existe depuis la version `8.2`, sortie le 5 décembre 2006. La robustesse de ce mécanisme simple est prouvée depuis longtemps.

Les journaux de transactions (généralement appelés `WAL`, pour *Write Ahead Log*) sont immédiatement envoyés au serveur secondaire après leur écriture. Le serveur secondaire est dans un mode spécial d'attente (le mode de restauration), et lorsqu'un journal de transactions est reçu, il est automatiquement appliqué à l'esclave.

Étant donné que le serveur distant n'applique que les journaux de transactions qu'il reçoit, il y a toujours un risque de pertes de données en cas de panne majeure sur le serveur primaire avant l'envoi du journal de transactions en cours. On peut cependant moduler le risque de trois façons:

- sauf en cas d'avarie très grave sur le serveur primaire, le journal de transactions courant peut être récupéré et appliqué sur le serveur secondaire ;
- on peut réduire la fenêtre temporelle de la réplication en modifiant la valeur de la clé de configuration `archive_timeout`... au delà du délai exprimé avec cette variable de configuration, le serveur change de journal de transactions, provoquant l'archivage du précédent. On peut par exemple envisager un `archive_timeout` à 30 secondes, et ainsi obtenir une réplication à 30 secondes près ;
- on peut utiliser l'outil `pg_receivewal` (apparu en 9.2, et nommé `pg_receivexlog` jusqu'en 9.6) pour créer à distance les journaux de transactions d'après le flux de réplication.

2.3.4 HOT STANDBY

- Évolution du `Warm Standby` apparue en 9.0
- Basé sur le même mécanisme
- Le serveur secondaire est ouvert aux connexions
  - et aux requêtes en lecture seule
- Différentes configurations
  - asynchrone ou synchrone
  - application immédiate ou retardée

Le `Hot Standby` est une évolution du `Warm Standby` en ce sens qu'il comble le gros défaut du `Warm Standby`. Un esclave en `Hot Standby` accepte les connexions des utilisateurs. Il permet aussi d'exécuter des requêtes en lecture seule.

2.3.5 EXEMPLE

Cet exemple montre un serveur maître en Streaming Replication vers un serveur esclave. Ce dernier est en plus en Hot Standby. De ce fait, les utilisateurs peuvent se connecter sur l'esclave pour les requêtes en lecture et sur le maître pour des lectures comme des écritures. Cela permet une forme de répartition de charge sur les lectures, la répartition étant gérée par le serveur d'applications ou par un outil spécialisé.

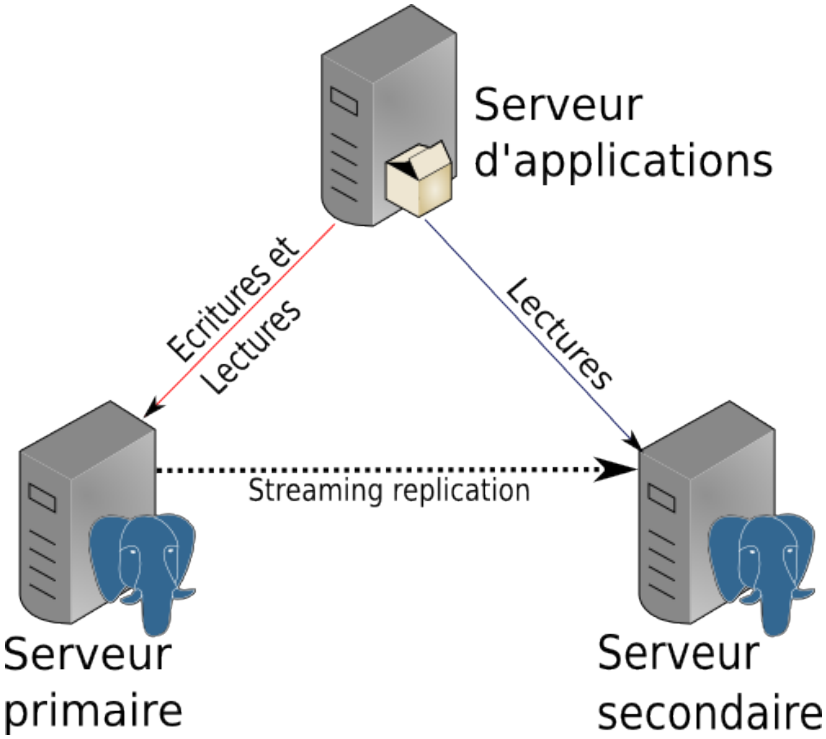
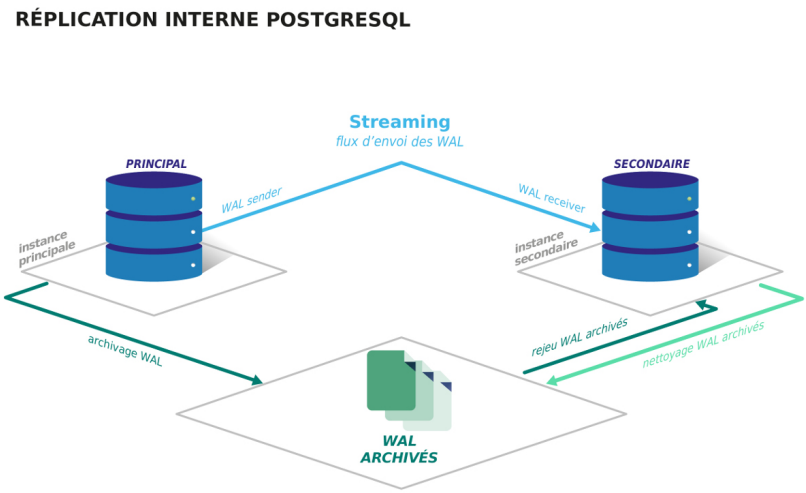
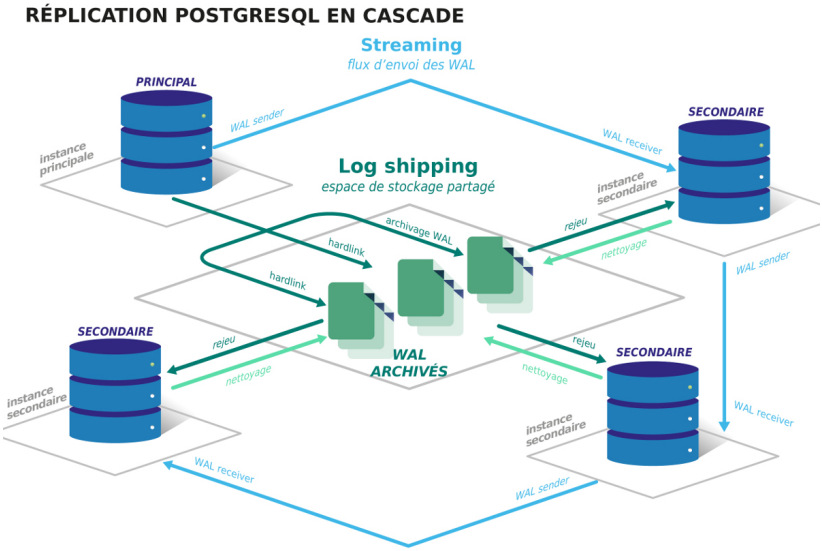


FIGURE 2: EXEMPLE D'ARCHITECTURE

2.3.6 RÉPLICATION INTERNE



2.3.7 RÉPLICATION EN CASCADE



2.4 RÉPLICATION INTERNE LOGIQUE

- Réplique les changements
  - sur une seule base de données
  - d'un ensemble de tables défini
- Uniquement INSERT / UPDATE / DELETE
  - pas les DDL, ni les TRUNCATE

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

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

Une publication peut être définie sur n'importe quel serveur primaire de réplication physique. Le nœud sur laquelle la publication est définie est nommé éditeur. Le nœud

où un abonnement a été défini est nommé abonné.

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

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

2.4.1 FONCTIONNEMENT

- Création d'une publication sur un serveur
- Souscription d'un autre serveur à cette publication

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

2.4.2 LIMITATIONS

- Non répliqués :
  - Schémas
  - Séquences
  - *Large objects*
- Pas de publication des tables parentes du partitionnement
- Ne convient pas comme *fail-over*

Le schéma de la base de données ainsi que les commandes DDL ne sont pas répliquées, ci-inclus l'ordre `TRUNCATE`. Le schéma initial peut être créé en utilisant par exemple `pg_dump --schema-only`. Il faudra dès lors répliquer manuellement les changements de structure.

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

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

17.12

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

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

Les *large objects* ne sont pas répliqués. Les séquences non plus, y compris celles utilisées des colonnes de type `serial`.

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

La réplication logique vise d'autres objectifs, tels la génération de rapports sur une instance séparée ou la mise à jour de version majeure de PostgreSQL avec une indisponibilité minimale.

2.5 RÉPLICATION EXTERNE

- Un très large choix !
- Quel solution choisir ?
  - pgPool
  - Slony / Bucardo / Londiste
  - Postgres-XC

On dénombre plus de 15 projets de réplication externe pour PostgreSQL. Jusqu' en 2010, PostgreSQL ne disposait pas d'un système de réplication évolué, ce qui explique en partie une telle profusion de solutions. Bien sûr, l'arrivée de la réplication interne physique (avec les technologies `Hot Standby` et `Streaming Replication`) change la donne mais ne remet pas en cause l'existence de tous ces projets. Par contre, l'arrivée de la réplication logique en version 10 risque de les mettre à mal.

Dans cette partie, nous ferons un zoom sur cinq logiciels de réplication :

- pgpool, réplication au niveau SQL
- Slony, réplication asynchrone et asymétrique
- Londiste, réplication asynchrone et asymétrique
- Bucardo, réplication asynchrone et symétrique
- Postgres-XC, réplication synchrone et symétrique

La liste exhaustive est trop longue pour que l'on puisse évoquer en détail chacune des solutions, surtout que certaines sont considérées maintenant comme obsolètes ou tout

du moins non maintenues. Voici les plus connues :

- PGCluster
- Mammoth Replicator
- Daffodil Replicator
- RubyRep
- pg_comparator

L'essentiel est donc de trouver le logiciel adapté à votre besoin !

Plus de détails à cette adresse[1] .

2.5.1 PGPOOL : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Synchrone / Symétrique
- Réplication des requêtes SQL
- Site web[2]

pgPool est un outil libre développé principalement par la société SRA OSS. Il propose un grand nombre de fonctionnalités tournant autour de la haute-disponibilité : pooler de connexions, répartition de charges et réplication. La réplication est en mode synchrone et symétrique. pgPool récupère la requête, la renvoie sur tous les serveurs et attend la réponse de chaque serveur avant de communiquer la réponse au client.

2.5.2 PGPOOL : FONCTIONNALITÉS

- Réplication basée sur les requêtes
- Capable de récupérer certaines fonctions non déterministes

pgPool récupère la requête à exécuter sur chaque serveur. Avant de l' envoyer, il l'analyse pour voir si elle fait appel à des fonctions non déterministes comme celles de récupération de la date et de l'heure : `now()` , `current_timestamp()`, etc. pgPool est capable de faire l'appel à `now()` dans une requête séparée, puis de remplacer l'appel à la fonction par le résultat de la requête séparée. Pour la requête

```
INSERT INTO t1 VALUES (2,now());
```

voici ce qu'il exécute sur le premier serveur :

[1] http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
[2] http://www.pgpool.net

```
LOG:  duration: 0.064 ms  statement: BEGIN
LOG:  duration: 0.143 ms  statement: SELECT now()
LOG:  duration: 0.243 ms  statement: INSERT INTO "t1" VALUES
        (2,"pg_catalog"."timestamptz"('2013-01-21 17:24:37.346359+01'::text))
LOG:  duration: 6.090 ms  statement: COMMIT
```

Il a bien remplacé l' appel à `now()` par le résultat de la précédente requête. Et voici ce qu'il exécute sur le deuxième serveur :

```
LOG:  duration: 0.037 ms  statement: BEGIN
LOG:  duration: 0.215 ms  statement: INSERT INTO "t1" VALUES
        (2,"pg_catalog"."timestamptz"('2013-01-21 17:24:37.346359+01'::text))
LOG:  duration: 161.343 ms  statement: COMMIT
```

Cependant, ce n'est fonctionnel que pour certaines fonctions que pgPool connaît, pas pour les autres (pas pour `random()` par exemple). De plus, si la fonction `now()` est la valeur par défaut de la colonne, pgPool ne sait pas le gérer. Il sera possible de voir une légère différence entre les différents serveurs :

```
-- sur le premier serveur
  2 | 2013-01-21 17:26:59.814946+01
-- sur le deuxième serveur
  2 | 2013-01-21 17:26:59.815214+01
```

Même si la différence est minime, il s'agissait d'un cas simple où les deux serveurs PostgreSQL sont sur la même machine. Avec une machine séparée, la différence peut être très importante, surtout si les horloges des deux machines ne sont pas synchronisées.

2.5.3 PGPOOL : TECHNIQUE

- pgPool est à l'origine un *pooler* de connexions
- Configuration propre
- Plus ou moins transparent pour les applications
- Réplication de toutes les requêtes
  - y compris le `DDL`

Au départ, pgPool n'était qu'un pooler de connexions. Il a ensuite évolué pour intégrer d'autres fonctionnalités, comme le répartiteur de charges, la réplication, le cache de requête, le partitionnement, un peu de pacemaker.

C'est un logiciel supplémentaire qui se fait passer pour un serveur PostgreSQL. Il est très simple à mettre en place, sa configuration est plutôt aisée et il est plus ou moins transpar-

ent pour les applications. Cependant, il faut vous attendre à devoir modifier vos applications suivant les fonctionnalités demandés.

Contrairement aux autres outils de réplication (notamment ceux par trigger), il réplique directement toutes les requêtes, y compris les DDL.

2.5.4 PGPOOL : LIMITES

- pgPool est un *SPOF*
  - SPOF => Single Point Of Failure
- Réplication basée sur la réplication des requêtes `SQL`
  - obligation de passer par pgpool
- Prise en main et fonctionnement complexe
- Authentification en mode réplication
  - pas de `md5`
  - pas de méthodes externes (LDAP, Radius, etc.)
- Effet « couteau suisse »

pgPool est le serveur sur lequel toutes les applications vont se connecter à PostgreSQL. S'il tombe, les bases de données ne sont plus accessibles. Il faudra donc veiller à ce qu'un autre service pgpool-II existe sur une autre machine et à mettre en place un système de bascule automatique. Cela est généralement fait avec des infrastructures redondantes basées sur `heartbeat` /`pacemaker`, `lvm`, etc.

De plus, il faut veiller à ce que les bases de données ne puissent être accédées que via pgPool. Si jamais un utilisateur peut se connecter sur une base sans passer par pgPool, il pourrait y exécuter des requêtes en écriture qui, de ce fait, ne seront exécutées que sur ce serveur. Cela provoquerait une désynchronisation des données entre les différents serveurs.

Comme pgPool agit en proxy, il est impossible d'utiliser une authentification `md5` ou des authentifications basées sur des méthodes externes comme LDAP, Radius, etc. De même, la gestion des certificats peut poser de gros soucis.

Pgpool semblait prometteur mais certaines fonctions (notamment le load-balancing et la réplication) se révèlent souvent complexes à mettre en œuvre, difficile à stabiliser et limitées à des cas d'utilisation très spécifiques. Malgré son ancienneté il y a encore beaucoup de corrections de bugs à chaque mise à jour. Sa maîtrise demande du temps et nécessite d' écrire les scripts de gestion de Haute Disponibilité.

Dans la communauté PostgreSQL, les critiques sont récurrentes à l'encontre de pgPool

: beaucoup lui reprochent de tout faire un peu. pgPool est en effet à la fois un pooler de connexions, un outil de réplication, un outil d'exécution de requêtes en parallèle, un outil de répartition de charge, un simili-pacemaker, etc. Ces critiques sont plutôt fondées. Ceux qui ne s'intéressent qu'au mode de pooling peuvent toujours opter pour l'excellent PgBouncer de Skype.

2.5.5 PGPOOL : UTILISATIONS

- Base de données de secours
- Répartition de charge en lecture

L'utilisation principale de pgPool actuellement réside en sa fonctionnalité de répartition de charge. Son mode réplication est très souvent déconseillé, de meilleures implémentations de la réplication existant ailleurs.

Vous trouverez plus d'informations dans cet article[3] .

2.5.6 SLONY : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Asymétrique
- Diffusion des résultats (triggers)
- Site web[4]

Slony est un très ancien projet libre de réplication pour PostgreSQL. C'était l'outil de choix avant l'arrivée de la réplication en interne dans PostgreSQL.

2.5.7 SLONY : FONCTIONNALITÉS

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

[3] http://www.dalibo.org/hs44_pgpoolii_la_replication_par_duplication_des_requetes
[4] http://slony.info/

Contrairement à la réplication interne de PostgreSQL qui réplique forcément tout (ce qui est un avantage et un inconvénient), Slony vous laisse choisir les tables que vous voulez répliquer. Cela a pour conséquence que, si vous ajoutez une table, il faudra en plus dire à Slony s'il est nécessaire ou non de la répliquer.

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

2.5.8 SLONY : TECHNIQUE

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

Slony est un système de réplication asynchrone/asymétrique, donc un seul maître et un ou plusieurs esclaves mis à jour à intervalle régulier. La récupération des données modifiées se fait par des triggers, qui stockent les modifications dans les tables systèmes de Slony avant leur transfert vers les esclaves. Un système de démon récupère les données pour les envoyer sur les esclaves 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 maître est souvent appelé un « provider » (il fournit les données aux esclaves) et les esclaves sont souvent des « subscribers » (ils s'abonnent au flux de réplication pour récupérer les données modifiées).

2.5.9 SLONY : POINTS FORTS

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

Slony dispose de nombreux points forts qui font défaut à la réplication interne de PostgreSQL.

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.

17.12

Le maître et les esclaves 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.

2.5.10 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.

2.5.11 SLONY : UTILISATIONS

- Clusters en cascade
- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc)

La réplication proposée par Slony est surtout intéressante pour les besoins complexes, comme mettre des serveurs en cascade (quoique PostgreSQL est capable de le faire depuis la version 9.2).

Pour avoir un esclave permettant la création de rapports, Slony peut se révéler plus intéressant car il est possible d'avoir des tables de travail en écriture sur l'esclave avec Slony. Il est aussi possible d'ajouter des index sur l'esclave qui ne seront pas présents sur le maître

(on évite donc la charge de maintenance des index par le maître, tout en permettant de bonnes performances pour la création des rapports).

Pour plus d'informations sur Slony, n'hésitez pas à lire un de nos articles disponibles sur notre site[5] . Le thème des réplications complexes a aussi été abordé lors d'un PostgreSQL Sessions[6] .

2.5.12 LONDISTE : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Asymétrique
- Réplication des résultats
- Site web[7]

Londiste est un projet libre, conçu et développé par la société Skype. Il s 'agit là-aussi de réplication asynchrone/asymétrique, par trigger, tout comme Slony.

2.5.13 LONDISTE : FONCTIONNALITÉS

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

Tout comme Slony, Londiste offre davantage de finesse concernant les tables à répliquer. De la même manière, il faut explicitement indiquer les tables à répliquer.

Les procédures de bascule sont tout aussi simples que celles de Slony. Il est ainsi possible de basculer un maître et son esclave autant de fois qu'on le souhaite, très rapidement, sans avoir à reconstruire quoi que ce soit.

2.5.14 LONDISTE : TECHNIQUE

- Réplication basée sur des triggers

[5] http://www.dalibo.org/hs44_slony_la_replication_des_donnees_par_trigger
[6] http://www.dalibo.org/replications_sophistiquees_avec_slony
[7] http://skytools.projects.pgfoundry.org/skytools-3.0/doc/londiste3.html

- Démons externes, écrits en Python
- Utilise un autre outil provenant des *Skytools* : PgQ
- 1 maître / N esclaves

Londiste est un outil qui ressemble beaucoup à Slony dans ces caractéristiques. La différence principale tient dans le fait qu'il ne gère pas de groupes de tables. C'est donc aussi de la réplication par triggers (attention, pas de triggers sur l'esclave qui empêchent les modifications sur les tables répliquées). Les triggers récupèrent les modifications et les stockent dans des tables systèmes. Un démon récupère les données dans les tables systèmes pour les envoyer et les écrire sur l'esclave. Pour une table, il n'y a qu'un maître et un ou plusieurs esclaves.

2.5.15 LONDISTE : POINTS FORTS

- PgQ est robuste, fiable et flexible
- Pas de groupes de réplication, mais des tables appartenant à différentes *queues*

On peut ainsi avoir des tables dans le serveur "maître" qui alimentent la *queue* principale à laquelle les différents "esclaves" auront souscrit, mais aussi d'autres *queues* qui vont alimenter certaines autres tables du "maître".

Cela rend donc possible la mise en place de réplications "croisées".

2.5.16 LONDISTE : LIMITES

- Technique de propagation des *DDL* basique
- Très peu de fonctionnalités.

Pour la réplication des requêtes DDL, c'est encore plus basique que Slony. Il faut soi-même envoyer les requêtes DDL sur les différents serveurs du cluster d'instances. Néanmoins depuis la version 3 de Londiste, il existe une commande permettant d'exécuter un script SQL sur un ou plusieurs nœuds.

Londiste est aussi plus simple à mettre en place et à maîtriser, tout simplement parce qu'il propose moins de fonctionnalités que Slony.

2.5.17 LONDISTE : UTILISATIONS

- Clusters en cascade
- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc)

L'utilisation de Londiste porte sur les mêmes cas que Slony. Le choix entre les deux se fera plutôt sur un avis personnel.

Pour plus d'informations sur Londiste, voir cet article[8] .

2.5.18 BUCARDO : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Symétrique ...
- ... ou Asynchrone / Asymétrique
- Réplication des résultats (dits *deltas*)
- Site web[9]

Ce projet libre a été créé par la société End Point Corporation pour le besoin d'un client particulier. Ils ont fini par décider de libérer le code qu 'ils continuent à maintenir.

C'est un des rares outils à proposer du multi-maître. Jusqu'à la version 5, seuls deux nœuds peuvent être en maître. À partir de la version 5, il sera possible d'avoir plus de deux serveurs maîtres. Il fonctionne aussi en utilisant des triggers mais sa mise en place est vraiment différente par rapport à Slony et Londiste.

2.5.19 BUCARDO : FONCTIONNALITÉS

- Réplication maître-maitre ou maître-esclave
- Répartition des écritures
- Failover manuel
- Plusieurs méthodes de résolution des conflits

Bucardo offre plusieurs types de réplication. « fullcopy » et « pushdelta » permettent de répliquer les données en mode maître-esclave, « swap » est le mode de synchronisation multi-maître.

[8] http://www.dalibo.org/hs44_londiste_la_replicaton_vue_par_skype
[9] http://bucardo.org/

17.12

Bucardo ne s'attèle qu'à la réplication des données, c'est à l' administrateur de réagir en cas de panne et de réaliser les opérations de bascule et de remise en réplication.

2.5.20 BUCARDO : TECHNIQUE

- Réplication basée sur des triggers
- Démons externes, écrits en Perl
- Maître / maître (1 seul couple)
  - ou maître / esclave(s)

En mode maître-esclave, le principe de fonctionnement de Bucardo est très proche de Slony : des jeux de réplication appelés « sync » utilisent des triggers, un service en Perl se charge alors de propager les modifications.

En mode maître-maître, le type de réplication « sync » indique au service en Perl de réaliser la réplication dans les deux sens avec la gestion des conflits.

2.5.21 BUCARDO : POINTS FORTS

- Basé sur un(des) set(s) de réplication et non sur un(des) schéma(s)
- Simplicité d'utilisation
- Résolution standard des conflits

Bucardo introduit les concepts suivants pour la résolution des conflits en réplication multi-maître :

- `source` : la base de données d'origine gagne toujours
- `target` : la base de destination gagne toujours
- `random` : l'une des deux bases est choisie au hasard comme étant la gagnante
- `latest` : la ligne modifiée le plus récemment gagne
- `abort` : la réplication est arrêtée
- `skip` : aucune décision ni action n'est prise

Il est également possible de créer son propre gestionnaire de résolution de conflit personnalisé.

2.5.22 BUCARDO : LIMITES

- Aucune technique de propagation des *DDL*
  - (en cours de développement)
- Limité à deux nœuds en mode multi-maîtres
- Le réseau doit être fiable
  - peu de retard, pas ou peu de coupures
- Sous Linux/Unix uniquement
- Un seul développeur sur le projet
- Manque de fiabilité de l'outil

Le projet est porté par pratiquement un seul développeur, Greg Sabino Mulane, développeur très connu (et apprécié) dans la communauté PostgreSQL. Cela explique un développement en dent de scie, même si la correction des bugs est généralement très rapide.

La propagation des DDL n'est pas prise en compte. Il faut donc, comme pour Slony, exécuter les DDL sur chaque serveur séparément.

2.5.23 BUCARDO : UTILISATIONS

- Cluster maître/maître simple
- Base de données de secours

La mise en place de Bucardo est intéressante pratiquement uniquement quand on veut mettre en place un cluster maître/maître. En dehors de cela, il est préférable de se baser sur des solutions comme Slony ou Bucardo.

2.5.24 POSTGRES-XC : CARTE D'IDENTITÉ

- Projet libre (Licence PostgreSQL)
- Réplication multi-maîtres synchrone
- Réplication des résultats
- Ancien site web[10]
- Projet renommé en postgres-x2[11]

[10] http://postgres-xc.sourceforge.net
[11] https://github.com/postgres-x2/postgres-x2

17.12

Postgres-XC est une solution de réplication multi-maîtres synchrones et de répartition des données entre plusieurs serveurs de données. Le projet a été initié en 2010 par NTT en libérant un projet de recherche interne. EnterpriseDB s'ajoute alors à NTT pour soutenir le développement de Postgres-XC. La version 1.0, basée sur PostgreSQL 9.1, est sortie en juin 2012. La version 1.1 est basée sur PostgreSQL 9.2, tandis que la version 1.2 est basée sur PostgreSQL 9.3. Une nouvelle version basée sur PostgreSQL 9.4 est prévue pour suivre l'arrivée à l'état stable de cette version majeure du projet.

En 2015, lors du passage du projet sur GitHub, le projet a été renommé en `postgres-x2`.

2.5.25 POSTGRES-XC : FONCTIONNALITÉS

- Réplication multi-maîtres synchrone
- Distribution des données entre serveurs
  - partitionnement horizontal
- Répartition de charge
- Haute-disponibilité

Le principal intérêt de Postgres-XC est sa capacité à gérer plusieurs serveurs, tous en maître. En ce sens, il rejoint Bucardo, sauf qu'il est capable de prendre en compte un plus grand nombre de serveurs.

Il dispose de requêtes SQL permettant de distribuer les données entre plusieurs nœuds, ce qui permet un partitionnement horizontal facilité.

2.5.26 POSTGRES-XC : TECHNIQUE

- *Global Transaction Manager*
  - gestion des XID et conflits
- *Coordinator*
  - Catalogue / Répartiteur
- *Datanode*
  - Stockage

L'architecture de Postgres-XC est divisé en trois éléments.

En premier lieu, le *Global Transaction Manager* (GTM) est le service de gestion global des transactions. Il gère les conflits et fournit les identifiants de transactions.

Ensuite, le *Coordinator* fournit le point d'entrée pour les sessions clientes. Il équilibre la charge entre les *datanodes*, agrège les données réparties au besoin, et communique avec les autres *coordinators* par l'intermédiaire du GTM, il conserve le catalogue du cluster Postgres-XC.

Enfin, un *datanode* est un serveur PostgreSQL modifié qui exécute les requêtes fournies par les *coordinators* et stocke la totalité ou une partie des données selon la configuration.

2.5.27 POSTGRES-XC : POINTS FORTS

- Installation relativement aisée et bien documentée
- Intégration au cœur de PostgreSQL
- Ajout d'ordres DDL spécifiques

L'intégration au sein même du code de PostgreSQL est garant de performances optimales. Cependant, c'est aussi à double tranchant, comme cela sera exposé plus tard.

Les fonctionnalités de réplication de PostgreSQL sont, par exemple, mises à profit pour rendre une plate-forme distribuée (partitionnement horizontal) hautement-disponible.

Les tests et benchmarks produits au moment de la sortie de la version 1.0 indique une bonne tenue en charge jusqu'à 10 nœuds. La limite théorique est placée à 20 nœuds par le projet.

Le choix de la réplication ou du partitionnement se fait au niveau des ordres SQL de type DDL, ainsi une seule API permet de gérer l'organisation des données.

2.5.28 POSTGRES-XC : LIMITES

- Version dérivée de PostgreSQL
  - basée sur la version 9.3
- Chaque ordre SQL doit être porté
- Fonctionnalités manquantes
  - triggers, SQL/MED, savepoints, SSI
  - contraintes globales, détection globale des deadlock

L'inconvénient majeur de créer une version dérivée de PostgreSQL (fork) concerne le suivi des modifications du code initial : en plus de fournir ses fonctionnalités propres, Postgres-XC doit intégrer les nouvelles fonctionnalités introduites dans PostgreSQL.

17.12

Toutes les fonctionnalités de PostgreSQL n'ont pas encore été portées dans Postgres-XC, comme le support des extensions ou encore les triggers. Il est donc indispensable de valider les fonctionnalités disponibles par la lecture des notes de version et des maquettes de test avant d'adopter la solution.

Il faut donc être conscient que PostgreSQL et Postgres-XC sont deux projets distincts, même s'ils sont très liés. Autrement dit, une application fonctionnant sur PostgreSQL ne fonctionnera pas forcément sur Postgres-XC.

2.5.29 POSTGRES-XC : UTILISATIONS

- Très grandes bases transactionnelles sous forte charge
- Bases spécialisées
- Peu de cas d'utilisation connus en production

Postgres-XC est un projet prometteur, la documentation est fournie et à jour, une roadmap est disponible. Il est surtout soutenu par deux acteurs incontournables de l'écosystème PostgreSQL.

Il s'agit quand même d'un projet jeune qu'il est intéressant de laisser mûrir.

2.5.30 SOLUTIONS OBSOLÈTES

- Postgres-R
- PGCluster
- replicator
- cybercluster

Comme beaucoup de logiciels, certains deviennent obsolètes pour diverses raisons. La maintenance d'un logiciel est une tâche qui représente un coup humain non négligeable.

Parmi les solutions de réplication externe pour PostgreSQL, certains projets ne sont plus développés ou maintenus faute de main d'œuvre, comme Postgres-R. D' autres ont perdu leur intérêt suite à l'arrivée de la réplication interne (Hot-Standby et Streaming Replication) en version 9.0.

2.6 RÉPLICATION BAS NIVEAU

- RAID
- DRBD
- SAN Mirroring
- À prendre évidemment en compte...

Cette présentation est destinée à détailler les solutions logicielles de réplication pour PostgreSQL, uniquement. On peut tout de même évoquer les solutions de réplication de "bas niveaux", voire matérielles.

De nombreuses techniques matérielles viennent en complément essentiel des technologies de réplication utilisées dans la haute disponibilité. Leur utilisation est généralement obligatoire, du `RAID` en passant par les `SAN` et autres techniques pour redonder l'alimentation, la mémoire, les processeurs, etc...

2.6.1 RAID

- Obligatoire
- Fiabilité d'un serveur.
- RAID 1 ou RAID 10
- Lectures plus rapides
  - dépend du nombre de disques impliqués

Un système RAID1 ou RAID10 permet d'écrire les mêmes données sur plusieurs disques en même temps. Si un disque meurt, il est possible d'utiliser l'autre disque pour continuer. C'est de la réplication bas-niveau. Le disque défectueux peut être remplacé sans interruption de service. Ce n'est pas une réplication entre serveur mais cela contribue à la haute-disponibilité du système.

Le système RAID10 est plus intéressant pour les fichiers de données alors qu' un système RAID1 est suffisant pour les journaux de transactions.

2.6.2 DRBD

- Simple / synchrone / Bien documenté
- Lent / Esclave inaccessible / Linux uniquement

17.12

DRBD est un outil capable de répliquer le contenu d'un périphérique blocs. En ce sens, ce n'est pas un outil spécialisé pour PostgreSQL contrairement aux autres outils vus dans ce module. Il peut très bien servir à répliquer des serveurs de fichiers ou de mails. Il réplique les données en temps réel et de façon transparente, pendant que les applications modifient leur fichiers sur un périphérique. Il peut fonctionner de façon synchrone ou asynchrone. Tout ça en fait donc un outil intéressant pour répliquer le répertoire des données de PostgreSQL.

Pour plus de détails, consulter cet article[12] .

DRBD est un système simple à mettre en place. Son gros avantage est la possibilité d' avoir une réplication synchrone, son inconvénient direct est sa lenteur et la non-disponibilité des esclaves.

2.6.3 SAN MIRRORING

- Comparable à DRBD
- Solution intégrée
- Manque de transparence

La plupart des constructeurs de baie de stockage propose des systèmes de réplication automatisé avec des mécanismes de failover / failback parfois sophistiqués. Ces solutions présentent généralement les mêmes caractéristiques que DRBD. Ces technologies ont en revanche le défaut d'être opaques et de nécessiter une main d'œuvre hautement qualifiée.

2.7 CONCLUSION

Points essentiels :

- bien définir son besoin
- identifier tous les *SPOF*
- superviser son *cluster*
- tester régulièrement les procédures de *Failover* (Loi de Murphy...)

---

[12] http://www.dalibo.org/hs45_drbd_la_replication_des_blocs_disques

2.7.1 BIBLIOGRAPHIE

- Doc officielle : Chapitre 25
- Hors série « Haute-disponibilité avec PostgreSQL » dans GNU/Linux France Magazine
- Article de blog « Hot Standby : Un exemple concret »
- « 25. Haute disponibilité, répartition de charge et réplication[13] ». PGDG, 2010
- « Haute-disponibilité avec PostgreSQL[14] ». Guillaume Lelarge, 2009
- « Hot Standby : Un exemple concret[15] ». Damien Clochard, 2010

Iconographie :

- La photo initiale[16] est sous licence CC-BY.

2.7.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[13] http://docs.postgresql.fr/9.0/high-availability.html
[14] http://www.dalibo.org/haute_disponibilite_avec_postgresql
[15] http://blog.taadeem.net/index.php?post/2010/07/19/Hot-Standby-:-un-exemple-concret
[16] http://www.flickr.com/photos/epsos/3574411866/

3 HOT STANDBY : INSTALLATION ET PARAMÉTRAGE

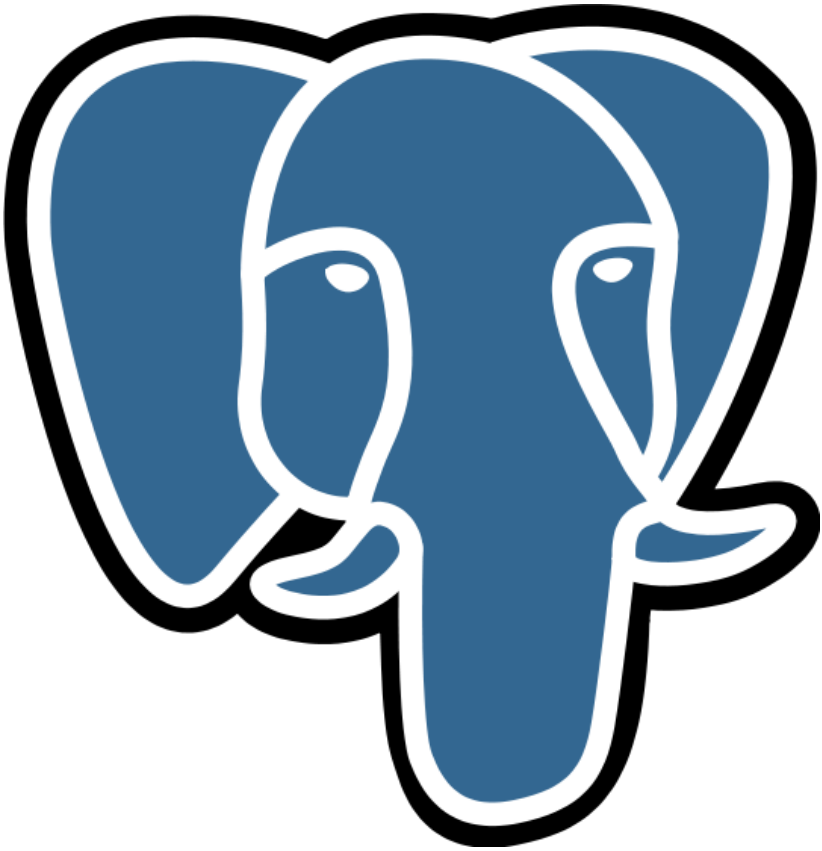


FIGURE 3: POSTGRESQL

3.1 INTRODUCTION

- Différentes solutions de réplication
  - par requêtes, par trigger, par journaux de transactions
- Jusqu'à la 9.0
  - installation d'un outil supplémentaire
- Mais à partir de la 9.0

  - réplication interne

PostgreSQL dispose de nombreuses solutions de réplication. Cela peut être de la réplication par requêtes comme le propose pgPool-II ou de la réplication par trigger (solution offerte par Slony, Londiste, Bucardo et quelques autres). Cependant, pour offrir de la réplication par journaux de transactions, la solution ne peut pas être externe. Cela demande de modifier les journaux de transactions pour ajouter les informations nécessaires à l'entretien d'un esclave, disponible en lecture seule ou non.

Le 29 mai 2008, Tom Lane envoie un mail sur la liste de discussion pgsql-hackers[17] . Ce message indique que la Core-Team de PostgreSQL a décidé du besoin d'ajouter un système de réplication simple mais fiable dans PostgreSQL. Cette réplication se baserait sur les journaux de transactions, qui semblent être la base la plus appropriée dans ce contexte. Le but était d'avoir ça pour la version 8.4. Le mail indique que les esclaves en lecture seule sont plus complexes à implémenter et pourraient n'être disponibles que pour la version 8.5.

En fait, la réplication a bien été intégrée à PostgreSQL. Elle se base bien sur les enregistrements des journaux de transactions. Cependant, elle n'a été disponible qu'en version 9.0 (ex 8.5). Les esclaves en lecture seule sont arrivés en même temps. La réplication synchrone, dont il n'est pas fait mention dans le mail de Tom Lane, arrive avec la version 9.1 de PostgreSQL. La réplication en cascade apparaît en 9.2. Les développeurs continuent évidemment à travailler sur la réplication, notamment sur un nouveau mode de réplication : la réplication logique.

3.1.1 AU MENU

- Mise en place du maître
- Mise en place d'un esclave
  - *Warm Standby*
  - *Hot Standby*
- Mise en place de la *Streaming Replication*
  - asynchrone
  - synchrone
  - en cascade

Ce module a pour but de présenter la mise en place d'une solution de réplication utilisant PostgreSQL. Nous allons commencer par le maître, avec une configuration standard. Ensuite nous aborderons les esclaves avec la possibilité de créer un *Warm Standby* et un *Hot*

[17] http://www.postgresql.org/message-id/26529.1212070375@sss.pgh.pa.us

Standby. Les deux possibilités seront expliquées intégralement, avec aussi la supervision et la gestion des conflits. Enfin, nous verrons la mise en place de la réplication en flux (*Streaming Replication*), tout d'abord en asynchrone, puis en synchrone.

3.2 CONCEPTS / PRINCIPES

- Les journaux de transactions contiennent toutes les modifications
  - utilisation du contenu des journaux
- L'esclave doit posséder une image des fichiers à un instant t
- La réplication modifiera les fichiers
  - d'après le contenu des journaux suivants

Chaque transaction, implicite ou explicite, réalisant des modifications sur la structure ou les données d'une base est tracée dans les journaux de transactions. Ces derniers contiennent des informations d'assez bas niveau, comme les blocs modifiés sur un fichier suite, par exemple, à un `UPDATE`. La requête elle-même n'apparaît jamais. Les journaux de transactions sont valables pour toutes les bases de données de l'instance.

Les journaux de transactions sont déjà utilisés en cas de crash du serveur. Lors du redémarrage, PostgreSQL rejoue les transactions qui n'auraient pas été synchronisées sur les fichiers de données.

Comme toutes les modifications sont disponibles dans les journaux de transactions et que PostgreSQL sait rejouer les transactions à partir des journaux, il suffit d'archiver les journaux sur une certaine période de temps pour pouvoir les rejouer.

3.2.1 VERSIONS

- 8.2 : Réplication par journaux (*Log Shipping*), *Warm Standby*
- 9.0 : Réplication en flux (*Streaming Replication*), *Hot Standby*
- 9.1 : Réplication synchrone
- 9.2 : Réplication en cascade
- 9.3 : Changement de maître en *streaming*
- 9.4 : Slots de réplication, délai de réplication, décodage logique
- 9.5 : `pg_rewind`, archivage depuis un esclave
- 9.6 : Rejeu synchrone
- 10 : Réplication synchrone sur base d'un quorum, slots de réplication temporaires

La version 8.0 contient déjà tout le code qui permet, après un crash du serveur, de pouvoir relire les journaux pour rendre cohérent les fichiers de données. La version 8.2 ajoute à cela la possibilité d'envoyer les journaux sur un esclave qui passe son temps à les rejouer journal après journal (*Log Shipping*). Cette version dispose déjà de la technologie *Warm Standby*. La version 8.3 ajoute la possibilité de supprimer les journaux archivés de façon sûre.

En 9.0, l'esclave est disponible pour des requêtes en lecture seule. C'est la technologie appelée *Hot Standby*. Cependant, cette version dispose d'une autre amélioration. Elle permet d'avoir une réplication en continue (*Streaming Replication*) : les modifications ne sont plus transférées journaux par journaux, mais groupe de transactions par groupe de transactions. Cela diminue d'autant le retard entre le maître et les esclaves.

La version 9.1 améliore encore le système en proposant un paramètre permettant de spécifier des esclaves en mode synchrone, tous les autres étant en mode asynchrone.

La version 9.2 propose des niveaux supplémentaires de synchronisation (en mémoire sur le serveur standby) et surtout la possibilité de placer les serveurs standby en cascade.

La version 9.3 permet à un esclave connecté en *streaming* à un maître nouvellement promu de poursuivre la réplication, sans redémarrage ni resynchronisation.

La version 9.4 ajoute la possibilité pour le maître de connaître la position de ses esclaves à l'aides des slots de réplication et ainsi de savoir précisément les journaux de transactions qui leur sont encore nécessaire. Cette version ajoute également la possibilité de configurer un délai au rejeu des transactions répliquées, ce qui permet d'avoir un retard de réplication contrôlé sur un esclave. Enfin, le nouveau niveau de paramétrage pour les journaux de transactions (`wal_level = logical`) et la configuration des slots logiques de réplication permettent d'en extraire les changement de données, et apportent les premières briques pour une réplication logique intégrée dans PostgreSQL.

L'outil `pg_rewind` arrive officiellement dans la version 9.5 (c'était une contribution externe depuis la version 9.3). Il permet de faciliter la reconstruction d'un ancien serveur maître devenu esclave. La version 9.5 apporte également la possibilité d'archiver depuis un esclave.

La 9.6 permet le rejeu synchrone sur l'esclave. La réplication synchrone apparue en 9.1 permettait de garantir que l'esclave avait reçu les transactions mais rien ne garantissait qu'il avait rejoué les transactions.

La version 10 offre la possibilité d'appliquer arbitrairement une réplication synchrone à un sous-ensemble d'un groupe d'instances. Précédemment, la réplication synchrone s'appliquait par ordre de priorité. Il sera désormais possible de se baser sur un quorum (`synchronous_standby_names = [FIRST]|[ANY] num_sync (node1, node2,...)`).

17.12

Cette version permet également de créer des slots de réplication temporaires. Dans ce cas, le slot de réplication n'est valide que pendant la durée de vie de la connexion qui l'a créé. À la fin de la connexion, le slot temporaire est automatiquement supprimé.

Comme on le voit, la réplication pouvait être simpliste au départ mais elle dispose maintenant d'un certain nombre d'améliorations qui en font un système beaucoup plus complet (et complexe).

3.2.2 AVANTAGES

- Système de rejeu éprouvé
- Mise en place simple
- Pas d'arrêt ou de blocage des utilisateurs
- Réplique tout

Le gros avantage de la réplication par enregistrements de journaux de transactions est sa fiabilité : le système de rejeu qui a permis sa création est un système éprouvé et déjà bien débuggé. La mise en place du système complet est simple car son fonctionnement est facile à comprendre. Elle n'implique pas d'arrêt du système, ni de blocage des utilisateurs.

L'autre gros avantage est qu'il réplique tout : modification des données comme de la structure d'une base. C'est une fonctionnalité que tous les systèmes de réplication par trigger aimeraient avoir mais n'ont malheureusement pas.

3.2.3 INCONVÉNIENTS

- Réplication de l'instance complète
- Impossible de changer d'architecture
- Blocage complet de l'esclave en dehors des lectures
- Même version majeure de PostgreSQL pour tous les serveurs

De manière assez étonnante, l'avantage de tout répliquer est aussi un inconvénient quand il s'avère qu'il n'est pas possible de ne répliquer qu'une partie. Avec la réplication interne de PostgreSQL, il n'est pas possible de ne répliquer qu'une base ou que quelques tables. Il faut toujours tout répliquer. De même, il n'est pas possible d'avoir des objets supplémentaires sur l'esclave, comme des index ou des tables de travail ce qui serait pourtant bien pratique pour de la création de rapports ou pour stocker des résultats intermédiaires de calculs statistiques. L'esclave est vraiment complètement bloqué en dehors des opérations de lecture seule.

Comme la réplication se passe au niveau du contenu des fichiers et des journaux de transactions, cela sous-entend qu'il n'est pas possible d'avoir deux nœuds du système de réplication ayant une architecture différente. Par exemple, ils doivent être tous les deux 32 bits ou 64 bits, mais pas un mix. Ils doivent être tous les deux *big endian* ou *little endian*. Et ils doivent aussi avoir la même version majeure.

3.3 MISE EN PLACE DU MAÎTRE

- 3 étapes :
  - création du répertoire d'archivage
  - configuration de PostgreSQL
  - rechargement de la configuration

La mise en place du maître est assez simple. L'action principale se limite à mettre en place l'archivage des journaux de transactions. Bien sûr, quelques nouveautés sont apparues avec la 9.0 et la *Streaming Replication* mais cela ne change pas le fond : il faut avoir un répertoire d'archivage et configurer PostgreSQL. Une fois que tout ceci est fait, il faut recharger la configuration de PostgreSQL pour que ce dernier agisse en concordance.

3.3.1 MEP DU MAÎTRE (1/4)

- Choix du répertoire d'archivage
- Répertoire local
  - ne pas oublier les droits et le propriétaire
- Répertoire distant
  - montage NFS
  - protocoles SSH, Samba, `rsync`

Le choix du répertoire est très personnel. Le seul conseil à donner est de ne pas archiver sur le serveur maître, mais plutôt sur l'esclave ou sur un serveur neutre.

Par ailleurs, suivant votre système d'exploitation, il se peut que des normes d'arborescence existent déjà. Par exemple, sous Unix, le standard communément appliqué est le FHS (*Filesystem Hierarchy Standard*). Il stipule que les fichiers de base de données, étant des fichiers variables (variant beaucoup), devraient être stockés dans `/var`, comme le décrit le FHS[18] .

[18] https://fr.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

Le répertoire, une fois sélectionné, doit être créé et l'utilisateur Unix `postgres` doit pouvoir écrire dans ce répertoire. Cela peut nous donner les actions suivantes :

```
$ mkdir /var/postgres/archives
$ chown postgres:postgres /var/postgres/archives
$ chmod 700 /var/postgres/archives
```

Attention : si le répertoire se trouve sur un autre serveur, il faut que l'utilisateur Unix `postgres` puisse y envoyer le fichier. Si cela passe par les commandes `scp` ou `sftp`, cela sous-entend la création d'un système de clé publique/clé privée, sans mot de passe. Pour d'autres systèmes, comme FTP ou Samba, il faut là-aussi résoudre le problème de l'authentification auprès de ces systèmes. Une solution plus simple pourrait être un montage NFS.

Un dernier point : l'archivage, s'il est au moins temporairement local au serveur, devrait préférentiellement être effectué sur un système de fichiers distinct, et idéalement sur un système RAID ou un disque distinct de ceux de production. Il est surtout recommandé de ne pas placer le répertoire des archives dans un sous-répertoire de `$PGDATA` ou un tablespace, pour éviter de dupliquer inutilement ces fichiers lors du backup de base (voir plus loin).

3.3.2 MEP DU MAÎTRE (2/4)

- Configuration (`postgresql.conf`)
  - `wal_level = replica` (ou `logical`)
  - `archive_mode = on` (ou `always`)
  - `archive_command = '... une commande ...'`
  - `archive_timeout` (optionnel)
- Ne pas oublier de forcer l'écriture de l'archive sur disque

Tout le reste se fait au niveau de la configuration du fichier `postgresql.conf`.

Il faut tout d'abord s'assurer que PostgreSQL enregistre suffisamment d'informations pour que l'esclave puisse rejouer toutes les modifications survenant sur le maître. Par défaut, PostgreSQL dispose de certaines optimisations qui lui permettent d'éviter certaines écritures quand cela ne pose pas de problème pour l'intégrité des données en cas de crash. Par exemple, il est inutile de tracer toutes les opérations d'une transaction qui commence par vider une table, puis qui la remplit. En cas de crash, l'opération complète est annulée. Cependant, dans le cas de la réplication, il est nécessaire d'avoir les étapes intermédiaires et il est donc essentiel d'enregistrer ces informations supplémentaires dans les journaux. Pour permettre les deux (avoir d'excellentes performances si on n'utilise pas la réplication,

et avoir la possibilité d'utiliser la réplication en acceptant des performances un peu moindres), le paramètre `wal_level` a été ajouté. Comme son nom l'indique, il permet de préciser le niveau d'informations que l'on souhaite avoir dans les journaux : `minimal` (valeur par défaut jusque 9.6 comprise), `replica` (lorsque le besoin concerne un archivage ou de la réplication, valeur par défaut à partir de la version 10), et enfin `logical` (permet en plus l'utilisation du décodage logique). Il est à noter que ce paramètre n'est disponible que depuis la version 9.0. Auparavant, la distinction était faite seulement avec le paramètre `archive_mode`.

Avant la version 9.6, il existait deux niveaux intermédiaires pour le paramètre `wal_level` : `archive` et `hot_standby`. Le premier permettait seulement l'archivage, le second permettait en plus d'avoir un serveur secondaire en lecture seule. Ces deux valeurs ont été fusionnées en `replica` avec la version 9.6. Les anciennes valeurs sont toujours acceptées, et remplacées silencieusement par la nouvelle valeur. La valeur `logical` apparaît en 9.4. La valeur `replica` devient la valeur par défaut en version 10.

Le paramètre `archive_mode` permet de passer en mode d'archivage. Il doit être à `on` dès qu'il s'agit d'utiliser la réplication ou plus simplement l'archivage des journaux de transactions. Il peut aussi prendre la valeur `always` pour permettre un archivage à partir d'un serveur secondaire, et ce à partir de la version 9.5.

Lorsque PostgreSQL a besoin d'archiver un journal de transactions, il exécute une commande système qui permet de faire cet archivage. Cela évite à PostgreSQL d'intégrer un grand nombre de protocoles de copie. C'est à l'administrateur de configurer l'outil adéquat. Cela peut être aussi simple que l'utilisation de la commande `cp` ou plus complexe par le développement et l'utilisation d'un script. PostgreSQL remplacera deux caractères jokers :

- `%p` par le chemin complet vers le journal de transactions à archiver ;
- `%f` par le nom du journal une fois archivé.

La configuration de ce paramètre pourrait ressembler à ceci :

```
archive_command = 'cp %p /mnt/nfs1/archivages/%f'
```

Une copie du fichier ne suffit pas. Par exemple, dans le cas de la commande `cp`, le nouveau fichier n'est pas immédiatement écrit sur disque. La copie est effectuée dans le cache disque du système d'exploitation. En cas de crash rapidement après la copie, il est tout à fait possible de perdre l'archive. Il est donc essentiel d'ajouter une étape de synchronisation du cache sur disque.

Si le code retour de cette commande est 0, PostgreSQL suppose que l'opération s'est bien passée, le journal pourra être recyclé le moment venu. Cependant, si la commande renvoie un autre code retour, le fichier est conservé et PostgreSQL essaiera autant de fois

que nécessaire de l'archiver. Tant qu'un fichier ne peut être archivé, les journaux suivants ne seront pas archivés non plus. Cela peut engendrer un problème de place sur le disque. C'est un point important à connaître pour la supervision du système.

Comme dit précédemment, l'archivage ne se fait qu'à partir du moment où PostgreSQL a terminé de travailler sur un journal de transactions. Que le remplissage de ce dernier se fasse en 10 secondes ou en 3 heures ne change rien à ce comportement. Il est parfois intéressant de s'assurer que le journal entamé puisse être archivé malgré tout. Il est donc possible de configurer un délai maximal avant archivage d'un journal de transactions. Cela se fait par l'intermédiaire du paramètre `archive_timeout`. Attention, si le fichier n'est pas rempli lorsqu'il est archivé, il occupe malgré tout 16 Mo. Cela peut générer une occupation très importante de la place disponible sur le serveur d'archivage.

3.3.3 MEP DU MAÎTRE (3/4)

- Configuration (`postgresql.conf`) - suite
  - `wal_keep_segments`
  - `vacuum_defer_cleanup_age`
  - `max_replication_slots` (>= 9.4)

Le paramètre `wal_keep_segments` indique le nombre de journaux de transactions à conserver sur le maître au cas où un serveur en Standby a besoin de les récupérer pour la réplication en flux. Le problème se pose généralement quand il y a une brusque activité très importante qui provoque un retard très important sur l'esclave. Évidemment, plus des journaux sont conservés, plus la taille occupée sur disque est importante.

Le paramètre `vacuum_defer_cleanup_age` précise le nombre de transactions avant qu'un VACUUM ou une mise à jour utilisant HOT puisse nettoyer les lignes mortes. La valeur par défaut est de 0. L'augmenter permet d'éviter les conflits sur l'esclave avec des requêtes en lecture seule plutôt longue.

À partir de PostgreSQL 9.4, il est possible de spécifier une certain nombre de slots de réplication grâce au paramètre `max_replication_slots`. La version 10 autorise par défaut 10 slots. Les esclaves pourront ensuite s'inscrire auprès du maître (« consommant » ainsi un slot de réplication), ce qui permettra au maître de retenir précisément le niveau de rejeu des transactions de chaque esclave inscrit ainsi. En conséquence, le maître sait précisément quels sont les journaux de transactions nécessaires par les esclaves, et les conservera automatiquement. Dans cette configuration, le paramètre `wal_keep_segments` n'est donc plus utile. Attention, les slots de réplication alloués à un esclave ne sont pas supprimés si celle-ci devient inactive, ce qui fait que les journaux de transactions seront

tous conservés sur le maître à partir de l'arrêt de l'esclave. Cela peut poser des problèmes d'espace disque si l'arrêt doit être très long. Dans ce cas, il est possible de libérer le slot à partir du maître en utilisant la fonction `pg_drop_replication_slot()`. Cela signifie naturellement que les fichiers journaux ne seront plus conservés pour le rattrapage du retard de cet esclave, et donc qu'il faudra probablement le reconstruire lorsqu'il sera de nouveau disponible.

3.3.4 MEP DU MAÎTRE (4/4)

- Rechargement de la configuration
  - par redémarrage de PostgreSQL (si `wal_level` ou `archive_mode` changés)
  - par envoi d'un signal à PostgreSQL

Pour que PostgreSQL prenne en compte sa nouvelle configuration, il faut lui demander de le faire. En cas de modification des paramètres `wal_level` ou `archive_mode`, il est nécessaire de redémarrer PostgreSQL. Dans tous les autres cas, il suffit simplement d'en demander la relecture avec l'action `reload` du script de démarrage ou de l'outil `pg_ctl`.

3.3.5 TESTS DU PRIMAIRE

- Seul test possible : l'archivage
- Générer de l'activité
- `pg_switch_wal`
- Vérifier le bon archivage
  - en regardant le contenu du répertoire d'archivage
  - et/ou en consultant `pg_stat_archiver`
- Sans archivage, vérifiez les logs

Après la mise en place du maître, il faut s'assurer que l'archivage est fonctionnel. Pour cela, l'archivage d'un journal doit avoir lieu. Soit on attend que des écritures surviennent, soit on les fait. Il n'est pas nécessaire de faire beaucoup d'écritures. Dès que quelques écritures ont eu lieu, on peut demander à PostgreSQL de passer au journal de transactions suivant grâce à l'exécution de la fonction `pg_switch_wal()` (nommée `pg_switch_xlog()` avant la version 10), ainsi :

```
SELECT pg_switch_wal();
```

Ceci doit déclencher l'arrêt de l'écriture du journal de transactions courant, sa demande d'archivage et l'utilisation d'un nouveau journal de transactions pour les écritures.

Il est ensuite possible de vérifier que le journal est bien archivé en consultant la vue statistique `pg_stat_archiver` ou en regardant le contenu du répertoire d'archivage.

En cas de problème, il est nécessaire de le corriger avant de procéder à la mise en place de l'esclave. La correction dépend évidemment du problème rencontré.

3.4 MISE EN PLACE D'UN WARM STANDBY

- Identique à la restauration d'une sauvegarde PITR
- Jusqu'en 8.4
  - configuration spécifique du `restore_command`
  - pas de `cp`, `scp`, `lftp`, `rsync`, etc.
  - à remplacer par un outil externe spécifique
- Depuis 9.0
  - fonctionnalité intégrée au moteur

La mise en place d'un serveur *Warm Standby* commence exactement comme une sauvegarde PITR suivie d'une restauration. Cependant, la commande utilisée dans le paramètre `restore_command` ne peut pas être un `cp`, `scp` ou `rsync` seul. En effet, avec ce type de commande, si le journal n'existe pas la commande renverra un code d'erreur et PostgreSQL basculera en mode autonome (lecture/écriture). Or, nous voulons **attendre** l'arrivée du prochain fichier pour le restaurer, puis attendre le suivant et ainsi de suite.

Jusqu'à la version 8.4 de PostgreSQL, nous ne pouvions donc pas utiliser un simple `cp` ou `scp`. Nous avions besoin d'un outil spécifique pour attendre l'arrivée du fichier avant de le fournir à PostgreSQL.

Un des principaux outils était `pg_standby`. Cet outil a fait son apparition dans les modules contrib à partir de la version 8.3 de PostgreSQL. Cependant, son code est très généraliste, ce qui fait qu'il est aussi utilisable avec une version 8.2.

Lorsque `pg_standby` est exécuté, il attend l'apparition du fichier dont le nom est donné sur la ligne de commande. Il copiera ce fichier dès son apparition, puis il rendra la main à PostgreSQL avec le code retour 0.

Il est possible de lui demander de détecter l'apparition d'un second fichier, appelé « fichier trigger », qui provoquera alors le retour d'un code d'erreur à PostgreSQL, permettant ainsi la bascule du serveur esclave en un serveur autonome.

Le délai entre chaque test de présence des fichiers est modifiable avec l' argument en ligne de commande `-s`. Il est même possible d'indiquer un délai maximum avec abandon de la récupération du journal de transactions grâce à l'argument `-w`.

Il existe un grand nombre d'autres arguments à cet outil, le plus simple est de consulter sa page de référence dans la documentation[19] . Se référer à la page de la kb[20] pour un exemple de mise en oeuvre.

Depuis la version 9.0 de PostgreSQL, un mécanisme d'attente du prochain journal de transactions a été intégré directement à PostgreSQL et peut être aisément configuré.

3.4.1 AVANTAGES

- Un deuxième serveur prêt à prendre la main

Le gros avantage du serveur en *Warm Standby* est qu'il s'agit d'un serveur prêt à être basculé en lecture/écriture. La bascule se fait simplement et rapidement.

3.4.2 INCONVÉNIENTS

- Un deuxième serveur inutilisable
  - tant qu'on ne le sort pas de la restauration
- Et tous les inconvénients de la sauvegarde de fichiers

Par contre, il faut bien comprendre que ce serveur en *Warm Standby* est bloqué dans son travail de restauration. Il est impossible de s'y connecter, y compris pour y exécuter des requêtes en lecture seule.

De plus, comme il s'agit d'une réplication par les journaux de transactions, tous les inconvénients relatifs à la sauvegarde de fichiers sont valables : instance complète (pas de granularité plus fine), même architecture de machines, etc.

3.4.3 MODE STANDBY

- Disponible depuis la 9.0
  - **Mode recommandé**
- configuration
  - `restore_command = cp` ou `scp` ou `lftp` ou ...
  - `standby_mode = on`

[19] http://docs.postgresql.fr/current/pgstandby.html
[20] https://kb.dalibo.com/pg_standby

  - `trigger_file = '/chemin/vers/fichier'`

Depuis la version 9.0 de PostgreSQL, il est possible de maintenir un serveur dans son état de restauration des données depuis les archives, même en cas d'échec de la commande `restore_command`. Pour ce faire, le paramètre `standby_mode` a été ajouté au fichier `recovery.conf`. Si ce paramètre est activé, l'instance ne deviendra autonome qu'au signal de l'administrateur. Nous pouvons alors désormais utiliser de simples commandes telles que `cp` ou `scp` dans le paramètre `restore_command`. Tout échec de la commande provoquera trois essais consécutifs, suivi d'une pause d'une minute avant de ré-exécuter ce cycle et ce, jusqu'à ce que la commande réussisse ou que l'administrateur intervienne.

Le paramètre `trigger_file` permet d'indiquer à PostgreSQL de surveiller la présence d'un fichier. PostgreSQL reste en mode restauration si ce fichier est absent. Dès que ce fichier est créé, le serveur en réplication devient disponible en lecture/écriture. Par ailleurs, depuis PostgreSQL 9.1, une nouvelle option `promote` a été ajoutée à l'outil d'administration en ligne de commande `pg_ctl`, remplissant la même fonction. La configuration de `trigger_file` n'est donc nécessaire que pour la version 9.0.

3.4.4 MISE EN PLACE D'UN WARM STANDBY (1/3)

- Pré-requis : archivage des journaux de transactions
- Copie des données
- Soit manuel
- Soit avec un outil
  - `pg_basebackup` (>= 9.1)

La première action à réaliser ressemble beaucoup à ce que propose la sauvegarde en ligne des fichiers. Il s'agit de copier le répertoire des données de PostgreSQL ainsi que les tablespaces associés.

Il est possible de le faire manuellement, tout comme pour une sauvegarde PITR. Cependant, depuis la version 9.1, un outil permet de se charger des trois étapes essentielles :

- le `pg_start_backup()` ;
- la copie des fichiers ;
- le `pg_stop_backup()`.

Cet outil s'appelle `pg_basebackup`. Son utilisation est très simple, en voici un exemple :

```
$ pg_basebackup -h 10.0.0.1 -D /var/lib/pgsql/data -c fast -P -v
19016/19016 kB (100%), 1/1 tablespace
```

```
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_basebackup: base backup completed
```

Attention, cet outil utilise une connexion passant par le protocole de réplication. Il est donc nécessaire que soient configurés le paramètre `max_wal_senders` ainsi que le fichier `pg_hba.conf` (pour accepter la connexion en mode réplication). Voir la partie sur la mise en place de la *Streaming Replication* pour plus d'informations.

Il est préférable de mettre en place un slot de réplication sur le serveur primaire et de faire utiliser ce slot à `pg_basebackup` avec l'option `-S`. Cependant, cette option n'est disponible que depuis la version 9.6. En version 10, `pg_basebackup` utilisera automatique un slot de réplication temporaire si aucun slot ne lui est explicitement spécifié avec l'option `-S`.

Il est à noter que cet outil est capable de créer le fichier `recovery.conf` qu'on aborde dans la prochaine slide. Pour cela, il faut utiliser l'option `--write-recovery-conf`.

3.4.5 MISE EN PLACE D'UN WARM STANDBY (2/3)

- Configuration (`recovery.conf`)
  - automatisable avec `pg_basebackup`
- À partir de la 9.0
  - `restore_command` avec `rsync`
  - `standby_mode = on`

Une fois que l'esclave dispose des fichiers de données, il faut configurer le fichier `recovery.conf` qui indique à PostgreSQL comment gérer la restauration. Ce fichier doit se trouver dans le répertoire des données de PostgreSQL, y compris sous Debian et Ubuntu.

Jusqu'en 8.4

Un seul paramètre nous intéresse. Il s'agit du paramètre `restore_command`. Sur une restauration PITR, nous utiliserions une commande `cp` ou `scp`, mais comme nous l'avons dit plus haut, nous avons besoin d'un outil spécifique dans notre cas. Si on souhaite utiliser `pg_standby`, cela pourrait donner lieu à cette configuration :

```
restore_command = 'pg_standby -s 1 -t /tmp/STOP /var/postgres/archives %f %p %r'
```

Ainsi PostgreSQL ira chercher le prochain journal de transactions dans le répertoire `/var/postgres/archives`. S'il ne le trouve pas, il vérifiera la présence du fichier `/tmp/STOP`. Si ce dernier existe, `pg_standby` quitte avec un code d'erreur 1 indiquant à PostgreSQL que la restauration est terminée. Si ce fichier trigger n'existe pas, il attendra une seconde (option `-s 1`) avant de chercher de nouveau le journal de transactions.

17.12

À partir de la 9.0

Depuis la version 9.0 de PostgreSQL il est possible de maintenir un serveur dans son état de restauration des données depuis les archives, même en cas d'échec de la commande `restore_command`. Pour ce faire, le paramètre `standby_mode` a été ajouté au fichier `recovery.conf`. Si ce paramètre est positionné à `on`, l'instance ne deviendra autonome qu'au signal de l'administrateur. Nous pouvons alors désormais utiliser de simples commandes telles que `cp` ou `scp` dans le paramètre `restore_command`. Tout échec de la commande provoquera 3 essais consécutifs, suivi d'une pause d'une minute avant de réexécuter ce cycle et ce, jusqu'à ce que la commande réussisse ou que l'administrateur intervienne.

La configuration du fichier `recovery.conf` devient alors :

```
restore_command = 'rsync /mnt/nfs1/archivages/%f %p'
standby_mode = on
```

Il est possible d'utiliser d'autres outils de copie comme `cp`, `scp`, etc. Nous recommandons `rsync` car il est le seul, à notre connaissance, à faire des copies sécurisées (*ie*, si la commande est arrêtée avant la fin de la copie, on ne se retrouve pas avec une moitié de fichier).

À partir de 9.3

Il est possible de demander à `pg_basebackup` de créer un fichier `recovery.conf` minimal. Ceci est réalisé par l'ajout d'un argument :

```
pg_basebackup --write-recovery-conf
```

Cette option permet d'avoir un esclave démarrable directement, mais il est préférable de vérifier le contenu de ce fichier avant.

3.4.6 MISE EN PLACE D'UN WARM STANDBY (3/3)

- Démarrer PostgreSQL

Il ne reste plus qu'à démarrer l'esclave. Au démarrage, il constatera la présence du fichier `recovery.conf`, le lira et exécutera la commande indiquée au niveau du paramètre `restore_command` chaque fois qu'il a besoin d'un nouveau journal de transactions.

3.5 MISE EN PLACE D'UN HOT STANDBY

- Avec le *Warm Standby*, l'esclave n'est pas disponible
  - y compris pour des requêtes en lecture seule
- *Hot Standby* = esclave en lecture seule
- Activé par défaut en version 10

Le plus gros soucis du *Warm Standby* tient dans le fait que l'esclave n'est pas disponible, et ce même pour des requêtes en lecture seule. Or il est très intéressant de pouvoir déplacer une partie de l'activité sur ce deuxième serveur, pour exemple pour générer des rapports.

La solution arrive avec la version 9.0 de PostgreSQL. Il est possible de configurer l'esclave pour qu'il accepte les requêtes en lecture seule. Dans ce cas, il devient un *Hot Standby*.

La version 10 active ce comportement par défaut.

3.5.1 AVANTAGES

- Un deuxième serveur en lecture seule

L'avantage est évident. Le deuxième serveur est accessible en lecture seule, ce qui permet de vérifier dans le détail que la réplication se passe bien. Mais surtout, cela permet de décharger une partie de l'activité du premier serveur. La génération de rapports peut se faire sur ce serveur, permettant ainsi au maître de ne plus avoir cette charge et de se consacrer plutôt aux connexions de mise à jour de la base.

3.5.2 INCONVÉNIENTS

- Aucune modification possible
  - données comme structure
  - y compris tables temporaires
- Et tous les inconvénients de la sauvegarde de fichiers

L'esclave *Hot Standby* est toujours un esclave, il est donc impossible de modifier les données et la structure de la base à partir de l'esclave. Il n'est même pas possible d'ajouter des tables de travail (y compris temporaires) et des index, ce qui serait pourtant bien pratique pour générer plus rapidement des rapports. Il est évidemment possible de placer ces index sur le maître mais dans ce cas, ce dernier en sera ralenti lors de modifications dans les tables concernées par ces index.

3.5.3 MISE EN PLACE D'UN HOT STANDBY - MAÎTRE

- Configuration du `postgresql.conf`
  - `wal_level = replica`
- Redémarrage de PostgreSQL

Un serveur esclave en mode *Hot Standby* a besoin de plus d'informations pour accepter des requêtes en lecture seule. Il est donc nécessaire d'augmenter le volume d'informations dans les journaux de transactions. Cela se fait en passant la valeur du paramètre `wal_level` au niveau `replica`.

Avant la version 9.6, il existait deux niveaux intermédiaires pour le paramètre `wal_level` : `archive` et `hot_standby`. Le premier permettait seulement l'archivage, le second permettait en plus d'avoir un serveur secondaire en lecture seule. Ces deux valeurs ont été fusionnées en `replica` avec la version 9.6. Les anciennes valeurs sont toujours acceptées, et remplacées silencieusement par la nouvelle valeur.

Cette valeur `replica` est devenue valeur par défaut en version 10. Cette étape n'est donc plus nécessaire à partir de cette version.

Cette modification nécessite de redémarrer PostgreSQL sur le serveur maître pour que la modification soit prise en compte.

3.5.4 MISE EN PLACE D'UN HOT STANDBY - ESCLAVE

- Configuration du `postgresql.conf`
  - `hot_standby = on`
- Rechargement de la configuration

La configuration d'un serveur en *Hot Standby* change peu de celle d'un serveur en *Warm Standby*. Il suffit de déclarer dans le fichier `postgresql.conf` que le serveur doit se comporter comme un serveur *Hot Standby*. Il existe pour cela le paramètre `hot_standby` qu'il suffit d'activer en le positionnant à `on`.

Le redémarrage du serveur est nécessaire pour prendre en compte cette modification de la configuration.

Ce comportement est activé par défaut en version 10. Cette étape n'est donc plus nécessaire à partir de cette version.

3.6 MISE EN PLACE DE LA STREAMING REPLICATION

- Ne pas faire fichier par fichier
  - mais en flux
- Un processus du maître discute avec un processus de l'esclave
  - d'où un lag moins important
- Asynchrone ou synchrone (9.1)
- En cascade (9.2)

Maintenant que les esclaves sont disponibles en lecture seule, cela nous permet de constater encore plus facilement le retard de l'esclave sur le maître. En effet, nous procédons toujours fichier par fichier. Or la version 9.0 a aussi ajouté la réplication en flux (*Streaming Replication*). Cette méthode de réplication fonctionne de la façon suivante : quand il n'y a plus de journaux de transactions à rejouer sur l'esclave, ce dernier lance un processus appelé `walreceiver` dont le but est de se connecter au maître et d'attendre les modifications de la réplication.

Le `walreceiver` a donc besoin de se connecter sur le maître. Ce dernier doit être configuré pour accepter ce nouveau type de connexion. Lorsque la connexion est acceptée par le maître, le serveur PostgreSQL du maître lance un nouveau processus, appelé `walsender`. Ce dernier a pour but d'envoyer les données de réplication à l'esclave. Les données de réplication sont envoyées suivant l 'activité et certains paramètres de configuration que nous allons aborder dans ce chapitre.

Cette méthode permet une réplication un peu plus proche avec le maître. La version 9.1 permet aussi d'avoir une réplication réellement synchrone, le client ne récupère alors pas la main tant que la modification qu'il a demandé est enregistrée sur le maître et sur l'esclave synchrone.

Enfin, la réplication en cascade permet à un esclave de fournir les informations de réplication à un autre esclave, déchargeant ainsi le maître d'un certain travail et diminuant aussi la bande passante réseau utilisée par le maître.

3.6.1 AVANTAGES

- Un lag de réplication beaucoup moins important
- Donc moins de perte en cas de bascule
- Aucune perte dans le cas du synchrone

L'avantage le plus évident est un retard dans la réplication beaucoup moins important, ce qui sous-entend qu'en cas de bascule forcée la perte sera moins importante que si la réplication se faisait journal de transactions par journal de transactions. La perte est même garantie nulle dans le cas de la mise en place d'une réplication synchrone.

3.6.2 INCONVÉNIENTS

- Si asynchrone
  - perte de données possibles pour ce qui n'a pas été envoyé à l'esclave
- Si synchrone
  - lenteurs à prévoir

Dans le cas de la réplication asynchrone, même si la fenêtre de perte de données a été considérablement réduite, elle n'est pas nulle pour autant.

Dans le cas de la réplication synchrone, la perte de données en cas de bascule est nulle. Cependant, cela engendre forcément un coût sur les performances du système. L'utilisateur n'a la confirmation de la validation d'une transaction qu'à partir du moment où les deux serveurs (le maître et l'esclave synchrone) ont enregistré les données modifiées. Il faut donc prendre en considération le serveur le plus lent et la rapidité du réseau. De plus, si les données sont enregistrées en même temps sur les deux serveurs, cela ne veut pas pour autant dire que les informations sont visibles. Une donnée enregistrée sur le maître ne sera pas forcément immédiatement visible sur l'esclave synchrone. Cette nuance est à prendre en considération avec l'utilisation d'outils de répartition de charge comme pgPool-II.

3.6.3 MEP DE LA STREAMING REPLICATION - MAÎTRE (1/2)

- Configurer `postgresql.conf`
  - `max_wal_senders = X`
  - `wal_sender_timeout = 60s`

Pour passer de la réplication journal par journal à la réplication en flux, il faut tout d'abord configurer le serveur maître pour qu'il accepte la connexion du serveur esclave au niveau de son fichier `postgresql.conf`.

Le serveur maître accepte un nombre maximum de connexions de réplication. Il s 'agit du paramètre `max_wal_senders`. Ce paramètre vaut par défaut 0, et doit être incrémenté d'au moins 1 pour chaque esclave susceptible de se connecter.

À partir de la version 10, ce paramètre vaut par défaut 10.

Le paramètre `wal_sender_timeout` (`replication_timeout` avant la 9.3) permet de couper toute connexion inactive après le délai indiqué par ce paramètre. Par défaut, le délai est d'une minute. Cela permet à un maître de détecter un défaut de connexion de l'esclave.

3.6.4 MEP DE LA STREAMING REPLICATION - MAÎTRE (2/2)

- Configurer `pg_hba.conf`
- L'esclave doit pouvoir se connecter au maître
- Pseudo-base `replication`
- Recharger la configuration

Il est nécessaire après cela de configurer le fichier `pg_hba.conf`. Dans ce fichier, une ligne (par esclave) doit indiquer les connexions de réplication. L'idée est d'éviter que tout le monde puisse se connecter pour répliquer l'intégralité des données.

Pour distinguer une ligne de connexion standard et une ligne de connexion de réplication, la colonne indiquant la base de données doit contenir le mot « replication ». Par exemple :

```
host replication user_repli 10.0.0.2/32 md5
```

Dans ce cas, l'utilisateur `user_repli` pourra entamer une connexion de réplication vers le serveur maître à condition que la demande de connexion provienne de l'adresse IP `10.0.0.2` et que cette demande de connexion précise le bon mot de passe (au format MD5).

A noter qu'à partir de la version 10, les connexions locales de réplication sont autorisées par défaut sans mot de passe.

Après modification du fichier `postgresql.conf` et du fichier `pg_hba.conf`, il est temps de demander à PostgreSQL de recharger sa configuration. L'action `reload` suffit dans tous les cas, sauf celui où `max_wal_senders` est modifié (auquel cas il faudra redémarrer PostgreSQL).

3.6.5 MEP DE LA STREAMING REPLICATION - ESCLAVE

- Configuration du `recovery.conf`

    - standby_mode
    - primary_conninfo
    - trigger_file
    - archive_cleanup_command
- Configuration de postgresql.conf
    - wal_receiver_timeout
- Redémarrage de PostgreSQL

L'esclave doit savoir qu'en cas d'échec de la commande indiquée par le paramètre restore_command, il doit lancer une connexion de réplication vers le maître. Ce type de comportement n'est activé que si le paramètre standby_mode est configuré à on. Il faut aussi qu'il sache comment se connecter au serveur maître. C'est le paramètre primary_conninfo qui le lui dit. Il s'agit d'un DSN standard où il est possible de spécifier l'adresse IP de l'hôte ou son alias, le numéro de port, le nom de l'utilisateur, etc. Il est aussi possible de spécifier le mot de passe mais c'est risqué en terme de sécurité. En effet, PostgreSQL ne vérifie pas si ce fichier est lisible par quelqu'un d'autre que lui. Il est donc préférable de placer le mot de passe dans le fichier .pgpass qui, lui, n'est utilisé que s'il n'est lisible que par son propriétaire.

Pour s'assurer que l'on passera à la connexion de réplication, il ne faut surtout pas utiliser un outil comme pg_standby en tant que restore_command. En effet, pg_standby attend l'arrivée du prochain journal alors que nous voulons que la commande renvoie une erreur si le fichier attendu n'existe pas. Pour cela, nous devons utiliser à une commande comme cp, scp, lftp, etc.

La question qui se pose ensuite est de savoir comment nous allons pouvoir basculer l'esclave en serveur autonome. La solution apportée par PostgreSQL est inspirée directement de celle proposée par pg_standby... par un fichier trigger. Ce dernier se configure avec le paramètre trigger_file. Depuis la version 9.1, nous avons aussi la possibilité d'utiliser pg_ctl avec l'action promote, par exemple :

```
pg_ctl -D $PGDATA promote
```

3.6.6 ASYNCHRONE OU SYNCHRONE

- Par défaut asynchrone
- À partir de la 9.1
    - synchrone possible
    - paramètre synchronous_standby_names
- À partir de la 9.6

  - plusieurs synchrones simultanés possibles
- À partir de la 10
  - synchrones basé sur un quorum
- Changement du statut avec `synchronous_commit`
  - `on`, `off` depuis toujours
  - `local` depuis la 9.1
  - `remote_write` depuis la 9.2
  - `remote_apply` depuis la 9.6

La réplication synchrone est très fréquemment demandée sur tous les moteurs de bases de données. Lorsqu'une base est répliquée de façon asynchrone, cela signifie que lorsqu'un utilisateur écrit une transaction sur le maître, ce dernier indique à l'utilisateur que celle-ci a été validée lorsqu'il a fini d'enregistrer les données dans ses journaux de transactions sur disque. Dans ce mode, il n'attend donc pas de savoir si l'esclave a reçu et encore moins enregistré les données sur disque. Le problème survient alors quand le maître s'interrompt soudainement et qu'il faut basculer l'esclave en maître. Les dernières données enregistrées sur le maître n'ont peut-être pas eu le temps d'arriver sur l'esclave. Par conséquent, on peut se trouver dans une situation où le serveur a indiqué une transaction comme enregistrée mais qu'après le *failover*, elle ne soit plus disponible. Utiliser une réplication synchrone évite ce problème en faisant en sorte que le maître ne valide la transaction auprès de l'utilisateur qu'à partir du moment où l'esclave synchrone a lui-aussi enregistré la donnée sur disque.

Le gros avantage de cette solution est donc de s'assurer qu'en cas de *failover*, aucune donnée ne soit perdue. L'immense inconvénient de cette solution est que cela ajoute de la latence dans les échanges entre le client et le serveur maître pour chaque écriture. En effet, il ne faut pas seulement attendre que le maître fasse l'écriture, il faut aussi attendre l'écriture sur l'esclave sans parler des interactions entre le maître et l'esclave. Même si le coût est minime, il reste cependant présent et dépend aussi de la qualité du réseau. Pour des serveurs réalisant beaucoup d'écritures, le coût n'en sera que plus grand.

Ce sera donc du cas par cas. Pour certains, la réplication synchrone sera obligatoire (due à un cahier des charges réclamant aucune perte de données en cas de *failover*). Pour d'autres, malgré l'intérêt de la réplication synchrone, la pénalité à payer sera trop importante pour se le permettre.

Avec PostgreSQL, passer d'une réplication asynchrone à une réplication synchrone est très simple : il suffit simplement de configurer la variable `synchronous_standby_names`. Ce paramètre doit contenir la liste des esclaves utilisant la réplication synchrone, en les séparant par des virgules. L'ordre des esclaves a un sens : le premier esclave cité sera de priorité 1, le deuxième de priorité 2, etc. Avant la version 9.6, seul un esclave était consid-

éré comme synchrone, les autres indiqués dans la liste étant des remplaçants possible en cas de défaillance du premier. À partir de la version 9.6, il est possible d'indiquer le nombre de serveurs synchrones simultanés. Les serveurs surnuméraires sont des synchrones potentiels. Si ce nombre est indiqué, la liste des serveurs est mis entre parenthèses, comme ceci :

```
synchronous_standby_names = '2 (s1,s2,s3)'
```

Il est à noter que :

```
synchronous_standby_names = '1 (s1,s2,s3)'
```

est strictement équivalent à :

```
synchronous_standby_names = 's1,s2,s3'
```

A partir de la version 10, il est possible de se baser sur un quorum pour choisir les serveurs synchrones :

```
synchronous_standby_names = 'ANY 2 (s1,s2,s3)'
```

La version 10 introduit également le mot clé `FIRST` qui, en remplacement de `ANY`, préserve le même comportement qu'en 9.6.

Mais comment indiquer le nom d'un esclave ? Ce nom dépend d'un paramètre de connexion appelé `application_name`, apparu avec la version 9.0 et est utilisé ici pour distinguer les esclaves. Il doit donc apparaître dans la chaîne de connexion de l'esclave au maître pour la réplication au travers du paramètre `primary_conninfo` dans le fichier `recovery.conf`.

Il est à noter que le statut asynchrone/synchrone peut se changer grâce à un paramètre nommé `synchronous_commit`. Pour les utilisateurs d'anciennes versions de PostgreSQL, ce paramètre était déjà utilisé pour permettre des insertions plus rapides en acceptant un délai [21] dans l'écriture et la synchronisation des journaux de transactions sur disque. Il conserve ce comportement dans le cadre de la réplication asynchrone (et évidemment sans réplication).

Dans le cas de la réplication synchrone, ce paramètre contrôle aussi le fait que les écritures sont faites ou non sur l'esclave. Il peut donc prendre plusieurs valeurs :

- `off` : La transaction est directement validée, mais elle pourra être écrite plus tard dans les journaux. Cela correspond au maximum à 3 fois la valeur de `wal_writer_delay` (200 ms par défaut). **Ce paramétrage peut causer la perte de certaines transactions si le serveur se crashe** et que les données n'ont pas encore été écrites dans les fichiers journaux.

[21] généralement très court, inférieur à trois fois la valeur du paramètre `wal_writer_delay`, soit 600 ms par défaut

- `local` : Permet de fonctionner, pour les prochaines requêtes de la session, en mode de réplication asynchrone. Le commit est validé lorsque les données ont été écrites et synchronisées sur le disque de l'instance primaire. En revanche, l'instance primaire ne s'assure pas que le secondaire a reçu la transaction.
- `remote_write` : Permet d'avoir de la réplication synchrone en mémoire. Cela veut dire que la donnée n'est pas écrite sur disque au niveau de l'esclave mais il l'a en mémoire. Les modifications sont écrites sur disque via le système d'exploitation, mais sans avoir demandé le vidage du cache système sur disque. Les informations sont donc dans la mémoire système. Cela permet de gagner beaucoup en performance, avec une fenêtre de perte de données bien moins importante que le mode asynchrone, mais toujours présente. Si c'est l'instance primaire PostgreSQL qui se crashe, les informations ne sont pas perdues. Par contre, il est possible de perdre des données si l'instance secondaire crashe (en cas de bascule).
- `on` (par défaut) : Le commit est validé lorsque les données ont été écrites et synchronisées sur le disque de l'instance primaire et secondaire. C'est la réplication synchrone. En revanche, l'instance primaire ne s'assure pas que le secondaire a **rejoué** la transaction.
- `remote_apply` : À partir de la version 9.6, les modifications doivent être enregistrées dans les journaux et rejouées avant que la confirmation ne soit envoyée. Cette méthode est la seule garantissant qu'une transaction validée sur le maître sera visible sur le secondaire. Cependant, elle rajoute une latence supplémentaire.

Une autre façon de le voir :

Tableau récapitulatif :

TABLE 1: VALEUR PARAMÈTRE `SYNCHRONOUS_COMMIT`

Ecriture	WAL local (synchronisé sur disque)	Mémoire distant (non-synchronisé sur disque)	WAL distant (synchronisé sur disque)	Rejeu distant (enregistrement visible)
`off`				
`remote_write`	X	X		
`local`	X			
`on`	X		X	
`remote_apply`	X		X	X

3.6.7 RÉPLICATION EN CASCADE

- Avant la 9.2
  - informations de réplication disponibles à partir du maître
  - donc beaucoup de travail sur le maître
  - et beaucoup de bande passante réseau
- À partir de la 9.2
  - un esclave peut fournir les informations de réplication
  - permet de décharger le maître de ce travail
  - permet de diminuer la bande passante du maître

Imaginons un système PostgreSQL installé à Paris et un esclave installé à Marseilles. Il s'avère que le site de Marseilles devient plus important et qu'un deuxième esclave doit y être installé. Avant la 9.2, il aurait fallu faire en sorte que ce deuxième esclave se connecte sur le maître à Paris, faisant doubler la consommation de la bande passante. À partir de la 9.2, il est possible de faire en sorte que le deuxième esclave se connecte au premier esclave (donc en local) pour récupérer les informations de réplication. La bande passante est maîtrisée.

La configuration d'un tel système est très simple. Il suffit d'indiquer l'adresse IP ou l'alias du serveur esclave (à la place de celui du serveur maître) dans le paramètre `primary_conninfo` du fichier `postgresql.conf` du deuxième esclave.

3.7 CONCLUSION

- Système de réplication fiable
- Simple à maîtriser et à configurer
- Forte dynamique de développement

PostgreSQL possède plusieurs solutions de réplications plus au moins complémentaires. La réplication interne à PostgreSQL est le résultat de travaux remontant aussi loin que la version 8.0. Elle est fondée sur des bases solides et saines.

Cette réplication, encore récente, reste fidèle aux principes du moteur de PostgreSQL :

- simple à maîtriser ;
- simple à configurer ;
- fonctionnelle ;
- stable.

La partie fonctionnelle reste encore à étoffer, mais PostgreSQL possède déjà des fonctionnalités de réplication très avancées, telle que le choix du synchronisme de la réplication à la transaction près !

Enfin, cette partie du code de PostgreSQL avance aujourd'hui à grand pas. Les bases de la réplication étant désormais jetées et éprouvées, le développement reste très actif pour y ajouter toujours plus de fonctionnalités en fonction des retours du terrain.

3.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.8 TRAVAUX PRATIQUES

3.8.1 ÉNONCÉS

Informations importantes Tout se fait sur la même machine. N'oubliez pas qu'il faut un répertoire de données et un numéro de port par serveur PostgreSQL.

Réplication asynchrone avec un seul secondaire

Créer un serveur principal.

Ajouter un serveur secondaire par archivage des fichiers de transactions.

Assurez-vous que la réplication fonctionne bien. Pouvez-vous vous connecter au serveur secondaire ?

Réplication asynchrone avec un seul secondaire Warm Standby

Comment faire pour empêcher les connexions en lecture sur le secondaire ?

Assurez-vous que la réplication fonctionne bien. Pouvez-vous vous connecter au serveur secondaire ? Si oui, quel type de requêtes pouvez-vous exécuter ?

Réplication asynchrone en flux avec un seul secondaire Hot Standby

Passer à la *Streaming Replication*.

17.12

Assurez-vous que la réplication fonctionne bien. Que constatez-vous ?

Réplication asynchrone en flux avec deux secondaires

Ajouter un deuxième serveur secondaire avec l'outil `pg_basebackup`. Ce serveur secondaire sera aussi mis à jour grâce à la *Streaming Replication*.

Assurez-vous que la réplication fonctionne bien. Pouvez-vous vous connecter au serveur secondaire ? Si oui, quel type de requêtes pouvez-vous exécuter ?

Nettoyage des journaux

Actuellement il n'y a aucun nettoyage des journaux archivés même s'ils ont été rejoués sur les secondaires. Quel paramètre modifier pour supprimer les anciens journaux?

Sachant que les deux secondaires puisent leur journaux depuis le même répertoire d'archive, quel pourrait être le risque?

Slots de réplication

Depuis la version 9.4

Il est possible d'éviter d'avoir recours à l'archivage en utilisant les slots de réplication. Ces derniers permettent au serveur principal de savoir quels sont les journaux encore nécessaires au serveur secondaire.

Créer un slot de réplication et configurer le deuxième secondaire pour utiliser ce slot. Contrôler que le slot est bien actif.

Arrêter le deuxième secondaire et générer beaucoup d'activité. Où sont conservés les journaux de transaction? Quel est le journal le plus ancien sur le serveur principal? Et dans les journaux archivés?

Que se serait-il passé sans slot de réplication ?

Démarrer le deuxième secondaire et contrôler que les deux secondaires sont bien en *Streaming Replication*.

Réplication synchrone en flux avec deux secondaires

Passer la réplication en synchrone pour un seul secondaire.

Arrêter le secondaire synchrone. Exécuter une requête de modification sur le principal. Que se passe-t-il ?

Redémarrer le secondaire synchrone.

Passer le deuxième secondaire en synchrone.

Arrêter le premier secondaire synchrone. Exécuter une requête de modification sur le principal. Que se passe-t-il ?

Redémarrer le premier secondaire.

Est-ce que les deux secondaires sont synchrones ? Quel paramètre modifier pour avoir deux secondaires synchrones ?

Pour la suite du TP repasser sur un seul serveur synchrone.

Réplication synchrone : cohérence des lectures (optionnel)

Depuis la version 9.6

Exécuter la commande `SELECT pg_wal_replay_pause();` sur le premier secondaire synchrone. Ajouter des données sur le principal et contrôler leur présence sur le secondaire. Que constatez-vous ?

Est-ce que les instances sont bien synchrones (utilisez la vue `pg_stat_replication`) ? Relancer le rejeu et contrôler la présence des enregistrements sur les deux instances.

Quel paramètre modifier pour obtenir les mêmes résultats sur les deux instances ?

Appliquer ce paramètre et effectuer la même opération (pause du rejeu puis insertion d'enregistrements sur le principal). Que constatez-vous ?

Réplication en cascade (optionnel)

Créer un troisième secondaire et placer le en cascade du premier secondaire.

3.8.2 SOLUTIONS

Le prompt `#` indique une commande à exécuter avec l'utilisateur `root`. Le prompt `$` est utilisé pour les commandes de l'utilisateur `postgres`.

Quelques informations sur la solution proposée ici

17.12

Cette solution se base sur un système CentOS 6, installé à minima. Des adaptations seraient à faire pour un autre système (Linux ou Windows).

La solution apportée ici se base beaucoup sur la commande `sudo` qui permet à un utilisateur d'exécuter des commandes que seul un autre utilisateur est habituellement permis à exécuter. Si `sudo` n'est pas installé, son installation est donc conseillée :

```
# yum install sudo
```

La configuration de `sudo` se fait avec la commande `visudo`. Le but est d'ajouter la ligne suivante :

```
utilisateur ALL=(ALL) ALL
```

En remplaçant `utilisateur` par le nom de l'utilisateur à qui on veut autoriser l'exécution de toute commande via la commande `sudo`.

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

```
# yum install -y https://download.postgresql.org/pub/repos/yum
           /testing/10/redhat/rhel-6-x86_64/pgdg-centos10-10-1.noarch.rpm
Retrieving https://download.postgresql.org/pub/repos/yum
           /testing/10/redhat/rhel-6-x86_64/pgdg-centos10-10-1.noarch.rpm
warning: /var/tmp/rpm-tmp.RxZogq: Header V4 DSA/SHA1 Signature,
                                  key ID 442df0f8: NOKEY
Preparing...                ########################################**[100%]
   1:pgdg-centos10          ########################################**[100%]
```

```
# yum install -y postgresql10-server postgresql10-contrib
```

Les paquets suivants seront installés :

```
(1/12): libicu-4.2.1-14.el6.x86_64.rpm
(2/12): libxslt-1.1.26-2.el6_3.1.x86_64.rpm
(3/12): perl-5.10.1-144.el6.x86_64.rpm
(4/12): perl-Module-Pluggable-3.90-144.el6.x86_64.rpm
(5/12): perl-Pod-Escapes-1.04-144.el6.x86_64.rpm
(6/12): perl-Pod-Simple-3.13-144.el6.x86_64.rpm
(7/12): perl-libs-5.10.1-144.el6.x86_64.rpm
(8/12): perl-version-0.77-144.el6.x86_64.rpm
(9/12): postgresql10-10.0-beta4_1PGDG.rhel6.x86_64.rpm
(10/12): postgresql10-contrib-10.0-beta4_1PGDG.rhel6.x86_64.rpm
(11/12): postgresql10-libs-10.0-beta4_1PGDG.rhel6.x86_64.rpm
(12/12): postgresql10-server-10.0-beta4_1PGDG.rhel6.x86_64.rpm
```

Réplication asynchrone avec un seul secondaire Commençons par créer une instance PostgreSQL 10 :

```
# service postgresql-10 initdb
Initialisation de la base de données :                     [  OK  ]
```

Il reste maintenant à configurer ce serveur pour qu'il archive les journaux de transactions. Mais tout d'abord, il faut créer le répertoire d'archivage.

```
$ mkdir /var/lib/pgsql/10/archives
```

La commande ayant été exécutée par l'utilisateur `postgres`, les droits sont automatiquement bons.

Modifions maintenant le fichier `/var/lib/pgsql/10/data/postgresql.conf` pour que PostgreSQL archive les journaux de transactions et écrive ses logs en anglais. Voici les modifications à apporter :

```
archive_mode = on
archive_command = 'rsync %p /var/lib/pgsql/10/archives/%f'
lc_messages = 'C'
```

La commande `rsync` n'est pas installée par défaut. S'il est nécessaire de l'installer, voici la commande adéquate :

```
# yum install -y rsync
```

Le paramètre `archive_mode` étant modifié, il nous faut redémarrer PostgreSQL :

```
# service postgresql-10 start
Démarrage du service postgresql-10 :                       [  OK  ]
```

Forçons PostgreSQL à changer de journal de transactions, pour voir si l'archivage fonctionne bien :

```
$ psql -c "SELECT pg_switch_wal()"
 pg_switch_wal
---------------
 0/168B71C
(1 row)

$ ls -l /var/lib/pgsql/10/archives/
total 16384
-rw------- 1 postgres postgres 16777216 Sep 29 10:04 000000010000000000000001
```

Parfait.

La valeur renvoyée par la fonction `pg_switch_wal()` peut varier suivant la quantité de données écrites précédemment par PostgreSQL.

Maintenant que l'archivage fonctionne, ajoutons un secondaire. Nous pourrions utiliser `pg_basebackup`, c'est d'ailleurs ce qu'on fera pour le troisième secondaire. Nous allons le faire manuellement pour cette fois.

Tout d'abord, nous devons utiliser la procédure stockée `pg_start_backup()` :

```
$ psql -c "SELECT pg_start_backup('un_label', true)"
 pg_start_backup
-----------------
 0/2000028
(1 row)
```

Ensuite, il faut copier le répertoire des données :

```
$ cp -rp /var/lib/pgsql/10/data /var/lib/pgsql/10/secondaire1
```

Il est possible d'appeler `pg_stop_backup()` :

```
$ psql -c "SELECT pg_stop_backup()"
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/20000B8
(1 row)
```

Sur CentOS, les fichiers de configuration se trouvent dans le répertoire de données, ils sont donc copiés automatiquement. Cependant, avant d'aller plus loin et de procéder à la configuration de la mise en réplication, il faut configurer cette instance pour démarrer sur le port 5433 et lui permettre de démarrer automatiquement.

Le fichier `/etc/sysconfig/pgsql/secondaire1` va permettre de définir l'emplacement de l'instance et son port d'écoute par défaut :

```
PGDATA=/var/lib/pgsql/10/secondaire1
PGPORT=5433
```

Il faut également positionner le paramètre `port` dans le fichier de configuration `/var/lib/pgsql/10/secondaire1/postgresql.conf` :

```
port=5433
```

Enfin, il faut créer le script de démarrage `/etc/init.d/secondaire1` qui nous permettra de démarrer l'instance :

```
# ln -s /etc/init.d/postgresql-10 /etc/init.d/secondaire1
```

Il ne reste plus qu'à configurer la restauration dans le fichier `recovery.conf`. Attention, ce dernier, contrairement aux autres, se trouvera dans le répertoire des données (donc ici `/var/lib/postgresql/10/secondaire1`). Pour se faciliter la vie, nous allons copier le fichier d'exemple :

```
$ cp /usr/pgsql-10/share/recovery.conf.sample \
     /var/lib/pgsql/10/secondaire1/recovery.conf
```

Enfin, il reste à configurer le fichier /var/lib/pgsql/10/secondaire1/recovery.conf en modifiant ces paramètres :

```
restore_command = 'cp /var/lib/pgsql/10/archives/%f %p'
standby_mode = on
```

Un peu de ménage au niveau des fichiers du secondaire :

```
$ rm /var/lib/pgsql/10/secondaire1/postmaster.pid
     /var/lib/pgsql/10/secondaire1/pg_wal/*
```

Le message d'erreur sur le répertoire archive_status peut être ignoré sans conséquence.

Il ne reste plus qu'à démarrer le secondaire :

```
# service secondaire1 start
Démarrage du service secondaire1 :                          [  OK  ]
```

La commande ps suivante permet de voir que les deux serveurs sont lancés :

```
$ ps -o pid,cmd fx
```

La première partie concerne le serveur secondaire :

```
PID CMD
881 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire1
883  \_ postgres: logger process
884  \_ postgres: startup process   waiting for 000000010000000000000006
886  \_ postgres: checkpointer process
887  \_ postgres: writer process
888  \_ postgres: stats collector process
```

La deuxième partie concerne le serveur principal :

```
PID CMD
535 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
537  \_ postgres: logger process
539  \_ postgres: checkpointer process
540  \_ postgres: writer process
541  \_ postgres: wal writer process
542  \_ postgres: autovacuum launcher process
543  \_ postgres: archiver process   last was 000000010000000000000005.00000028.backup
544  \_ postgres: stats collector process
545  \_ postgres: bgworker: logical replication launcher
```

Pour différencier les deux instances, il est possible d'identifier le répertoire de données (l'option `-D`), les autres processus sont des fils du processus postmaster. Il est aussi possible de configurer le paramètre `cluster_name`.

Le processus de démarrage (*startup process*) indique qu'il attend le journal 000000010000000000000006. Pour le récupérer, il exécute la `restore_command` en indiquant le journal à récupérer.

Lançons un peu d'activité sur le principal pour générer (et surtout archiver) quelques journaux de transactions :

```
$ createdb b1
$ psql b1
psql (10)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);
CREATE TABLE
b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
INSERT 0 1000000
b1=# \! ps -o pid,cmd fx | egrep "(archiver|startup)"
  884  \_ postgres: startup process   waiting for 000000010000000000000007
  543  \_ postgres: archiver process   last was 000000010000000000000006
[...]
```

Le processus d'archivage (« archiver process ») a archivé le journal 000000010000000000000006. Le secondaire attend donc le suivant, à savoir le 000000010000000000000007. Les journaux applicatifs indiquent ceci :

```
$ grep "restored log file" /var/lib/pgsql/10/secondaire1/log/postgresql*.log
2017-09-06 10:13:32.862 EDT [790] LOG: restored log file
                                       "000000010000000000000003" from archive
2017-09-06 10:14:30.358 EDT [884] LOG: restored log file
                                       "000000010000000000000004" from archive
2017-09-06 10:44:07.671 EDT [884] LOG: restored log file
                                       "000000010000000000000005" from archive
2017-09-06 10:44:10.069 EDT [884] LOG: restored log file
                                       "000000010000000000000006" from archive
```

Essayons de nous connecter au secondaire et d'exécuter quelques requêtes :

```
$ psql -p 5433
psql (10)
Type "help" for help.
```

```
postgres=# SELECT * FROM pg_tablespace;
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
(2 rows)

postgres=# CREATE TABLE t1(id integer);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

On peut se connecter, lire des données, mais pas écrire.

Le comportement est visible dans les logs de l'instance secondaire dans le répertoire `/var/lib/pgsql/10/secondaire1/log` :

```
2017-09-06 10:05:44.546 EDT [535] LOG: database system is ready to accept connections
```

PostgreSQL indique bien qu'il accepte des connexions en lecture seule.

Réplication asynchrone avec un seul secondaire en *Warm Standby*

Depuis la version 10, le comportement de PostgreSQL a changé et un secondaire sera par défaut en *Hot Standby*. Un secondaire sera donc accessible par défaut en lecture seule. Pour désactiver ce comportement, il faut passer le paramètre `hot_standby` à `off` sur le secondaire.

Il faut redémarre l'instance secondaire pour prendre en comtpe les changements :

```
# service secondaire1 restart
Stopping secondaire1 service:                              [  OK  ]
Starting secondaire1 service:                              [  OK  ]
```

```
$ ps -o pid,cmd fx | egrep "(archiver|startup)"
  884  \_ postgres: startup process   waiting for 000000010000000000000007
  543  \_ postgres: archiver process   last was 000000010000000000000006
[...]
```

Si nous essayons de nous connecter à l'instance nous obtenons le message :

```
$ psql -p 5433
psql: FATAL:  the database system is starting up
```

Il est impossible de se connecter à l'instance en *Warm Standby*

Réplication asynchrone en flux avec un seul secondaire

Depuis la version 10, le comportement de PostgreSQL a changé et la réplication est activée par défaut en local.

Nous allons cependant modifier le fichier `/var/lib/pgsql/10/data/pg_hba.conf` pour que l'accès en réplication soit autorisé pour l'utilisateur `repli` :

```
host  replication  repli  127.0.0.1/32  md5
```

Cette configuration indique que l'utilisateur `repli` peut se connecter en mode réplication à partir de l'adresse IP `127.0.0.1`. L'utilisateur `repli` n'existant pas, il faut le créer (nous utiliserons le mot de passe *repli*) :

```
$ createuser -SRD --replication -P repli
Enter password for new role:
Enter it again:
```

Pour prendre en compte la configuration, la configuration de l'instance principale doit être rechargée :

```
$ psql -c 'SELECT pg_reload_conf()'
```

Maintenant, il faut configurer le secondaire pour qu'il se connecte au principal. Cela se fait au travers du fichier `/var/lib/pgsql/10/secondaire1/recovery.conf` et voici les paramètres à modifier :

```
restore_command = 'cp /var/lib/pgsql/10/archives/%f %p'
standby_mode = on
primary_conninfo = 'host=127.0.0.1 port=5432 user=repli password=repli'
trigger_file = '/tmp/secondaire1_autonome'
```

Il faut que la `restore_command` renvoie un code retour différent de zéro pour indiquer au moteur qu'il n'y a plus de journaux de transaction et de passer en réplication en flux. Notez aussi que le fichier `trigger_file` ne doit pas exister. Une vérification est nécessaire avant de procéder à la suite.

Il ne reste plus qu'à redémarrer le secondaire :

```
# service secondaire1 restart
```

Voici la liste des processus :

```
$ ps -o pid,cmd fx
  PID CMD
 4971 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire1
 4973  \_ postgres: logger process
 4974  \_ postgres: startup process   recovering 000000010000000000000006
 4976  \_ postgres: checkpointer process
```

```
 4977  \_ postgres: writer process
 5006  \_ postgres: stats collector process
 5008  \_ postgres: wal receiver process   streaming 0/7000000
 4634 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
 4636  \_ postgres: logger process
 4638  \_ postgres: checkpointer process
 4639  \_ postgres: writer process
 4640  \_ postgres: wal writer process
 4641  \_ postgres: autovacuum launcher process
 4642  \_ postgres: archiver process   last was 000000010000000000000006
 4643  \_ postgres: stats collector process
 4644  \_ postgres: bgworker: logical replication launcher
 5009  \_ postgres: wal sender process repli 127.0.0.1(45754) streaming 0/7D52830
```

Nous avons bien les deux processus de réplication en flux `wal sender` et `wal receiver`.

Réplication asynchrone en flux avec deux secondaires

Cette fois-ci, nous allons utiliser `pg_basebackup` pour créer le deuxième secondaire.

```
$ pg_basebackup -D /var/lib/pgsql/10/secondaire2 -P -h 127.0.0.1 -U repli
Password:
61401/61401 kB (100%), 1/1 tablespace
```

Les emplacements et le numéro de port du deuxième secondaire doit être différents des deux autres serveurs. Il faut donc créer le fichier `/etc/sysconfig/pgsql/secondaire2` pour indiquer :

```
PGDATA=/var/lib/pgsql/10/secondaire2
PGPORT=5434
```

Il faut également mettre en place le script de démarrage :

```
# ln -s /etc/init.d/postgresql-10 /etc/init.d/secondaire2
```

Il est aussi nécessaire de modifier le numéro de port d'écoute de l'instance dans `/var/lib/pgsql/10/secondaire2/postgresql.conf` :

```
port = 5434
```

Le fichier de restauration doit être copié :

```
$ cp /var/lib/pgsql/10/secondaire1/recovery.conf
     /var/lib/pgsql/10/secondaire2/recovery.conf
```

Et légèrement modifié sur le nouvel secondaire :

```
trigger_file = '/tmp/secondaire2_autonome'
```

Et on peut enfin démarrer le deuxième secondaire :

```
# service secondaire2 start
```

Testons cette nouvelle architecture :

```
$ psql -p 5434
psql (10)
Type "help" for help.

postgres=# SELECT * FROM pg_tablespace;
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
(2 rows)

postgres=# CREATE TABLE t1(id integer);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
postgres=# \c "port=5432 dbname=b1"
Vous êtes maintenant connecté à la base de données « b1 » en tant qu'utilisateur
« postgres » via le socket dans « /var/run/postgresql » via le port « 5432 ».
b1=# INSERT INTO t1 SELECT generate_series(1, 1000);
INSERT 0 1000
postgres=# \c "port=5433 dbname=b1"
Vous êtes maintenant connecté à la base de données « b1 » en tant qu'utilisateur
« postgres » via le socket dans « /var/run/postgresql » via le port « 5433 ».
b1=# SELECT count(*) FROM t1;
  count
---------
 1001000
(1 row)

b1=# \c "port=5434 dbname=b1"
Vous êtes maintenant connecté à la base de données « b1 » en tant qu'utilisateur
« postgres » via le socket dans « /var/run/postgresql » via le port « 5434 ».
b1=# SELECT count(*) FROM t1;
  count
---------
 1001000
(1 row)
```

Nettoyage des journaux Actuellement il n'y a aucun nettoyage des journaux archivés même s'ils ont été rejoués sur les secondaires. Quel paramètre modifier pour supprimer

les anciens journaux ?

> Le paramètre `archive_cleanup_command` du fichier recovery.conf permet de spécifier une commande exécutée à la fin d'un `restartpoint` (équivalent d'un checkpoint sur un secondaire). L'outil `pg_archivecleanup` est utilisé pour supprimer les journaux inutiles.

```
archive_cleanup_command =
     '/usr/pgsql-10/bin/pg_archivecleanup -d /var/lib/pgsql/10/archives/ %r'
```

En générant de l'activité et en forçant des `CHECKPOINT`, le moteur va recycler ses journaux :

```
$ psql -c "INSERT INTO t1 SELECT * FROM generate_series(1,1000);" b1
$ psql -c "CHECKPOINT;"
$ psql -p 5433 -c "CHECKPOINT;"
```

L'option `-d` permet d'avoir des informations supplémentaires dans les traces :

```
pg_archivecleanup: keeping WAL file "/var/lib/pgsql/10/archives//000000010000000000000
                   and later
pg_archivecleanup: removing file "/var/lib/pgsql/10/archives//000000010000000000000060
pg_archivecleanup: removing file "/var/lib/pgsql/10/archives//00000001000000000000005F
pg_archivecleanup: removing file "/var/lib/pgsql/10/archives//000000010000000000000061
```

> Sachant que les deux secondaires puisent leur journaux depuis le même répertoire d'archive. Quel pourrait être le risque ?

Le premier secondaire pourrait supprimer des journaux indispensables au deuxième secondaire. Sans ces journaux, la réplication du deuxième secondaire serait impossible et nécessiterait la reconstruction de celui-ci.

Slots de réplication > Il est possible d'éviter d'avoir recours à l'archivage en utilisant les slots de réplication. Ces derniers permettent au serveur principal de connaître les journaux encore nécessaires au serveur secondaire.

> Créer un slot de réplication et configurer le deuxième secondaire pour utiliser ce slot. Contrôler que le slot est bien actif.

Depuis la version 10, les slots de réplication sont activés par défaut. Le nombre maximum de slots est fixé à 10 :

```
# SHOW max_replication_slots;
 max_replication_slots
-----------------------
 10
(1 row)
```

La commande suivante permet de créer un slot de réplication sur le serveur principal :

```
$ psql -c "SELECT pg_create_physical_replication_slot('slot_secondaire2');"
 pg_create_physical_replication_slot
-------------------------------------
 (slot_secondaire2,)
(1 row)
```

Il faut ensuite spécifier le slot dans le fichier `recovery.conf` :

```
primary_slot_name = 'slot_secondaire2'
```

Puis redémarrer le serveur secondaire.

```
# service secondaire2 restart
Stopping secondaire2 service:                              [  OK  ]
Starting secondaire2 service:                              [  OK  ]
```

Enfin, l'appel à la vue `pg_replication_slots` permet de s'assurer que le slot est bien actif :

```
postgres=# select * from pg_replication_slots ;
-[ RECORD 1 ]-------+-----------------
slot_name           | slot_secondaire2
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 1666
xmin                |
catalog_xmin        |
restart_lsn         | 0/62D5ACD8
confirmed_flush_lsn |
```

> Arrêter le deuxième secondaire et générer beaucoup d'activité. Où sont conservés les journaux de transaction ? Quel est le journal le plus ancien sur le serveur principal ? Et dans les journaux archivés ?

Pour générer de l'activité :

```
psql -c "INSERT INTO t1 SELECT * FROM generate_series(1,1000000);" b1
```

En regardant les journaux au niveau du serveur principal et au niveau des archives :

```
$ ls -alh /var/lib/pgsql/10/data/pg_wal/
(...)
-rw-------  1 postgres postgres  16M  7 sept. 05:58 000000010000000000000008C
-rw-------  1 postgres postgres  16M  7 sept. 05:58 000000010000000000000008D

$ ls -alh /var/lib/pgsql/10/archives/
(...)
-rw------- 1 postgres postgres  16M  7 sept. 06:06 000000010000000000000064
-rw------- 1 postgres postgres  16M  7 sept. 06:06 000000010000000000000065
```

On constate que le principal a conservé les anciens journaux dans le répertoire `pg_wal`.

> Que ce serait-il passé sans slot de réplication ?

Le deuxième secondaire n'aurait pas pu récupérer des journaux indispensables à la réplication.

> Démarrer le deuxième secondaire et contrôler que les deux secondaires sont bien en *Streaming Replication*.

Au démarrage :

```
$  psql -x -c "SELECT * FROM pg_stat_replication"
-[ RECORD 1 ]----+------------------------------
pid              | 1249
usesysid         | 16388
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 45932
backend_start    | 2017-09-07 05:47:43.316465-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/66A951D0
write_lsn        | 0/66A951D0
flush_lsn        | 0/66A951D0
replay_lsn       | 0/66A951D0
write_lag        |
```

```
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 1801
usesysid         | 16388
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 45946
backend_start    | 2017-09-07 06:08:02.109247-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/66A951D0
write_lsn        | 0/66A951D0
flush_lsn        | 0/66A951D0
replay_lsn       | 0/66A951D0
write_lag        | 00:00:01.766658
flush_lag        | 00:00:01.934458
replay_lag       | 00:00:02.142817
sync_priority    | 0
sync_state       | async
```

Au bout de quelques minutes il est enfin synchronisé :

```
$ psql -x -c "SELECT * FROM pg_stat_replication"
-[ RECORD 1 ]----+------------------------------
pid              | 1249
usesysid         | 16388
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 45932
backend_start    | 2017-09-07 05:47:43.316465-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/66A951D0
write_lsn        | 0/66A951D0
flush_lsn        | 0/66A951D0
replay_lsn       | 0/66A951D0
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
```

```
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 1801
usesysid         | 16388
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 45946
backend_start    | 2017-09-07 06:08:02.109247-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/66A951D0
write_lsn        | 0/66A951D0
flush_lsn        | 0/66A951D0
replay_lsn       | 0/66A951D0
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

Réplication synchrone en flux avec deux secondaires Nous allons passer le premier secondaire en tant que secondaire synchrone. Pour cela, il doit avoir un nom, indiqué par le paramètre de connexion `application_name`. Le fichier de configuration `/var/lib/pgsql/10/secondaire1/recovery.conf` doit être modifié ainsi :

```
primary_conninfo = 'host=127.0.0.1 port=5432 user=repli password=repli
                    application_name=secondaire1'
```

Ensuite, nous devons indiquer le serveur `secondaire` dans la liste des serveurs synchrones initialisée par le paramètre `synchronous_standby_names` du fichier `/var/lib/pgsql/10/data/postgresql.conf` :

```
synchronous_standby_names = 'secondaire1'
```

Il ne reste plus qu'à recharger la configuration pour les deux serveurs :

```
# service postgresql-10 reload
# service secondaire1 restart
```

Il n'est pas nécessaire de redémarrer les trois serveurs. Un `reload` du principal et un redémarrage du premier secondaire suffisent.

Vérifions l'état de la réplication pour les deux secondaires :

```
$ psql -p 5432
psql (10)
Type "help" for help.

postgres=# \x
Expanded display is on.
postgres=# SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
application_name | walreceiver
backend_start    | 2017-09-07 06:08:02.109247-04
state            | streaming
sync_state       | async
-[ RECORD 2 ]----+------------------------------
application_name | secondaire1
backend_start    | 2017-09-07 06:11:06.67367-04
state            | streaming
sync_state       | sync
```

Nous avons bien un serveur synchrone et un serveur asynchrone.

Exécutons une requête de modification sur le principal :

```
$ psql -p 5432 b1
psql (10)
Type "help" for help.

b1=# CREATE TABLE t2(id integer);
CREATE TABLE
```

La table est bien créée, sans attendre. Maintenant, arrêtons le serveur secondaire synchrone et faisons une nouvelle modification sur le principal :

```
# service secondaire1 stop
$ psql -p 5432 b1
psql (10)
Type "help" for help.

b1=# CREATE TABLE t3(id integer);
```

La requête reste bloquée. En effet, le secondaire ne peut pas répondre à la demande de la réplication car il est éteint. Du coup, le principal est bloqué en écriture. Il faut soit démarrer le secondaire, soit modifier la configuration du paramètre `synchronous_standby_names`.

Démarrons le secondaire synchrone à partir d'un autre terminal et la requête se termine.

Nous allons maintenant passer le deuxième secondaire en synchrone avec le

`application_name` positionné à secondaire2 afin de les différencier (il est possible d'utiliser le même `application_name`). Ensuite ajoutons `secondaire2` à la liste des `synchronous_standby_names` sur l'instance principale.

```
# service postgresql-10 reload
# service secondaire2 restart

$ psql -x -p 5432 -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"
-[ RECORD 1 ]----+------------------------------
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sync_state       | potential
-[ RECORD 2 ]----+------------------------------
application_name | secondaire1
backend_start    | 2017-09-07 06:12:46.841986-04
state            | streaming
sync_state       | sync

$ psql -p 5432 -c "SHOW synchronous_standby_names"
 synchronous_standby_names
---------------------------
 secondaire1,secondaire2
```

Cette fois les deux serveurs correspondent au `synchronous_standby_names`, on peut constater qu'un serveur est `sync` et l'autre `potential`. Il ne peut y avoir qu'un seul serveur synchrone avec le principal. Si les deux serveurs avaient le même `application_name`, il n'y aurait eu qu'un seul serveur `sync`.

Arrêtons ensuite le premier secondaire synchrone :

```
# service secondaire1 stop

$ psql -p 5432 -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"
-[ RECORD 1 ]----+------------------------------
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sync_state       | sync
```

Et faisons une modification sur le principal :

```
$ psql -p 5432 -c "CREATE TABLE t4(id integer);" b1
CREATE TABLE
```

Cette fois, tout se passe bien. Le premier secondaire n'est pas disponible mais le second l'est. Il prend donc la suite du premier secondaire en tant que secondaire synchrone.

Si on souhaite avoir deux serveurs synchrones, il faut utiliser la syntaxe suivante pour le paramètre `synchronous_standby_names` :

- `synchronous_standby_names = 'N (liste serveurs)'`
- `N` = nombre de serveurs synchrones

Dans notre cas `synchronous_standby_names = '2(secondaire1,secondaire2)'`

Après un reload du principal on constate bien que les deux serveurs sont synchrones :

```
$ psql -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"
-[ RECORD 1 ]----+------------------------------
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sync_state       | sync
-[ RECORD 2 ]----+------------------------------
application_name | secondaire1
backend_start    | 2017-09-07 06:19:59.248803-04
state            | streaming
sync_state       | sync
```

Pour repasser sur un seul serveur synchrone :

```
synchronous_standby_names = 'secondaire1,secondaire2'
```

ou :

```
synchronous_standby_names = '1(secondaire1,secondaire2)'
```

ou encore depuis la version 10 :

```
synchronous_standby_names = 'FIRST 1(secondaire1,secondaire2)'
```

L'ordre de cette liste donne la priorité qui sera utilisée pour l'application de la réplication synchrone. La version 10 permet de se baser sur un quorum :

```
synchronous_standby_names = 'ANY 1(secondaire1,secondaire2)'
```

On obtient alors un `sync_state` à la valeur `quorum` :

```
$ psql -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"
FROM pg_stat_replication;"
-[ RECORD 1 ]----+------------------------------
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
```

```
application_name | secondaire1
backend_start    | 2017-09-07 06:19:59.248803-04
state            | streaming
sync_state       | quorum
```

Réplication synchrone : cohérence des lectures (optionnel) > Exécuter la commande `SELECT pg_wal_replay_pause();` sur le premier secondaire synchrone. Ajouter des données sur le principal et contrôler leur présence sur le secondaire. Que constatez-vous ?

```
$ psql -p 5433 -c "SELECT pg_wal_replay_pause()"
```

```
$ psql -p 5432 b1
b1=# INSERT INTO t4 VALUES ('1');
INSERT 0 1
b1=# SELECT * FROM t4;
 c1
----
  1
(1 row)
```

```
$ psql -p 5433 -c "SELECT * FROM t4;" b1
 id
----
(0 rows)
```

```
$ psql -p 5434 -c "SELECT * FROM t4;" b1
 id
----
  1
(1 row)
```

La table est vide sur le premier secondaire synchrone mais elle est bien remplie sur le deuxième secondaire !

> Est-ce que les instances sont bien synchrones (utilisez la vue `pg_stat_replication`) ? Relancer le rejeu et contrôler la présence des enregistrements sur les deux serveurs.

```
$ psql -p 5432
postgres=# \x
Expanded display is on.
postgres=# SELECT application_name, backend_start, state, sent_lsn,
                  write_lsn, flush_lsn, replay_lsn, sync_state
           FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
```

```
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sent_lsn         | 0/66AC5098
write_lsn        | 0/66AC5098
flush_lsn        | 0/66AC5098
replay_lsn       | 0/66AC5098
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
application_name | secondaire1
backend_start    | 2017-09-07 06:19:59.248803-04
state            | streaming
sent_lsn         | 0/66AC5098
write_lsn        | 0/66AC5098
flush_lsn        | 0/66AC5098
replay_lsn       | 0/66AC4F18
sync_state       | quorum
```

Le premier secondaire est bien synchrone, on constate que les deux ont bien reçu les enregistrements mais seul le secondaire2 a rejoué les journaux.

On réactive le rejeu sur le premier secondaire :

```
$ psql -p 5433 -c "SELECT pg_wal_replay_resume()"
 pg_wal_replay_resume
----------------------


(1 row)

$ psql -p 5432
postgres=# \x
Expanded display is on.
postgres=# SELECT application_name, backend_start, state, sent_lsn,
                  write_lsn, flush_lsn, replay_lsn, sync_state
           FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
application_name | secondaire2
backend_start    | 2017-09-07 06:15:40.969515-04
state            | streaming
sent_lsn         | 0/66AC5098
write_lsn        | 0/66AC5098
flush_lsn        | 0/66AC5098
replay_lsn       | 0/66AC5098
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
application_name | secondaire1
backend_start    | 2017-09-07 06:19:59.248803-04
state            | streaming
```

```
sent_lsn          | 0/66AC5098
write_lsn         | 0/66AC5098
flush_lsn         | 0/66AC5098
replay_lsn        | 0/66AC5098
sync_state        | quorum
```

Cette fois, le secondaire1 a bien rejoué les journaux. Les enregistrements sont bien présents dans la table t4 :

```
$ psql -p 5433 -c "SELECT * FROM t4;" b1
 id
----
  1
(1 row)
```

> Quel paramètre modifier pour obtenir les mêmes résultats sur les deux instances ?

Par défaut la réplication synchrone garantie qu'aucune transaction n'est perdue mais elle ne s'assure pas que le secondaire synchrone a bien rejoué la transaction. Pour cela, il faut placer le paramètre `synchronous_commit` à `remote_apply` sur le principal.

Appliquer ce paramètre et effectuer la même opération (pause du rejeu puis insertion d'enregistrement sur le principal). Que constatez-vous?

Dans `/var/lib/pgsql/10/data/postgresql.conf` :

```
synchronous_commit = remote_apply
```

Faire un rechargement de la configuration du serveur principal :

```
# service postgresql-10 reload
```

```
$ psql -p 5433 -c "SELECT pg_wal_replay_pause()"
 pg_wal_replay_pause
---------------------

(1 row)
$ psql -p 5434 -c "SELECT pg_wal_replay_pause()"
 pg_wal_replay_pause
---------------------

(1 row)
$ psql -p 5432 -c "INSERT INTO t4 VALUES ('2');" b1
```

Cette fois la requête est bloquée, il faut relancer le rejeu sur au moins un secondaire pour qu'elle puisse s'effectuer.

17.12

Réplication en cascade (optionnel) Nous allons créer un troisième secondaire et le placer en cascade du premier secondaire.

Créons le troisième secondaire avec `pg_basebackup` :

```
$ pg_basebackup -D /var/lib/pgsql/10/secondaire3 -P -h 127.0.0.1 -U repli
Password:
178657/178657 Ko (100%), 1/1 tablespace
```

Remarquez que nous le créons à partir du principal. Nous aurions très bien pu le créer à partir du secondaire.

Les emplacements et le numéro de port du troisième secondaire doivent être différents des trois autres serveurs. Il faut donc créer le fichier `/etc/sysconfig/pgsql/secondaire3` pour indiquer :

```
PGDATA=/var/lib/pgsql/10/secondaire3
PGPORT=5435
```

Il faut également mettre en place le script de démarrage :

```
# ln -s /etc/init.d/postgresql-10 /etc/init.d/secondaire3
```

Il est aussi nécessaire de modifier le numéro de port d'écoute de l'instance dans `/var/lib/pgsql/10/secondaire3/postgresql.conf` :

```
port = 5435
```

Le fichier de restauration doit être copié :

```
$ cp /var/lib/pgsql/10/secondaire1/recovery.conf
     /var/lib/pgsql/10/secondaire3/recovery.conf
```

Et légèrement modifié sur le nouveau secondaire :

```
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli password=repli
                    application_name=secondaire3'
trigger_file = '/tmp/secondaire3_autonome'
```

C'est dans cette dernière modification que tout se joue : notez que le numéro de port indique bien le premier secondaire et non pas le principal. Il faut également désactiver l'`archive_cleanup_command`.

Avant d'aller plus loin, il faut également s'assurer que les connections de préplication sont autorisées sur l'instance `secondaire1`. Le fichier `/var/lib/pgsql/10/secondaire1/pg_hba.conf` doit contenir la ligne suivante :

```
host  replication  repli  127.0.0.1/32  md5
```

Suite à la modification des paramètres de sécurité, la configuration du secondaire1 doit être rechargée :

```
$ psql -p 5433 -c "SELECT pg_reload_conf()"
 pg_reload_conf
----------------
 t
(1 ligne)
```

On peut enfin démarrer le troisième secondaire :

```
# service secondaire3 start
```

Le secondaire1 a bien un processus walreceiver et walsender :

```
-bash-4.1$ ps -o pid,cmd fx
  PID CMD
 2603 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire3
 2605  \_ postgres: logger process
 2606  \_ postgres: startup process   recovering 000000010000000000000068
 2608  \_ postgres: checkpointer process
 2609  \_ postgres: writer process
 2611  \_ postgres: stats collector process
 2612  \_ postgres: wal receiver process   streaming 0/68000060
 2406 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire1
 2408  \_ postgres: logger process
 2409  \_ postgres: startup process   recovering 000000010000000000000068
 2411  \_ postgres: checkpointer process
 2412  \_ postgres: writer process
 2413  \_ postgres: stats collector process
 2414  \_ postgres: wal receiver process   streaming 0/68000060
 2613  \_ postgres: wal sender process repli 127.0.0.1(36364) streaming 0/68000060
 2262 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire2
 2264  \_ postgres: logger process
 2265  \_ postgres: startup process   recovering 000000010000000000000068
 2267  \_ postgres: checkpointer process
 2268  \_ postgres: writer process
 2269  \_ postgres: stats collector process
 2270  \_ postgres: wal receiver process   streaming 0/68000060
  443 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
  445  \_ postgres: logger process
  447  \_ postgres: checkpointer process
  448  \_ postgres: writer process
```

```
  449  \_ postgres: wal writer process
  450  \_ postgres: autovacuum launcher process
  451  \_ postgres: archiver process   last was 000000010000000000000067.00000028.back
  452  \_ postgres: stats collector process
  453  \_ postgres: bgworker: logical replication launcher
 2271  \_ postgres: wal sender process repli 127.0.0.1(45956) streaming 0/68000060
 2415  \_ postgres: wal sender process repli 127.0.0.1(45958) streaming 0/68000060
```

Une connexion sur l'instance secondaire1 permettra de vérifier :

```
$ psql -p 5433 -c "\x" -c "SELECT * FROM pg_stat_replication;"
Affichage étendu activé.
-[ RECORD 1 ]----+------------------------------
pid              | 2613
usesysid         | 16388
usename          | repli
application_name | secondaire3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 36364
backend_start    | 2017-09-07 06:33:48.669222-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/68000140
write_lsn        | 0/68000140
flush_lsn        | 0/68000140
replay_lsn       | 0/68000140
write_lag        | 00:00:00.000479
flush_lag        | 00:00:00.015034
replay_lag       | 00:00:00.015219
sync_priority    | 0
sync_state       | async
```

4 HOT STANDBY : FAILOVER ET FAILBACK

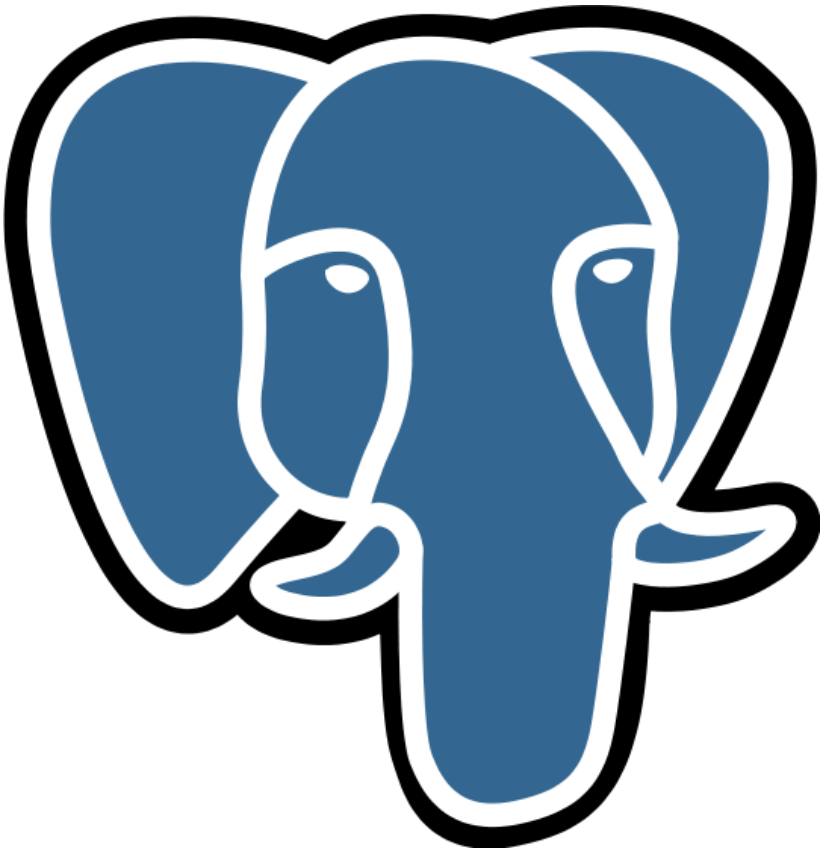


FIGURE 4: POSTGRESQL

4.1 INTRODUCTION

- Bascule par promotion d'un serveur *standby*
- Procédure de reconnexion au nouveau serveur principal
- Automatisation
- Points à superviser

17.12

La réplication interne de PostgreSQL permet de garantir la continuité de service, grâce à un ou plusieurs serveurs en attente. Ce n'est cependant pas le rôle de PostgreSQL de surveiller l'état des différents serveurs et de choisir s'il faut effectuer une bascule. Ce travail est laissé à la discrétion de l'administrateur ou d'un service de haute-disponibilité tiers qui peut être intégré à la plate-forme.

4.1.1 AU MENU

- Pré-requis et configuration
- Bascules
  - programmées (*switch over*)
  - d'urgence (*fail over*)
- Retour à l'état stable (*fail back*)
- Automatisation de ces opérations
- Supervision

Ce module a pour but de présenter le déclenchement d'une bascule du service en écriture sur un serveur standby et la reconnexion des autres serveurs standby à ce nouveau serveur principal. Nous allons commencer par donner un ensemble de conseils et prérequis utiles pour faciliter la reprise sur panne. Ensuite, nous verrons comment déclencher une bascule avec les actions effectuées par le serveur standby pour devenir serveur principal. Puis nous proposerons une configuration du logiciel de haute-disponibilité Pacemaker pour automatiser la bascule. Enfin, nous aborderons les points de supervision indispensables au suivi d'un ensemble de serveurs PostgreSQL en haute-disponibilité.

4.2 PRÉ-REQUIS ET ARCHITECTURE

- Configuration du système
- Configuration de PostgreSQL
- Stockage des journaux de transactions archivés

La création d'un service PostgreSQL hautement disponible nécessite d'effectuer des choix d'architecture qui ont un impact sur la configuration du système et de PostgreSQL. Ce chapitre décrit les points à considérer pour permettre une administration aisée de la plate-forme et l'impact des choix de configuration.

4.2.1 CONFIGURATION DU SYSTÈME

- Choisir un matériel de puissance équivalente
  - paramétrage mémoire et disque de PostgreSQL unique
  - perte de capacité en mode dégradé
- Points de défaillance possibles
  - baies, alimentation électrique, onduleur, routeurs, switchs...
- Avoir une ou plusieurs adresses IP dédiées au service pour éviter la reconfiguration des applications après bascule

Le choix du matériel à utiliser pour une architecture de haute disponibilité doit être basé sur la qualité de service minimum souhaitée lorsque le service est en mode dégradé. C'est pourquoi il est préconisé de choisir des machines de puissance équivalente pour chacun des nœuds. Comme la configuration de PostgreSQL est très liée à la mémoire et aux disques disponibles sur le système qui l'héberge, avoir des machines identiques permet d'unifier la configuration.

Il est également très important de considérer les différents points de défaillance au niveau de l'infrastructure physique pour garantir la disponibilité maximale du service. Ces points de défaillance ne sont pas spécifiques à PostgreSQL, mais à tout service qu'on souhaite rendre hautement disponible. Notamment, dans un système hautement disponible, une attention particulière doit être portée sur la fiabilité du réseau entre les différents nœuds.

Enfin, en cas de défaillance d'un serveur PostgreSQL, les applications et autres standby doivent pouvoir se connecter à un serveur de remplacement sans intervention manuelle sur leur configuration. Pour cela, le point d'entrée du service doit être défini sur une ou plusieurs adresses IP indépendantes des nœuds du cluster.

4.2.2 CONFIGURATION DE POSTGRESQL

- Unifier la configuration entre les nœuds :
  - matériel identique <=> paramètres mémoire et disque identiques
  - configurer les paramètres relatifs à la réplication sur tous les nœuds
  - créer un fichier modèle pour `recovery.conf` sur le principal
- S'assurer que `pg_hba.conf` ne bloque pas les clients en cas de bascule
- Propager les modifications de la configuration

Dans l'optique de la continuité du service et la facilité d'administration, il est indispensable de configurer PostgreSQL pour éviter le besoin de modifier la configuration en cas de bascule. Cela permet de conserver un plan de bascule le plus simple et succinct possible,

évitant ainsi de multiplier le nombre d' erreurs possibles lors de ces opérations souvent délicates.

Comme abordé précédemment, le fait d'avoir des machines identiques pour chaque nœud du cluster permet d'avoir un paramétrage unique pour la configuration mémoire et disque de PostgreSQL. De plus, il est tout à fait possible de configurer des paramètres relatifs au standby sur le serveur principal et inversement. Selon le mode de fonctionnement, principal ou standby, le serveur ignore les paramètres dont il n'a pas besoin. Ce comportement prend toute sa valeur par rapport aux paramètres dont la modification nécessite un redémarrage du moteur : on peut activer l'archivage sur un serveur standby, celui-ci archive alors les journaux de transactions dès qu'il est promu serveur principal.

Ensuite, il est indispensable d'appliquer ces principes à la configuration des accès, par l'intermédiaire d'un fichier `pg_hba.conf` commun à tous les nœuds de la plate-forme. On s'assure alors que les clients pourront accéder au serveur standby promu serveur principal sans intervention de l'administrateur. Ne pas oublier le fichier `pg_ident.conf` s'il est utilisé.

Chaque serveur standby étant créé à partir du serveur principal auquel on souhaite se connecter en *streaming replication*, il est recommandé de préparer un fichier modèle pour facilement créer le fichier `recovery.conf` sur le serveur principal. Un minimum de configuration est alors nécessaire pour préparer la configuration du mode standby. Il est possible par exemple de créer un fichier `recovery.conf.sample` dans le répertoire `$PGDATA` de l'instance principale. Ce dernier sera alors naturellement présent à la création d'un nouveau serveur de standby, et ne nécessitera qu'un simple renommage sur ce dernier pour l'activer.

Il est fortement déconseillé de créer un fichier nommé précisément `recovery.conf` sur l'instance principale. Ce dernier serait lu et interprété à chaque démarrage de l'instance !

Enfin, chaque modification de la configuration doit être reportée sur chacun des nœuds, le cas échéant. Ceci permet de garantir qu'une bascule ne changera pas le comportement du service.

4.2.3 STOCKAGE DES JOURNAUX ARCHIVÉS

- Plusieurs politiques d'archivages possibles
- Le serveur principal doit toujours archiver les journaux de transactions :
  - un archivage en panne peut provoquer un arrêt de service
  - la reprise sur panne nécessite des opérations pouvant retarder le retour à l'activité

- Chaque standby doit « nettoyer » les archives lui étant destinées
- Un standby indisponible doit être rapidement exclu de l'archivage s'il bloque celui-ci

PostgreSQL écrit le résultat de chaque transaction dans son journal de transactions avant de rendre la main à la session au moment du COMMIT. Lorsque le serveur est répliqué, le changement de segment du journal de transactions provoque l'archivage du fichier complété. Si la commande définie dans `archive_command` retourne une erreur, le serveur principal conserve le fichier et ré-exécute la commande d'archivage indéfiniment jusqu'à sa réussite. Dans cette situation, le serveur principal conserve tous les journaux de transactions jusqu'à épuisement de l'espace disponible dans le répertoire `$PGDATA/pg_wal`. L'instance s'arrête alors car ne peut plus écrire de données de transactions. Voici un exemple de message d'erreur obtenu:

```
PANIC: could not write to file "pg_wal/xlogtemp.32743": No space left on device
```

De par sa souplesse de configuration, PostgreSQL s' adapte facilement à de multiple politiques d'archivage :

- archivage sur un disque local ou un *filer* où les standby viendront récupérer les archives ;
- archivage de l'instance principale vers ses standby ;
- rétention planifiée des archives (`crontab`, tâche planifiée...) ;
- suppression des archives par les standby ;
- politique commune avec les sauvegardes PITR.

Dans le cas d'un archivage direct sur les serveurs standby, un serveur standby indisponible peut alors bloquer le serveur principal. Il faut alors modifier la configuration de l'archivage pour exclure ce serveur standby indisponible afin d 'éviter toute saturation d'espace disque.

Pour un archivage en local ou sur un *filer*, il faut veiller à disposer d'espace disque à tout moment.

Pour ces différentes raisons, il est recommandé d'utiliser un script d' archivage dans le paramètre `archive_command`. Il est ainsi plus aisé de modifier ce script pour exclure un standby par exemple et il n'est alors pas nécessaire de faire relire son fichier de configuration à l'instance. Attention toutefois à porter une attention particulière au code retour de votre script.

Enfin, l'utilisation du paramètre `archive_cleanup_command` du fichier `recovery.conf` permet d'automatiser le processus de nettoyage depuis les standby. Ceux-ci ont alors la responsabilité de supprimer les fichiers WAL leur étant destiné et devenu inutile. Cette configuration est indispensable pour garantir la disponibilité du service.

Voici un exemple de politique utile à la fois à la réplication et à la sauvegarde PITR:

- les archives sont transférées vers un répertoire de sauvegarde dédié sur un *filer* ;
- un répertoire supplémentaire par standby ;
- des liens « en dur » (*hardlink*) de tous les répertoires vers les différentes archives dans le répertoire de sauvegarde ;
- les standby nettoient chacun leur répertoire via le paramètre `archive_cleanup_command` ;
- une purge des archives du répertoire de sauvegarde programmée (3 jours, 1 semaine ou 1 mois par exemple).

Grâce à cette politique et aux liens « en dur », une archive est naturellement supprimée une fois qu'elle n'a plus de lien existant, ni pour la sauvegarde PITR, ni par un des standby.

Jusqu'en version 9.6 le répertoire des journaux se nommait `pg_xlog`. Les répertoires, fonctions et outils utilisant `xlog` ont été renommés en `wal` en version 10.

4.3 BASCULES PROGRAMMÉES ET EN URGENCE

- Bascule du service en écriture
- Cas de bascule :
  - perte du serveur principal
  - perte d'un serveur standby asynchrone
  - perte d'un serveur standby synchrone

En cas de panne d'un des nœuds du cluster, il peut être nécessaire de basculer le service (accès en écriture) sur un serveur standby. Cette partie montre comment réaliser une telle bascule, les actions effectuées par PostgreSQL ainsi que les points à surveiller selon le rôle et la configuration du serveur en panne.

4.3.1 BASCULE DU SERVICE EN ÉCRITURE

- Promotion du serveur de standby :
  - paramètre `trigger_file` dans `recovery.conf`
  - méthode préférée : action `promote` de `pg_ctl`
- Déconnexion de la streaming replication (bascule programmée)
- Rejeu des dernières transactions en attente d'application
- Choix d'une nouvelle timeline du journal de transaction
- Ouverture aux écritures

Il existe plusieurs méthodes pour promouvoir un serveur PostgreSQL en mode standby. L'utilisation d'un fichier de déclenchement (*Trigger File*) à travers le paramètre `trigger_file` est disponible depuis la version 9.0. Le serveur de standby vérifie en permanence si ce fichier existe. Dès que ce dernier apparaît, l'instance est alors promue. Par mesure de sécurité il est préconisé d'utiliser un emplacement accessible uniquement aux administrateurs pour un tel fichier.

Depuis la version 9.1, l'action `promote` a été ajoutée à l'outil pg_ctl. Cette dernière est la plus appropriée des deux méthodes.

Une fois le serveur promu, il finit de rejouer les données de transaction en provenance du serveur principal en sa possession et se déconnecte de celui-ci s'il est configuré en streaming replication.

Ensuite, il choisit une nouvelle timeline pour son journal de transactions. La timeline est le premier numéro dans le nom du segment (fichier WAL). Enfin, il autorise les connexions en écriture (ou toutes les connexions s'il s'agit d'un *Warm Standby*).

Comme le serveur a potentiellement reçu des modifications différentes du serveur principal qu'il répliquait précédemment, il ne peut être reconnecté à ce serveur. Le choix d'une nouvelle timeline permet à PostgreSQL de rendre les journaux de transactions de ce nouveau serveur en écriture incompatibles avec son ancien serveur principal. De plus, créer des journaux de transactions avec un nom de fichier différent rend possible l'archivage depuis ce nouveau serveur en écriture sans perturber l'ancien. Il n'y a pas de fichiers en commun si l'espace d'archivage est partagé.

4.3.2 PERTE DU SERVEUR PRINCIPAL

- Promotion d'un serveur standby défini par :
  - option `trigger_file` dans `recovery.conf`
  - méthode préférée : action `promote` de `pg_ctl`
- Reconnexion des applications :
  - déplacer l'adresse IP du service
  - ou changer la configuration des applications
- Reconnexion des standby restants au nouveau principal par reconstruction

La perte du serveur principal rend le service indisponible pour les écritures ansi que les lectures si les serveurs standby ne sont pas configurés en *hot standby*. Il est alors indispensable de basculer le service sur un serveur de standby en déclenchant sa promotion qui le force à sortir du mode standby et à accepter les écritures.

La bascule n'est complète que si les clients sont reconnectés au nouveau serveur principal, ce qui est facilement possible en déplaçant l'adresse IP réservée au service sur le nouveau serveur. Les transactions en cours au moment de la panne sont perdues. Les applications doivent alors recommencer leur traitement une fois connecté au serveur de remplacement. La nature transactionnelle des opérations sur la base de données permet d'éviter les corruptions de données.

Enfin, il est indispensable de reconstruire les autres serveurs standby pour remettre en place la réplication et retrouver la haute disponibilité du service.

4.3.3 PERTE D'UN STANDBY ASYNCHRONE

- Sortie de la ferme de standby en lecture seule
  - ne plus archiver les journaux de transactions sur ce serveur
  - reconfigurer les applications pour ne plus utiliser ce serveur
  - suppression du slot de réplication en cas de trop longue indisponibilité
- Ajouter un nouveau serveur de remplacement ?
- Reconnexion du standby une fois disponible
  - selon la disponibilité des journaux archivés

La perte d'un serveur standby asynchrone a potentiellement moins d'impact que celle du serveur principal. Néanmoins, il faut ajouter un nouveau serveur de remplacement pour ce serveur en panne et s'assurer que les lectures équilibrées sur ce serveur n'ont plus lieu, par reconfiguration des applications.

De plus, selon la configuration de l'archivage, une telle situation provoque un risque de panne de l'archivage des journaux de transactions qui peut être propagé au serveur principal. C'est pourquoi une action de l'administrateur est indispensable pour isoler ce serveur et s'assurer que l'archivage est maîtrisé sur la plate-forme.

Par ailleurs, si les slots de réplication (>= 9.4) sont utilisés pour mémoriser depuis l'instance principale la position de la réplication sur le standby arrêté, cela signifie que les journaux de transactions nécessaires à le reprise de la réplication seront conservés tant que l'instance secondaire sera indisponible. Cela peut conduire à un remplissage du système de fichiers de l'instance principale, et donc à une interruption de service. Si l'arrêt du standby est trop longue ou définitive, il convient donc de supprimer le slot de réplication associé sur l'instance principale, à l'aide de la fonction `pg_drop_replication_slot()`.

Enfin, dans le cas d'une maintenance programmée, toute opération qui provoquerait une rupture de la chaîne des journaux archivés impose une reconstruction de ce serveur standby pour permettre sa reconnexion.

4.3.4 PERTE D'UN STANDBY SYNCHRONE

- Passage automatique sur un autre serveur standby synchrone en attente
- Ajout d'un nouveau standby synchrone de remplacement
- Cas d'un standby synchrone unique
  - les opérations en écriture sur le principal sont bloquées
  - passer `synchronous_commit` à `local`
  - recharger la configuration du serveur principal
  - annuler les transactions synchrones

La réplication synchrone impose que les données de transaction soient écrites sur un stockage permanent sur le premier serveur standby synchrone de la liste définie par le paramètre `synchronous_standby_names`. Dans le cas de la perte de l'ensemble des serveurs standby synchrones, les transactions synchrones réalisées sur le serveur principal restent bloquées au moment du COMMIT car le serveur principal attend le retour d'un serveur standby synchrone et de sa confirmation pour rendre la main au client.

Ce comportement impose donc de disposer d'au moins un serveur standby synchrone supplémentaire par mesure de sécurité. On a ainsi besoin d'un cluster à trois nœuds pour éviter les blocages avec la réplication synchrone.

Pour débloquer une telle situation, il faut changer la valeur du paramètre `synchronous_commit` de `on` à `local`, recharger la configuration de PostgreSQL sur le serveur principal et envoyer un signal d'annulation des transactions en attente de la réplication synchrone. Cette annulation ne va pas annuler la transaction en elle même, mais simplement la réplication synchrone. Les transactions seront alors validée et durable **que** sur le serveur principal. Il est conseillé d'envoyer un tel signal en utilisant la fonction `pg_cancel_backend(pid integer)` plutôt qu'une commande `kill -TERM $PID`.

Dans tous les cas de blocage dus à l'indisponibilité d'un serveur standby synchrone, le synchronisme n'est plus possible, les opérations de déblocage ont d'ailleurs pour effet de la désactiver.

4.4 RETOUR À L'ÉTAT STABLE

- Si un standby a été promu avant la version 9.3 :
  - reconstruire les esclaves à partir d'une sauvegarde des fichiers du nouveau serveur principal

  - procédure la plus sûre pour tout serveur à connecter en standby
- À partir de la 9.3 : synchronisation automatique une fois la connexion rétablie
  - reconstruction obligatoire si l'esclave était plus avancé que le serveur promu

La phase de reconstruction des standby peut être optimisée en utilisant des outils de synchronisation de fichiers tels que `rsync` pour réduire le volume de données à transférer. L'utilisation de `pg_basebackup` n'est pas recommandée car cet outil impose une copie de l'ensemble des données du serveur principal.

Le fait de disposer de l'ensemble des fichiers de configuration sur tous les nœuds permet de gagner un temps précieux lors des phases de reconstruction, qui peuvent également être scriptées. Par contre, les opérations de reconstructions se doivent d'être lancées manuellement pour éviter tout risque de corruption de données dues à des opérations automatiques externes, comme lors de l'utilisation de solutions de haute disponibilité.

Enfin, on rappelle qu'il ne faut pas oublier de prendre en compte les *tablespaces* lors de la reconstruction.

4.5 RETOUR À L'ÉTAT STABLE, SUITE

- Si un standby a été momentanément indisponible, reconnexion directe possible si
  - les journaux de transactions nécessaires sont encore présents sur l'instance principale
  - les journaux de transactions nécessaires sont présents en archives
  - dans les autres cas, reconstruction nécessaire

4.6 AUTOMATISATION

- Tâches à automatiser avec des scripts
- Outils disponibles
  - pacemaker
  - pgpool
  - repmgr
  - monit

La mise en réplication, initiale ou après une bascule, nécessite de suivre une procédure bien déterminée. Pour faciliter la tâche d'administration, il est fortement recommandé d'automatiser certaines étapes avec des scripts. Cela permet d'être plus réactif en cas

de problème et de réduire le risque d'erreur dans l'exécution de la procédure : les situations de bascule et remise en réplication se produisent généralement sous la pression d'un service interrompu avec des utilisateurs bloqués.

Cette partie montre comment automatiser au mieux ces actions et les outils permettant cette automatisation.

4.6.1 TÂCHES À AUTOMATISER AVEC DES SCRIPTS

- Base backup
  - démarrage et arrêt de la sauvegarde des fichiers
  - utilisation de `rsync`
  - prise en compte des *tablespaces*
- Configuration de `recovery.conf`
  - partir d'un fichier modèle

À partir de la version 9.1, on dispose de l'outil `pg_basebackup` pour créer un serveur standby. On peut alors se poser la question du besoin de créer un script personnalisé pour créer manuellement un serveur standby. Ce travail est pertinent pour deux raisons :

- `pg_basebackup` s'attend à créer un standby de zéro. Il impose que le répertoire de données de destination soit vide (ainsi que les répertoires des *tablespaces*) ;
- selon les volumes de données mis en jeu, et encore plus avec une liaison instable, il est souvent plus intéressant d'utiliser `rsync` pour la copie des fichiers en synchronisant des données existantes ou partiellement copiées sur le serveur standby à créer. En effet, `rsync` ne transfère que les fichiers ayant subi une modification. En voici un exemple d'utilisation :

```
rsync -av -e 'ssh -o Compression=no' --whole-file --ignore-errors \
      --delete-before --exclude 'lost+found' --exclude 'pg_wal/*' \
      --exclude='*.pid' $MASTER:$PGDATA/ $PGDATA/
```

Noter que l'on utilise `--whole-file` par précaution pour forcer le transfert entier d'un fichier de données en cas de détection d'une modification. C'est une précaution contre tout risque de corruption (`--inplace` ne transférerait que les blocs modifiés). Les grosses tables sont fractionnées en fichiers de 1 Go donc elles ne seront pas intégralement retransférées.

De plus, la création du fichier `recovery.conf` est laissée à la discrétion de l'administrateur par `pg_basebackup` (cela étant dit, la version 9.3 a ajouté cette fonctionnalité à `pg_basebackup`). L'utilisation d'un script personnalisé qui se base sur un fichier modèle est envisageable pour automatiser la création du serveur standby.

17.12

Pour créer le script le plus simple possible, voici quelques conseils :

- s'assurer que le répertoire d'archivage soit accessible depuis toutes les machines, pour ne pas avoir à modifier le fichier `postgresql.conf` lors de la procédure ;
- créer dans le répertoire `$PGDATA` du serveur principal un fichier `recovery.conf.machine` où `machine` est le nom du serveur standby avec les paramètres qui conviennent, y compris pour la machine principale dans le cas où elle serait amenée à prendre le rôle de serveur standby... on peut alors créer la configuration avec une simple copie ;
- la liste des répertoires des *tablespaces* se trouve dans `$PGDATA/pg_tblspc` sous forme de liens symboliques pointant sur le répertoire du *tablespace* : on peut alors rendre le script générique pour la synchronisation des *tablespaces*.

4.6.2 HAUTE-DISPONIBILITÉ

- L'objectif est de minimiser le temps d'interruption du service en cas d'avarie
- Un peu de vocabulaire :
  - *SPOF* : *Single Point Of Failure*
  - *Redondance* : pour éviter les SPOF
  - *Ressources* : éléments gérés par un cluster
  - *Fencing* et *STONITH* (*Shoot The Other Node In The Head*) : protection contre les corruptions

Pour minimiser le temps d'interruption d'un service, il faut implémenter les éléments nécessaires à la tolérance de panne en se posant la question : que faire si le service n'est plus disponible ?

Une possibilité s'offre naturellement : on prévoit une machine prête à prendre le relai en cas de problème. Lorsqu'on réfléchit aux détails de la mise en place de cette solution, il faut considérer :

- les causes possibles de panne ;
- les actions à prendre face à une panne déterminée et leur automatisation ;
- les situations pouvant corrompre les données.

Les éléments de la plate-forme, aussi matériels que logiciels, dont la défaillance mène à l'indisponibilité du service, sont appelés *SPOF* dans le jargon de la haute-disponibilité, ce qui signifie *Single Point Of Failure* ou point individuel de défaillance. L'objectif de la mise en haute-disponibilité est d'éliminer ces SPOF. Ce travail concerne le service dans sa globalité : si le serveur applicatif n'est pas hautement-disponible alors que la base de

données l'est, le service peut être interrompu car le serveur applicatif est un SPOF. On élimine les différents SPOF possibles par la *redondance*, à la fois matérielle et logicielle.

Par la suite, on discutera de la complexité induite par la mise en haute-disponibilité. En effet, dans le cas d'une base de données, éviter les corruptions lors d'évènements sur le cluster est primordial.

On peut se trouver dans une situation où les deux machines considèrent qu'elles sont seules chacune dans leur cluster (*Split-Brain*) ou entrer en concurrence pour un accès en écriture sur un même disque partagé. Dans ce cas, on utilise des concepts comme le *Quorum*, qui permet de ne pas accéder aux ressources tant que le cluster ne se trouve pas dans un état cohérent.

Enfin, pour prévenir l'accès concurrent à des ressources qui ne le permettent pas, on utilise le *Fencing*, qui permet d'isoler une machine du cluster (*Nœud*) en lui interdisant l'accès : la technique la plus connue est le *STONITH* (*Shoot The Other Node In The Head*) pour éteindre une machine qui n'aurait pas dû tenter d'accéder à une ressource.

4.6.3 PRÉSENTATION DE PACEMAKER

- Solution de Haute-Disponibilité généraliste
- Se base sur Corosync, un service de messagerie inter-nœuds
- Permet de surveiller la disponibilité des machines
- Gère le quorum et les cas STONITH
- Gère les ressources d'un cluster et leur interdépendance

Pacemaker associe la surveillance de la disponibilité de machines et d' applications. Il offre l'outillage nécessaire pour effectuer les actions suite à une panne. Il s'agit d'une solution de haute-disponibilité extensible avec des scripts.

4.6.4 PRÉSENTATION DE PAF

- **P**ostgres **A**utomatic **F**ailover
- Ressource Agent pour Pacemaker et Corosync permettant de :
  - Détecter un incident
  - Relancer l'instance maître
  - Basculer sur un autre nœud en cas d'échec de relance
  - Élire le meilleur esclave (avec le retard le plus faible)
  - Basculer les rôles au sein du cluster en maître et esclave

- Avec les fonctionnalités offertes par Pacemaker & Corosync :
  - Surveillance de la disponibilité du service
  - Quorum & Fencing
  - Gestion des ressources du cluster

PAF est le fruit de la R&D de Dalibo visant à combler les lacunes des agents existant. Il s'agit d'un produit opensource, disponible sur ce dépôt github[22]

4.6.5 PRÉSENTATION DE PGPOOL

- Outil spécialisé PostgreSQL
- Propose l'exécution de scripts de bascule automatique en cas de problème
  - *health check*
  - *failover* automatique

pgpool est un outil bien connu dans la communauté PostgreSQL, notamment par ses fonctionnalités de pooling de connexion et de réplication. Il sait aussi faire de la répartition de charge, et dispose d'une configuration lui permettant de faire des bascules.

En effet, il dispose d'un processus de vérification de disponibilité du maître. Si ce dernier devient indisponible, il peut exécuter un script qui réalisera le failover. Voir les paramètres `health_check_*` ainsi que les paramètres `failover_command`, `failback_command`, `follow_master_command`, `fail_over_on_backend_error` dans la documentation de pgpool[23] .

4.6.6 PRÉSENTATION DE REPMGR

- Outil spécialisé PostgreSQL
- Gère automatiquement la bascule en cas de problème
  - Health check
  - Failover et switchover automatiques

L'outil repmgr[24] permet la gestion de la haute disponibilité avec notamment la gestion des opérations de clonage, de promotion d'une instance en primaire et la démotion d'une instance.

[22] https://github.com/dalibo/PAF/
[23] http://www.pgpool.net/docs/latest/pgpool-en.html
[24] http://www.repmgr.org/

L'outil repmgr peut également être en charge de la promotion automatique du nœud secondaire en cas de panne du nœud primaire, cela implique la mise en place d'un serveur supplémentaire par cluster HA (paire primaire/secondaire) appelée témoin. Cette machine hébergera une instance PostgreSQL dédiée au processus daemon `repmgrd`, processus responsable d'effectuer les contrôles d'accès réguliers à l'instance primaire et de promouvoir le nœud secondaire lorsque une panne est détectée et confirmée suite à plusieurs tentatives échouées consécutives.

Afin de faciliter la bascule du trafic sur l'instance secondaire en cas de panne du primaire, l'utilisation d'une adresse IP virtuelle (VIP) est requise. Les applications clientes (hors administration) doivent se connecter directement à la VIP.

4.6.7 PRÉSENTATION DE MONIT

- Outil de supervision des processus
- Détection de défaillance et exécution d'action selon le contexte

Monit est un outil de surveillance et de gestion de processus, capable de détecter une défaillance et d'exécuter des actions pertinentes en fonction des erreurs et des différentes situations.

On peut par exemple utiliser monit pour surveiller les processus PostgreSQL. Si monit s'aperçoit que le maître n'est plus disponible, il peut exécuter un script qui fera une bascule automatique d'un des esclaves en maître, et qui s'assurera de la bascule de l'adresse IP.

Voir le site officiel[25] .

4.7 SUIVI ET SUPERVISION

- Archivage
- État du serveur
- Gestion des conflits
- Retard de réplication

Que les bascules soient manuelles ou automatisées, il faut bien contrôler le serveur et surtout son état.

---

[25] http://mmonit.com

4.7.1 ÉTAT DE L'ARCHIVAGE

- Processus d'archivage
  - indique la réussite ou l'échec du dernier archivage
  - vue `pg_stat_archiver` (9.4)
- Fichiers en attente d'archivage
  - `pg_wal/archive_status/*.ready`
- Traces applicatives
  - message en cas d'échec de la commande d'archivage

L'état de l'archivage peut se contrôler de plusieurs façons.

En étant connecté au serveur, la commande `ps` donne déjà quelques informations :

- en cas de réussite, le nom du dernier journal archivé ;
- en cas d'échec, le fait que la commande a échoué.

Depuis la version 9.4, la vue `pg_stat_archiver` permet de vérifier l'état du processus d'archivage sans passer par le système.

Le contenu du répertoire `pg_wal/archive_status` est généralement plus intéressant. PostgreSQL crée un fichier par journal à archiver. Le nom du fichier correspondant au nom du journal de transactions avec une extension `.ready`. En comptant le nombre de fichiers, on peut se rendre compte d'un problème : si ce nombre ne fait qu'augmenter, il y a un soucis. S'il reste stable (généralement à 0), il n'y a pas de problème. Dans le même esprit, il est possible de surveiller la place prise par le répertoire `pg_wal`. Cela étant dit, ce type de vérification risque de provoquer des faux positifs, un batch d'insertion pouvant avoir le même résultat.

4.7.2 CONNAÎTRE L'ÉTAT DU SERVEUR

- Une fonction de statut
  - `pg_is_in_recovery()`
  - pour connaître l'état d'un serveur
- Une vue système
  - `pg_stat_replication`
  - pour connaître le type de réplication, le retard des esclaves
  - en 9.3, fonction `pg_wal_lsn_diff`
  - en 10, `write_lag`, `flush_lag`, `replay_lag`

Étant donné qu'il est maintenant possible de se connecter sur le maître comme sur l'esclave, il est important de pouvoir savoir sur quel type de serveur un utilisateur est connecté. Pour cela, il existe une procédure stockée appelée `pg_is_in_recovery()`. Elle renvoie la valeur `true` si l'utilisateur se trouve sur un serveur en *hot standby* et `false` sinon.

Depuis la version 9.1, pour connaître l'état des différents serveurs esclaves connectés au serveur maître, le plus simple est de récupérer les informations provenant de la vue `pg_stat_replication`. Elle n'est renseignée que sur le maître et permet de connaître l'état de tous les esclaves connectés. Voici un exemple du contenu de cette table :

```
postgres=# SELECT * FROM pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
pid              | 4189
usesysid         | 16388
usename          | repli
application_name | secondaire1
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 46324
backend_start    | 2017-09-07 11:22:50.262842-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/73D3C2F0
write_lsn        | 0/73D3C2F0
flush_lsn        | 0/73D3C2F0
replay_lsn       | 0/73D3C2F0
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

Le calcul de la différence de volumétrie de données entre le maître et ses esclaves est toujours à faire mais il n'est plus nécessaire de se connecter sur les deux serveurs. Depuis la version 9.3, le calcul de différentiel peut-être effectué avec la fonction `pg_wal_lsn_diff`:

```
# SELECT pg_wal_lsn_diff(pg_current_wal_lsn(), '0/73D3C1F0');
-[ RECORD 1 ]---+----
pg_wal_lsn_diff | 256
```

La plupart des colonnes se comprennent aisément. La colonne `state` indique l'état du flux de réplication : `streaming` indique que le flux est fonctionnel, `catchup` indique que le serveur esclave essaie de rattraper son retard, etc. `sync_state` peut avoir trois valeurs : `async` dans le cas d'une réplication asynchrone, `sync` et `potential` dans le cas d'une réplication synchrone.

17.12

Le retard d'un esclave sur son maître est toujours exprimé en octets, ce qui n'est pas simple à appréhender si on veut savoir si un esclave est trop en retard par rapport au maître. La version 9.1 propose une nouvelle procédure stockée, appelée `pg_last_xact_replay_timestamp()`, indiquant la date et l'heure de la dernière transaction rejouée. Soustraire la date et l'heure actuelle à cette fonction permet d'avoir une estimation sur le retard d'un esclave sous la forme d'une durée. Attention, cela ne fonctionne que si le maître est actif. En effet, si aucune requête ne s'exécute sur le maître, le calcul `now()-pg_last_xact_replay_timestamp()` aura pour résultat une durée croissante dans le temps, même si l'esclave est identique au maître (un peu comme Slony et son pseudo lag quand les événements de synchronisation ne passent plus).

La version 10 introduit 3 nouvelles colonnes pour permettre de suivre le retard d'une réplication synchrone plus aisément :

```
postgres=# SELECT write_lag,flush_lag,replay_lag FROM pg_stat_replication;
-[ RECORD 1 ]---------------
write_lag  | 00:00:00.013782
flush_lag  | 00:00:00.049256
replay_lag | 00:00:00.097876
```

`write_lag` mesure le délai en cas de `synchronous_commit = remote_write`. Cette configuration fera que chaque `COMMIT` attendra la confirmation de la réception en mémoire de l'enregistrement du `COMMIT` par le standby et son écriture via le système d'exploitation, sans que les données du cache du système ne soient vidées sur disque au niveau du serveur standby.

`flush_lag` mesure le délai jusqu'à confirmation que les données modifiées soient bien écrites sur disque au niveau du serveur standby.

`replay_lag` mesure le délai en cas de `synchronous_commit = remote_apply`. Cette configuration fera en sorte que chaque `COMMIT` devra attendre le retour des standbys synchrones actuels indiquant qu'ils ont bien rejoué la transaction, la rendant visible aux requêtes des utilisateurs.

4.7.3 GESTION DES CONFLITS

- Une requête en lecture pose des verrous
- Un conflit peut avoir lieu entre ces verrous et l'application des changements pour la réplication
- Quatre paramètres
  - `max_standby_archive_delay`

- max_standby_streaming_delay
- vacuum_defer_cleanup_age
- hot_standby_feedback

Un conflit peut survenir entre l'application des modifications de la réplication et la connexion sur une base d'un serveur esclave ou sur l'exécution d'une requête en lecture seule. Comme les modifications de la réplication doivent s'enregistrer dans l'ordre de leur émission, si une requête bloque l'application d'une modification, elle bloque en fait l'application de toutes les modifications suivantes pour cet esclave. Un exemple simple de conflit est l'exécution d'une requête sur une base que la réplication veut supprimer. PostgreSQL attend un peu avant de forcer l'application des modifications. S'il doit forcer, il sera contraint d'annuler les requêtes en cours, voire de déconnecter les utilisateurs. Évidemment, cela ne concerne que les requêtes et/ou les utilisateurs gênants.

La table du catalogue pg_stat_conflicts n'est renseignée que sur les esclaves d'une réplication. Il contient le nombre de conflits détectés sur cet esclave par type de conflit (conflit sur un *tablespace*, conflit sur un verrou, etc.). Elle contient une ligne par base de données. Il est à noter que pg_stat_database contient une nouvelle colonne contenant le décompte total des conflits. Cela nous donne ce résultat :

```
postgres=# SELECT * FROM pg_stat_database_conflicts
postgres-# WHERE datname='postgres' ;
-[ RECORD 1 ]----+---------
datid            | 12857
datname          | postgres
confl_tablespace | 0
confl_lock       | 0
confl_snapshot   | 3
confl_bufferpin  | 2
confl_deadlock   | 0
```

Les informations disponibles dans ce catalogue permettent aux administrateurs de mieux configurer les paramètres vacuum_defer_cleanup_age, max_standby_archive_delay et max_standby_streaming_delay.

Concernant les conflits, il est aussi à savoir que les esclaves peuvent maintenant envoyer des informations au maître sur les requêtes en cours d'exécution pour tenter de prévenir les conflits de requêtes lors du nettoyage des enregistrements (action effectuée par le VACUUM). Il faut pour cela activer le paramètre hot_standby_feedback. L'esclave envoie des informations au maître à une certaine fréquence, configurée par le paramètre wal_receiver_status_interval. Il va sans dire que ces deux paramètres doivent être maniés avec précaution, notamment si les tables du serveur ont tendance à se fragmenter facilement. L'inconvénient est que cela peut causer une fragmentation des tables plus importantes sur le maître. Cette fragmentation cependant ne sera pas plus importante que

si les requêtes exécutées sur l'esclave l'avaient été sur le maître lui même.

Grâce à cet envoi d'informations, PostgreSQL peut savoir si un esclave est indisponible, par exemple suite à une coupure réseau ou à un arrêt brutal de l'esclave. Si jamais l'esclave est indisponible pendant un temps déterminé par le paramètre `replication_timeout` (>= 9.1), il coupe la connexion avec l'esclave. Pour éviter des coupures intempestives, il faut donc configurer `wal_receiver_status_interval` avec une valeur inférieure à celle de `replication_timeout`.

4.7.4 CONTRÔLE DE LA RÉPLICATION

- Deux fonctions de contrôle
  - `pg_wal_replay_pause()` pour mettre en pause
  - `pg_wal_replay_resume()` pour reprendre
- Une fonction de statut
  - `pg_is_wal_replay_paused()`

Lancer un `pg_dump` sur un serveur esclave n'est pas simple à cause des risques d'annulation de requêtes en cas de conflits. L'exécution d'un `pg_dump` peut durer très longtemps et ce dernier travaille en exécutant des requêtes, parfois très longues (ie. `COPY`) et donc facilement annulées même après configuration des paramètres `max_standby_*_delay`. Il faut donc pouvoir mettre en pause l'application de la réplication. La version 9.1 dispose de trois fonctions intéressantes dans ce cadre :

- `pg_wal_replay_pause()`, pour mettre en pause la réplication sur l'esclave où est exécutée cette commande ;
- `pg_wal_replay_resume()`, pour relancer la réplication sur un esclave où la réplication avait été précédemment mise en pause ;
- `pg_is_wal_replay_paused()`, pour savoir si la réplication est en pause sur l'esclave où est exécutée cette commande.

Ces fonctions s'exécutent uniquement sur les esclaves et la réplication n'est en pause que sur l'esclave où la fonction est exécutée. Il est donc possible de laisser la réplication en exécution sur certains esclaves et de la mettre en pause sur d'autres.

4.8 CONCLUSION

- Procédures de bascule et retour à l'état stable simples et documentées

- Possibilité d'automatiser les bascules avec des solutions de clustering tierces

---

4.8.1 QUESTIONS

N'hésitez pas, c'est le moment !

4.9 TRAVAUX PRATIQUES

4.9.1 ÉNONCÉS

4.10 PRÉ-REQUIS

Dans le cadre de ce TP vous allez utiliser les instances principales et secondaires du précédent module (*W2 - Hot Standby, Installation et paramétrage*).

Normalement vous avez une instance principale et au moins 2 instances secondaires :

```
$ ps -o pid,cmd fx
  PID CMD
 4382 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire2
 4384  \_ postgres: logger process
 4385  \_ postgres: startup process   recovering 000000010000000000000077
 4387  \_ postgres: checkpointer process
 4388  \_ postgres: writer process
 4390  \_ postgres: stats collector process
 4391  \_ postgres: wal receiver process   streaming 0/77000060
 4180 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire1
 4182  \_ postgres: logger process
 4183  \_ postgres: startup process   recovering 000000010000000000000077
 4185  \_ postgres: checkpointer process
 4186  \_ postgres: writer process
 4187  \_ postgres: stats collector process
 4188  \_ postgres: wal receiver process   streaming 0/77000060
 3736 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
 3738  \_ postgres: logger process
```

```
3740  \_ postgres: checkpointer process
3741  \_ postgres: writer process
3742  \_ postgres: wal writer process
3743  \_ postgres: autovacuum launcher process
3744  \_ postgres: archiver process   last was 000000010000000000000076.00000028.back
3745  \_ postgres: stats collector process
3746  \_ postgres: bgworker: logical replication launcher
4189  \_ postgres: wal sender process repli 127.0.0.1(46324) streaming 0/77000060
4392  \_ postgres: wal sender process repli 127.0.0.1(46340) streaming 0/77000060
```

Afin de pouvoir utiliser pg_rewind à la fin de ce TP il est nécessaire d'activer la journalisation des *hint bits* grâce à l'option `wal_log_hints = on` dans le fichier `postgresql.conf`. Cette modification n'aurait pas été nécessaire si l'instance primaire avait été créée avec l'option `--data-checksums` à sa création avec `initdb`.

4.11 PROMOTION D'UN SECONDAIRE

Arrêter le principal et effectuer la promotion du secondaire1. Observez les traces de l'instance.

4.12 SUIVI DE TIMELINE

À partir de quelle version de PostgreSQL est-il possible de faire un suivi de timeline ?

Configurez le secondaire2 pour qu'il se connecte au secondaire1. Observez les traces de l'instance.

4.13 SWITCHOVER DU PRINCIPAL

- Si votre serveur principal a bien été arrêté avant la bascule vous allez pouvoir le « raccrocher » au secondaire1.
- Si celui-ci était toujours actif après la promotion du secondaire1 il y aurait des transactions supplémentaires par rapport au secondaire1. Cette situation nécessite soit de reconstruire l'instance soit d'utiliser l'outil `pg_rewind` (disponible à partir de la version 9.5).

Choisissez une des deux techniques afin de retrouver une situation avec deux secondaires.

4.13.1 SOLUTIONS

Pré-requis

Pour utiliser `pg_rewind`, il faut s'assurer que l'instance a été créée avec l'option `--data-checksums`. Dans le cas contraire, il faut activer `wal_log_hints = on` dans le fichier postgresql.conf. Pour ce faire, nous allons utiliser l'outil `pg_controldata` qui fournit des informations sur l'instance :

```
$ /usr/pgsql-10/bin/pg_controldata \
  -D /var/lib/pgsql/10/data/ | grep checksum
Data page checksum version:           0
```

Par défaut, l'instance est initialisée sans checksums. Nous devons donc configurer le paramètre `wal_log_hints` à la valeur `on`. Lançons un checkpoint afin de s'assurer que les nouveaux journaux contiendrons les *hints bits* :

```
$ psql -c "CHECKPOINT"
```

Promotion d'un secondaire

> Arrêter le principal et effectuer la promotion du secondaire1. Observez les traces de l'instance.

Pour effectuer la promotion, nous allons utiliser l'action `promote` de l'outil `pg_ctl` ou du script de démarrage. Il est également possible d'utiliser le *trigger file* à condition que l'option a bien été définie dans le fichier `recovery.conf`.

```
# service postgresql-10 stop
# service secondaire1 promote
```

Dans les traces, nous trouvons ceci :

```
2017-09-08 04:12:03.087 EDT [472] LOG:  received promote request
2017-09-08 04:12:03.087 EDT [472] LOG:  redo done at 0/7A000028
cp: cannot stat `/var/lib/pgsql/10/archives/000000010000000000000007A': No such file or
cp: cannot stat `/var/lib/pgsql/10/archives/00000002.history': No such file or directo
2017-09-08 04:12:04.489 EDT [472] LOG:  selected new timeline ID: 2
cp: cannot stat `/var/lib/pgsql/10/archives/00000001.history': No such file or directo
2017-09-08 04:12:05.245 EDT [472] LOG:  archive recovery complete
```

```
2017-09-08 04:12:06.256 EDT [469] LOG:  database system is ready to accept connections
```

Le serveur a bien reçu la demande de promotion, ensuite il cherche à rejouer les derniers journaux de transaction depuis les archives. Puis il vérifie la présente d'un fichier `00000002.history` pour déterminer une nouvelle timeline.

Une fois ces opérations effectuées, il détermine la nouvelle timeline (ici, 2) et devient accessible en lecture/écriture :

```
$ psql -p 5433 b1
psql (10)
Type "help" for help.

b1=# CREATE TABLE t5 (c1 int);
CREATE TABLE
b1=# INSERT INTO t5 VALUES ('1');
INSERT 0 1
```

Remarque : Le fichier `recovery.conf` a été renommé en `recovery.done`.

On avait copié le fichier `postgresql.conf` de l'instance primaire. Celui-ci comprenait une `archive_command` qui archivait vers `/var/lib/pgsql/10/archives/`. Vu que le paramètre `archive_mode` était à `on`, seul le serveur primaire effectuait l'archivage. En effectuant la promotion de l'instance secondaire1, celui-ci est devenu primaire et archive donc vers le même répertoire !

Il y a bien un archiver process :

```
  469 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/secondaire1
  471  \_ postgres: logger process
  474  \_ postgres: checkpointer process
  475  \_ postgres: writer process
  476  \_ postgres: stats collector process
  699  \_ postgres: wal writer process
  700  \_ postgres: autovacuum launcher process
  701  \_ postgres: archiver process   last was 00000002.history
  702  \_ postgres: bgworker: logical replication launcher
```

Le répertoire d'archive contient :

```
$ ls -alth /var/lib/pgsql/10/archives/
total 209M
drwxr-xr-x 2 postgres postgres 4.0K Sep  8 04:12 .
-rw------- 1 postgres postgres   42 Sep  8 04:12 00000002.history
-rw------- 1 postgres postgres  16M Sep  8 04:12 000000010000000000000007A.partial
```

```
-rw------- 1 postgres postgres  16M Sep  8 04:11 000000010000000000000079
-rw------- 1 postgres postgres  16M Sep  8 04:11 000000010000000000000078
(...)
```

Le serveur a archivé le dernier journal avec le suffixe `.partial` [26].

Suivi de timeline

> À partir de quelle version de PostgreSQL est-il possible de faire un suivi de timeline ?

Le suivi de timeline n'est disponible qu'à partir de la version 9.3.

> Configurez le secondaire2 pour qu'il se connecte au secondaire1. Observez les traces de l'instance.

Nous avons donc :

- l'ancienne instance primaire arrêtée ;
- l'instance secondaire1 promue ;
- l'instance secondaire2 qui est toujours secondaire de l'instance primaire arrêtée ;
- l'instance secondaire3 (ancien secondaire en cascade) qui est secondaire de l'instance secondaire1 promue ;
- en option, l'instance secondaire4 en cascade avec l'instance secondaire2.

Nous allons tenter de raccrocher les instances secondaire2 et secondaire3.

Instance secondaire2

Dans les traces de secondaire2 :

```
cp: cannot stat `/var/lib/pgsql/10/archives/00000001000000000000007A':
    No such file or directory
2017-09-08 04:20:15.362 EDT [1105] FATAL: could not connect to the primary
                server: could not connect to server: Connection refused
                Is the server running on host "127.0.0.1" and accepting
                TCP/IP connections on port 5432?
```

[26] Ceci afin d'éviter d'écraser un fichier journal si le maître était toujours actif. Avant la version 9.5, il pouvait y avoir un conflit entre l'ancien primaire et le secondaire promu archivant au même emplacement. Le fichier `00000001000000000000007A` aurait pu être écrasé. Voir : Archiving of last segment on timeline after promotion http://paquier.xyz/postgresql-2/postgres-9-5-feature-highlight-partial-segment-timeline/)

Le serveur ne sait pas que l'instance secondaire1 a été promue, il constate juste que l'instance primaire a été arrêtée.

Modifions le paramètre `primary_conninfo` de l'instance secondaire2 pour pointer vers l'instance secondaire1 :

```
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli
                    password=repli application_name=secondaire2'
```

Après redémarrage de l'instance, on peut constater que la modification n'est pas suffisante :

```
2017-09-08 04:21:55.621 EDT [1232] LOG:  database system is ready to accept
                                         read only connections
2017-09-08 04:21:55.621 EDT [1235] LOG:  invalid record length at 0/7A000098:
                                         wanted 24, got 0
2017-09-08 04:21:55.672 EDT [1240] LOG:  fetching timeline history file for
                                         timeline 2 from primary server
2017-09-08 04:21:55.739 EDT [1240] FATAL:  could not start WAL streaming: ERROR:
      replication slot "slot_secondaire2" does not exist
      cp: cannot stat `/var/lib/pgsql/10/archives/00000001000000000000007A':
          No such file or directory
```

Les slots de réplication ne sont pas journalisés, il faut créer le slot ou le désactiver. Pour faire simple, nous allons juste le désactiver.

```
2017-09-08 05:44:39.116 EDT [1360] LOG:  entering standby mode
cp: cannot stat `/var/lib/pgsql/10/archives/00000001000000000000000A':
    No such file or directory
2017-09-08 05:44:39.164 EDT [1360] LOG:  consistent recovery state reached
                                         at 0/A000098
2017-09-08 05:44:39.164 EDT [1360] LOG:  invalid record length at 0/A000098:
                                         wanted 24, got 0
2017-09-08 05:44:39.165 EDT [1356] LOG:  database system is ready to accept
                                         read only connections
2017-09-08 05:44:39.205 EDT [1365] LOG:  started streaming WAL from primary at
                                         0/A000000 on timeline 1
2017-09-08 05:44:39.206 EDT [1365] LOG:  replication terminated by primary
                                         server
2017-09-08 05:44:39.206 EDT [1365] DETAIL:  End of WAL reached on timeline 1
                                             at 0/A000098.
```

Ce n'est toujours pas suffisant. En effet, notre instance est restée sur la timeline 1. Pour

indiquer à l'instance de suivre le changement de timeline, il faut spécifier le paramètre `recovery_target_timeline` dans le fichier recovery.conf :

```
recovery_target_timeline = 'latest'
```

Ceci nous donne :

```
2017-09-08 04:45:27.410 EDT [2840] LOG:  database system was shut down in
                                         recovery at 2017-09-08 04:45:27 EDT
2017-09-08 04:45:27.419 EDT [2840] LOG:  restored log file "00000002.history"
                                         from archive
cp: cannot stat `/var/lib/pgsql/10/archives/00000003.history':
    No such file or directory
2017-09-08 04:45:27.428 EDT [2840] LOG:  entering standby mode
2017-09-08 04:45:27.437 EDT [2840] LOG:  restored log file "00000002.history"
                                         from archive
2017-09-08 04:45:27.550 EDT [2840] LOG:  restored log file
                                         "000000020000000000000007A" from archive
2017-09-08 04:45:27.860 EDT [2840] LOG:  consistent recovery state reached at
                                         0/7A000098
2017-09-08 04:45:27.861 EDT [2837] LOG:  database system is ready to accept
                                         read only connections
2017-09-08 04:45:27.862 EDT [2840] LOG:  redo starts at 0/7A000098
2017-09-08 04:45:28.400 EDT [2840] LOG:  restored log file
                                         "000000020000000000000007B" from archive
cp: cannot stat `/var/lib/pgsql/10/archives/000000020000000000000007C':
    No such file or directory
2017-09-08 04:45:28.711 EDT [2840] LOG:  unexpected pageaddr 0/77000000 in log
                                         segment 000000020000000000000007C, offset 0
2017-09-08 04:45:28.725 EDT [2850] LOG:  started streaming WAL from primary at
                                         0/7C000000 on timeline 2
```

Cette fois, le serveur a bien suivi le changement de timeline. La table `t5` existe bien et contient bien les enregistrements :

```
$ psql -p 5434 b1 -c "SELECT * FROM t5;"
 c1
----
  1
(1 row)
```

On peut également le vérifier dans la vue `pg_stat_replication` de secondaire1 :

```
$ psql -p 5433 -c "\x" -c "SELECT * FROM pg_stat_replication ;"
Expanded display is on.
-[ RECORD 1 ]----+------------------------------
pid              | 3069
usesysid         | 16388
usename          | repli
application_name | secondaire3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 38386
backend_start    | 2017-09-08 04:51:11.848485-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/7C000140
write_lsn        | 0/7C000140
flush_lsn        | 0/7C000140
replay_lsn       | 0/7C000140
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 2851
usesysid         | 16388
usename          | repli
application_name | secondaire2
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 38240
backend_start    | 2017-09-08 04:45:28.720828-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/7C000140
write_lsn        | 0/7C000140
flush_lsn        | 0/7C000140
replay_lsn       | 0/7C000140
write_lag        |
flush_lag        |
```

```
replay_lag       |
sync_priority    | 0
sync_state       | async
```

Instance secondaire3

Dans les traces de secondaire3, on constate que l'instance arrive toujours à se connecter à l'instance secondaire1 mais elle est toujours sur la timeline 1 :

```
2017-09-08 05:42:46.090 EDT [1040] LOG:  started streaming WAL from primary
                                         at 0/9000000 on timeline 1
2017-09-08 05:43:12.623 EDT [1040] LOG:  replication terminated by primary server
2017-09-08 05:43:12.623 EDT [1040] DETAIL:  End of WAL reached on timeline 1
                                            at 0/A000098.
2017-09-08 05:43:12.625 EDT [1040] LOG:  fetching timeline history file for
                                         timeline 2 from primary server
cp: cannot stat `/var/lib/pgsql/10/archives/00000001000000000000000A':
    No such file or directory
2017-09-08 05:43:12.804 EDT [1035] LOG:  invalid record length at 0/A000098:
                                         wanted 24, got 0
2017-09-08 05:43:12.811 EDT [1040] LOG:  restarted WAL streaming at 0/A000000
                                         on timeline 1
2017-09-08 05:43:12.811 EDT [1040] LOG:  replication terminated by primary server
2017-09-08 05:43:12.811 EDT [1040] DETAIL:  End of WAL reached on timeline 1
                                            at 0/A000098.
```

On peut également le constater dans la vue `pg_stat_replication` de secondaire1 :

```
postgres=# SELECT * FROM pg_stat_replication WHERE application_name='secondaire3';
-[ RECORD 1 ]----+------------------------------
pid              | 1041
usesysid         | 16385
usename          | repli
application_name | secondaire3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39258
backend_start    | 2017-09-08 05:42:46.073105-04
backend_xmin     |
state            | startup
sent_lsn         | 0/A000098
write_lsn        | 0/A000098
```

```
flush_lsn         | 0/A000098
replay_lsn        | 0/A000098
write_lag         |
flush_lag         |
replay_lag        |
sync_priority     | 0
sync_state        | async

postgres=# SELECT pg_current_wal_lsn();
-[ RECORD 1 ]------+----------
pg_current_wal_lsn | 0/A0001A8
```

Le rejeu s'est arrêté à l'emplacement `0/A000098` alors que l'emplacement actuel est `0/A0001A8`.

Il suffit de configurer `recovery_target_timeline` pour raccrocher l'instance secondaire3 à la nouvelle timeline, avant de la redémarrer :

```
recovery_target_timeline = 'latest'
```

Les traces nous confirment le changement de timeline :

```
2017-09-08 05:59:10.061 EDT [1717] LOG:  database system was shut down in
                                         recovery at 2017-09-08 05:59:09 EDT
2017-09-08 05:59:10.067 EDT [1717] LOG:  restored log file "00000002.history"
                                         from archive
cp: cannot stat `/var/lib/pgsql/10/archives/00000003.history':
    No such file or directory
2017-09-08 05:59:10.073 EDT [1717] LOG:  entering standby mode
2017-09-08 05:59:10.079 EDT [1717] LOG:  restored log file "00000002.history"
                                         from archive
cp: cannot stat `/var/lib/pgsql/10/archives/0000000200000000000000A':
    No such file or directory
cp: cannot stat `/var/lib/pgsql/10/archives/0000000100000000000000A':
    No such file or directory
2017-09-08 05:59:10.200 EDT [1717] LOG:  consistent recovery state reached
                                         at 0/A000098
2017-09-08 05:59:10.200 EDT [1717] LOG:  invalid record length at 0/A000098:
                                         wanted 24, got 0
2017-09-08 05:59:10.201 EDT [1714] LOG:  database system is ready to accept
                                         read only connections
2017-09-08 05:59:10.252 EDT [1726] LOG:  started streaming WAL from primary
```

```
                                        at 0/A000000 on timeline 2
2017-09-08 05:59:10.562 EDT [1717] LOG:  redo starts at 0/A000098
```

Ainsi que la vue pg_stat_replication de secondaire1 :

```
$ psql -p 5433 -c "\x" -c "SELECT * FROM pg_stat_replication ;"
Affichage étendu activé.
-[ RECORD 1 ]----+------------------------------
pid              | 1478
usesysid         | 16385
usename          | repli
application_name | secondaire2
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39310
backend_start    | 2017-09-08 05:46:41.616428-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/A0001A8
write_lsn        | 0/A0001A8
flush_lsn        | 0/A0001A8
replay_lsn       | 0/A0001A8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 1727
usesysid         | 16385
usename          | repli
application_name | secondaire3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39320
backend_start    | 2017-09-08 05:59:10.247828-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/A0001A8
write_lsn        | 0/A0001A8
```

```
flush_lsn        | 0/A0001A8
replay_lsn       | 0/A0001A8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

Switchover du principal

Pré-requis : pour tester les deux opérations nous allons sauvegarder l'instance primaire.

```
$ cp -a /var/lib/pgsql/10/data /var/lib/pgsql/10/data-backup
```

Nous allons également désactiver l'archivage sur l'instance primaire (dans `postgresql.conf` dans `/var/lib/pgsql/10/data/`) et sa sauvegarde (dans `/var/lib/pgsql/10/data-backup/`) afin d'éviter d'écraser les journaux archivés au démarrage de l'instance :

```
archive_mode = off
```

> - Si votre serveur principal a bien été arrêté avant la bascule, vous allez pouvoir le « raccrocher » au secondaire1.

Cette opération est assez simple. Il suffit de créer un fichier `recovery.conf` sur l'instance primaire en indiquant le secondaire1 comme serveur principal :

```
restore_command = 'cp /var/lib/pgsql/9.6/archives/%f %p'
recovery_target_timeline = 'latest'
standby_mode = on
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli password=repli
                    application_name=ex-primaire1'
```

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

Au redémarrage, l'instance primaire passera en `standby_mode` et se connectera à l'instance secondaire1. L'option `recovery_target_timeline` permet au serveur de raccrocher jusqu'à la dernière timeline :

```
2017-09-08 06:03:40.561 EDT [1862] LOG:  database system was shut down
                                         in recovery at 2017-09-08 06:03:39 EDT
2017-09-08 06:03:40.570 EDT [1862] LOG:  restored log file "00000002.history"
                                         from archive
cp: cannot stat `/var/lib/pgsql/10/archives/00000003.history':
```

```
    No such file or directory
2017-09-08 06:03:40.578 EDT [1862] LOG:  entering standby mode
2017-09-08 06:03:40.586 EDT [1862] LOG:  restored log file "00000002.history"
                                         from archive
cp: cannot stat `/var/lib/pgsql/10/archives/00000002000000000000000A':
    No such file or directory
cp: cannot stat `/var/lib/pgsql/10/archives/00000001000000000000000A':
    No such file or directory
2017-09-08 06:03:40.720 EDT [1862] LOG:  consistent recovery state reached
                                         at 0/A000098
2017-09-08 06:03:40.721 EDT [1859] LOG:  database system is ready to accept
                                         read only connections
2017-09-08 06:03:40.722 EDT [1862] LOG:  invalid record length at 0/A000098:
                                         wanted 24, got 0
2017-09-08 06:03:40.774 EDT [1871] LOG:  started streaming WAL from primary
                                         at 0/A000000 on timeline 2
2017-09-08 06:03:41.082 EDT [1862] LOG:  redo starts at 0/A000098
```

On peut le vérifier grâce à la vue `pg_stat_replication` sur secondaire1 :

```
$ psql -x -p 5433 -c "SELECT * FROM pg_stat_replication;"
-[ RECORD 1 ]----+------------------------------
pid              | 1872
usesysid         | 16385
usename          | repli
application_name | ex-primaire1
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39324
backend_start    | 2017-09-08 06:03:40.77027-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/A0001A8
write_lsn        | 0/A0001A8
flush_lsn        | 0/A0001A8
replay_lsn       | 0/A0001A8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
```

```
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 1478
usesysid         | 16385
usename          | repli
application_name | secondaire2
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39310
backend_start    | 2017-09-08 05:46:41.616428-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/A0001A8
write_lsn        | 0/A0001A8
flush_lsn        | 0/A0001A8
replay_lsn       | 0/A0001A8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 3 ]----+------------------------------
pid              | 1727
usesysid         | 16385
usename          | repli
application_name | secondaire3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39320
backend_start    | 2017-09-08 05:59:10.247828-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/A0001A8
write_lsn        | 0/A0001A8
flush_lsn        | 0/A0001A8
replay_lsn       | 0/A0001A8
write_lag        |
flush_lag        |
replay_lag       |
```

```
sync_priority     | 0
sync_state        | async
```

Nous allons maintenant tester le cas où l'instance primaire n'aurait pas été arrêtée avant la bascule. Il pourrait y avoir des transactions dans la timeline 1 qui n'existe pas dans la timeline 2 ce qui rend impossible la mise en réplication avec le secondaire1.

> - Si celui-ci était toujours actif après la promotion du secondaire1, il y aurait des transactions supplémentaires par rapport au secondaire1. Cette situation nécessite soit de reconstruire l'instance soit d'utiliser l'outil pg_rewind.

Pour utiliser `pg_rewind`, il faut s'assurer que l'instance a été créée avec l'option `--data-checksums` ou que le paramètre `wal_log_hints` est à la valeur `on` dans le fichier `postgresql.conf`.

Arrêtons l'ancienne instance primaire :

```
# service postgresql-10 stop
```

Utilisons l'instance sauvegardée précédement :

```
# cd /var/lib/pgsql/10
# mv data data-old
# mv data-backup data
```

Avant de démarrer l'instance il faut indiquer au serveur qu'il n'y a plus d'esclave synchrone :

```
synchronous_standby_names = ''
```

Démarrons l'instance :

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

Créons une table `t6` :

```
$ psql -c "CREATE TABLE t6 (c1 int)" b1
CREATE TABLE
$ psql -c "INSERT INTO t6 VALUES ('1')" b1
INSERT 0 1
```

La table `t6` existe dans la timeline 1 mais pas dans la timeline 2. Nous allons donc utiliser l'outil `pg_rewind` disponible à partir de la version 9.5. Il ne lit et ne copie que les données modifiées. L'outil identifie les blocs correspondants aux transactions « perdues ». Dans

notre exemple, la table t6 n'est présente que sur la nouvelle instance. Ensuite il copie les blocs correspondants sur l'instance cible.

Arrêtons l'instance primaire :

```
# service postgresql-10 stop
```

Maintenant passons à pg_rewind. L'option -n permet d'effectuer un test sans modification :

```
$ /usr/pgsql-10/bin/pg_rewind -D /var/lib/pgsql/10/data/ -n -P \
        --source-server="host=/tmp port=5433 user=postgres"
connected to server
servers diverged at WAL location 0/9000098 on timeline 1
rewinding from last common checkpoint at 0/9000028 on timeline 1
reading source file list
reading target file list
reading WAL in target
need to copy 149 MB (total source directory size is 174 MB)
152927/152927 kB (100%) copied
creating backup label and updating control file
syncing target data directory
Done!
```

On peut relancer la commande sans l'option -n :

```
$ /usr/pgsql-10/bin/pg_rewind -D /var/lib/pgsql/10/data/ -P \
        --source-server="host=/tmp port=5433 user=postgres"
connected to server
servers diverged at WAL location 0/9000098 on timeline 1
rewinding from last common checkpoint at 0/9000028 on timeline 1
reading source file list
reading target file list
reading WAL in target
need to copy 149 MB (total source directory size is 2174 MB)
152952/152952 kB (100%) copied
creating backup label and updating control file
syncing target data directory
Done!
```

On constate qu'il a juste été necessaire de copier 149Mo au lieu des 2 Go de l'instance.

pg_rewind se connecte sur le serveur source et identifie le point de divergence. Ensuite, il liste les fichiers à copier ou supprimer. Puis il copie les blocs modifiés. Enfin il met à

jour le fichier backup_label et le fichier pg_control.

On peut donc créer le fichier `recovery.conf` :

```
restore_command = 'cp /var/lib/pgsql/10/archives/%f %p'
standby_mode = on
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli password=repli
                    application_name=ex-primaire1-rewind'
recovery_target_timelinie = 'latest'
```

Le fichier `postgresql.conf` provenant du secondaire1, il faut pour modifier le port pour qu'il écoute sur le port 5432 :

```
port = 5432
```

Et démarrer l'instance :

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

Dans les traces :

```
2017-09-08 06:19:48.216 EDT [2840] LOG:
           database system was interrupted while in recovery
       at log time 2017-09-08 06:14:11 EDT
2017-09-08 06:19:48.216 EDT [2840] HINT:
           If this has occurred more than once some data might be corrupted and
       you might need to choose an earlier recovery target.
cp: cannot stat `/var/lib/pgsql/10/archives/00000003.history':
    No such file or directory
2017-09-08 06:19:49.149 EDT [2840] LOG:  entering standby mode
2017-09-08 06:19:49.155 EDT [2840] LOG:
           restored log file "00000002.history" from archive
cp: cannot stat `/var/lib/pgsql/10/archives/000000020000000000000009':
    No such file or directory
2017-09-08 06:19:49.306 EDT [2840] LOG:  redo starts at 0/9000098
2017-09-08 06:19:49.341 EDT [2840] LOG:
           unexpected pageaddr 0/9000000 in log segment
       000000020000000000000009, offset 196608
2017-09-08 06:19:49.355 EDT [2846] LOG:
           started streaming WAL from primary at 0/9000000 on timeline 2
2017-09-08 06:19:49.408 EDT [2840] LOG:
           consistent recovery state reached at 0/90318D0
2017-09-08 06:19:49.410 EDT [2837] LOG:
           database system is ready to accept read only connections
```

17.12

La table t6 a bien disparu de l'instance :

```
$ psql -p 5432 -c "SELECT * FROM t6;" b1
ERROR:  relation "t6" does not exist
```

Et l'instance est bien en réplication avec secondaire1 :

```
$ psql -x -p 5433 -c "SELECT * FROM pg_stat_replication WHERE application_name LIKE 'e
-[ RECORD 1 ]----+------------------------------
pid              | 2847
usesysid         | 16385
usename          | repli
application_name | ex-primaire1-rewind
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 39366
backend_start    | 2017-09-08 06:19:49.350772-04
backend_xmin     |
state            | streaming
sent_lsn         | 0/90326F0
write_lsn        | 0/90326F0
flush_lsn        | 0/90326F0
replay_lsn       | 0/90326F0
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

5 RÉPLICATION LOGIQUE

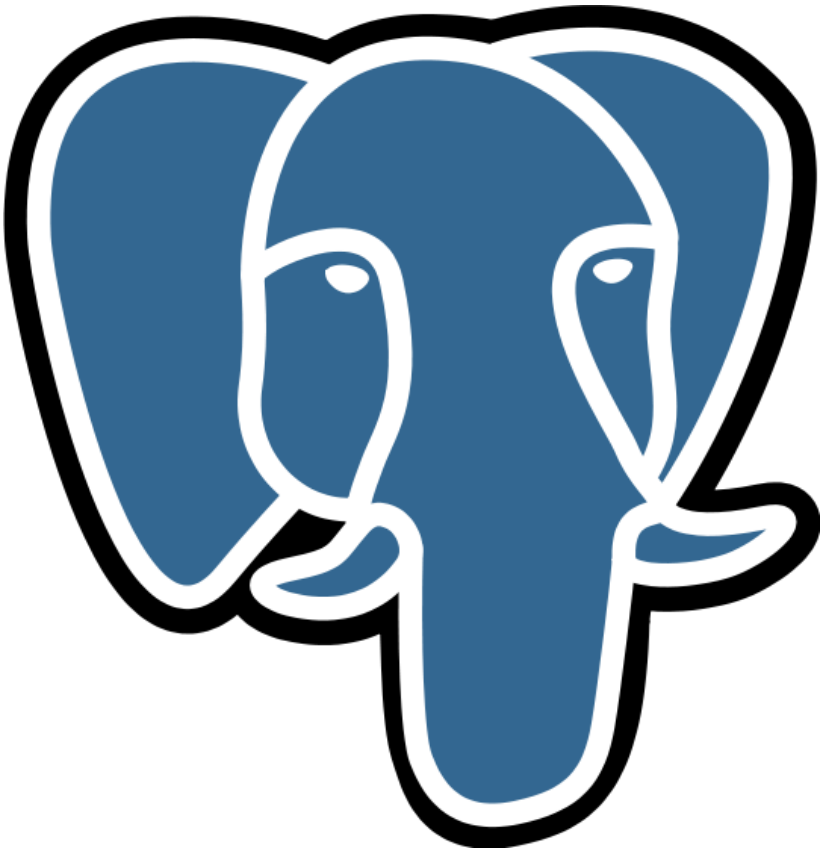


FIGURE 5: POSTGRESQL

5.1 INTRODUCTION

- Principes
- Mise en place
- Supervision

La version 10 ajoute la réplication logique à PostgreSQL. Cette réplication était attendue

depuis longtemps. Ce module permet de comprendre les principes derrière ce type de réplication, sa mise en place, son administration et sa supervision.

5.1.1 AU MENU

- Principes
- Mise en place
- Administration
- Supervision
- Limitations

5.1.2 OBJECTIFS

- Connaître les avantages et limites de la réplication logique
- Savoir mettre en place la réplication logique
- Savoir administrer et superviser une solution de réplication logique

---

5.2 PRINCIPES

- Réplication physique
  - depuis la 9.0
  - beaucoup de possibilités
  - mais des limitations
- Réplication logique
  - permet de résoudre certaines des limitations de la réplication physique
  - auparavant uniquement disponible via des solutions externes
  - en interne depuis la version 10

La réplication existe dans PostgreSQL depuis la version 9.0. Il s'agit d'une réplication physique, autrement dit par application de bloc d'octets ou de delta de bloc. Ce type de réplication a beaucoup évolué au fil des versions 9.X mais a des limitations difficilement contournables directement.

La réplication logique apporte des réponses à ces limitations. Seules des solutions tierces apportaient ce type de réplication à PostgreSQL. Il a fallu attendre la version 10 pour la voir intégrer en natif.

5.2.1 RÉPLICATION PHYSIQUE VS LOGIQUE

- Réplication physique
  - instance complète
  - par bloc
  - asymétrique
  - asynchrone/synchrone
- Réplication logique
  - par table
  - par type d'opération
  - asymétrique (une origine des modifications)
  - asynchrone/synchrone

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.

La réplication logique est une réplication du contenu des tables. 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. Cependant, 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. Enfin, elle fonctionne en asynchrone ou en synchrone.

5.2.2 LIMITATIONS DE LA RÉPLICATION PHYSIQUE

- Pas de réplication partielle
- Pas de réplication entre différentes versions majeures
- Pas de réplication entre différentes architectures
- Pas de réplication multidirectionnelle

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 in-

dex 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.

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).

La réplication logique propose une solution à tous ces problèmes, en dehors de la réplication multidirectionnelle.

5.2.3 QUELQUES TERMES ESSENTIELS

- Serveur origine
  - et serveurs de destination
- Publication
  - et abonnement

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. De lui sont enregistrées les modifications que d'autres serveurs pourront récupérer. Ces serveurs de destination indiquent leur intérêt sur ces modifications en s'abonnant à la publication.

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.

5.2.4 RÉPLICATION EN STREAMING

- Paramètre `wal_level`
- Processus `wal sender`
  - mais pas de `wal receiver`
  - un `logical replication worker` à la place
- Asynchrone / synchrone

La réplication logique utilise le même canal d'informations que la réplication physique : les enregistrements des journaux de transactions. Pour que les journaux disposent de suffisamment d'informations, le paramètre `wal_level` doit être configuré en adéquation.

Une fois cette configuration effectuée et PostgreSQL redémarré sur le serveur origine, le serveur destination pourra se connecter au serveur origine dans le cadre de la réplication. Lorsque cette connexion est faite, un processus `wal sender` apparaîtra sur le serveur origine. Ce processus sera 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`).

5.2.5 GRANULARITÉ

- Par table
  - publication pour toutes les tables
  - publications pour des tables spécifiques
- Par opération
  - insert, update, delete

La granularité de la réplication physique est simple : c'est l'instance et rien d'autre.

Avec la réplication logique, la granularité est la table. 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. Ceci n'est possible que dans un cas précis : la publication a été créée en demandant la publication de toutes les tables (clause `FOR ALL TABLES`).

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.

17.12

Par défaut, tout est publié.

5.2.6 LIMITATIONS DE LA RÉPLICATION LOGIQUE

- Pas de réplication des requêtes DDL
  - et donc pas de `TRUNCATE`
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Contraintes d'unicité obligatoires pour les `UPDATE`/`DELETE`

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. Donc une table nouvellement créée ne sera pas forcément répliquée. 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.

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

Les Large Objects étant stockés dans une table système, ils ne sont pas pris en compte par la réplication logique.

Les opérations `UPDATE` et `DELETE` nécessitent la présence d'une contrainte unique pour s'assurer de modifier ou supprimer les bonnes lignes.

5.3 MISE EN PLACE

- Cas simple
  - 2 serveurs
  - une seule origine
  - un seul destinataire
  - une seule publication
- Plusieurs étapes
  - configuration du serveur origine
  - configuration du serveur destination
  - création d'une publication
  - ajout d'une souscription

Dans cette partie, nous allons aborder un cas simple avec uniquement deux serveurs. Le premier sera l'origine, le second sera le destinataire des informations de réplication. Toujours pour simplifier l'explication, il n'y aura pour l'instant qu'une seule publication.

La mise en place de la réplication logique consiste en plusieurs étapes :

- la configuration du serveur origine ;
- la configuration du serveur destination ;
- la création d'une publication ;
- la souscription à une publication.

Nous allons voir maintenant ces différents points.

5.3.1 CONFIGURER LE SERVEUR ORIGINE

- Création et configuration de l'utilisateur de réplication
  - et lui donner les droits de lecture des tables à répliquer
- Configuration du fichier `postgresql.conf`
  - `wal_level = logical`
- Configuration du fichier `pg_hba.conf`
  - autoriser une connexion de réplication du serveur destination

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 le publieur.

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 :

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

Les journaux de transactions doivent disposer de suffisamment d'informations pour que le `wal sender` puisse envoyer les bonnes informations au `logical replication worker`. Pour cela, il faut configurer le paramètre `wal_level` à la valeur `logical` dans le fichier `postgresql.conf`.

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

```
host replication logrepli XXX.XXX.XXX.XXX/XX md5
```

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.

Si le paramètre `wal_level` a été modifié, il est nécessaire de redémarrer le serveur PostgreSQL. Si seul le fichier `pg_hba.conf` a été modifié, seul un rechargement de la configuration est demandé.

5.3.2 CONFIGURATION DU SERVEUR DESTINATION

- Création de l'utilisateur de réplication
- Création, si nécessaire, des tables répliquées
  - `pg_dump -h serveur_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`.

Cependant, il est nécessaire d'avoir l'utilisateur de réplication. La requête de création est identique :

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

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

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

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

5.3.3 CRÉER UNE PUBLICATION

- Ordre SQL

```
CREATE PUBLICATION nom
  [ FOR TABLE [ ONLY ] nom_table [ * ] [, ...]
| FOR ALL TABLES ]
  [ WITH ( parametre_publication [= valeur] [, ... ] ) ]
```

- parametre_publication étant seulement le paramètre publish
  - valeurs possibles : insert, update, delete
  - les trois par défaut

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.

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

Cette publication est concernée par défaut par toutes les opérations d'écriture (`INSERT`, `UPDATE`, `DELETE`). 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` et/ou `delete` et en les séparant par des virgules si on en indique plusieurs.

5.3.4 SOUSCRIRE À UNE PUBLICATION

- Ordre SQL

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

- infos_connexion est la chaîne de connexion habituelle

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

Le serveur s'indique avec la chaîne `infos_connexion`, dont la syntaxe est la syntaxe habituelle des chaînes de connexion. Pour rappel, on utilise les mots clés `host`, `port`, `user`, `password`, `dbname`, etc.

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

Les paramètres de souscription sont détaillés dans la slide suivante.

5.3.5 OPTIONS DE LA SOUSCRIPTION

- `copy_data`
  - copie initiale des données (activé par défaut)
- `create_slot`
  - création du slot de réplication (activé par défaut)
- `enabled`
  - activation immédiate de la souscription (activé par défaut)
- `slot_name`
  - nom du slot (par défaut, le nom de la souscription)
- `synchronous_commit`
  - pour surcharger la valeur du paramètre `synchronous_commit`
- `connect`
  - connexion immédiate (activé par défaut)

Les options de souscription sont assez nombreuses et permettent de créer une souscription pour des cas particuliers. Par exemple, si le serveur destination a déjà les données du serveur origine, il faut placer le paramètre `copy_data` à la valeur `off`.

5.4 EXEMPLES

- 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.

5.5 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_1 ADD PRIMARY KEY(id_t3, clepartition_t3);
ALTER TABLE t3_2 ADD PRIMARY KEY(id_t3, clepartition_t3);
ALTER TABLE t3_3 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

5.5.1 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.

5.5.2 CONFIGURATION DU SERVEUR ORIGINE

- 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
```

- Fichier `pg_hba.conf`

```
host replication 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`

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. 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.

5.5.3 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.

5.5.4 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.

5.5.5 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
- Les données initiales sont immédiatement transférées

17.12

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.

Les données initiales de la table t1 sont envoyées du serveur s1 vers le serveur s2.

5.5.6 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 :

```
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 le serveur s2 :

```
b1=# SELECT count(*) FROM t1;
 count
-------
   100
(1 row)

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)
```

5.5.7 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. La publication doit mentionner la liste des tables à répliquer. Chaque nom de table est séparé par une virgule.

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 dans le slide.

5.5.8 TESTS DE LA RÉPLICATION PARTIELLE

- Insertion, modification, suppression sur les différentes tables de s1
- Vérifications sur s3
  - seules t1 et t2 doivent avoir les mêmes données entre s1 et s3
  - t3 et ses partitions ne doivent pas changer sur s3

Sur s3, nous n'avons que les données de t1 et t2 :

```
b1=# SELECT count(*) FROM t1;
 count
-------
   100
(1 row)

b1=# SELECT count(*) FROM t2;
 count
-------
  1000
(1 row)

b1=# SELECT count(*) FROM t3;
 count
-------
     0
(1 row)
```

À noter que nous avons déjà les données précédemment modifiées :

```
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)
```

Maintenant, ajoutons une ligne dans chaque table de s1 :

```
b1=# INSERT INTO t1 VALUES (102, 't1, ligne 102');
INSERT 0 1
b1=# INSERT INTO t2 VALUES (1001, 't2, ligne 1002');
INSERT 0 1
b1=# INSERT INTO t3 VALUES (-1, 't3, cle 1, ligne -1', 1);
INSERT 0 1
```

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

```
b1=# SELECT * FROM t1 WHERE id_t1=102;
 id_t1 |   label_t1
-------+---------------
   102 | t1, ligne 102
(1 row)
```

```
b1=# SELECT * FROM t2 WHERE id_t2=1001;
 id_t2 |    label_t2
-------+----------------
  1001 | t2, ligne 1002
(1 row)

b1=# SELECT * FROM t3 WHERE id_t3 < 0;
 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)
```

5.5.9 RÉPLICATION CROISÉE

- On veut pouvoir écrire sur une table sur le serveur s1
  - et répliquer les écritures de cette table sur s4
- On veut aussi pouvoir écrire sur une (autre) table sur le serveur s4
  - et répliquer les écritures de cette table sur s1
- Pour rendre le système encore plus intéressant, on utilisera la table partitionnée

La réplication logique ne permet pas pour l'instant de faire du multidirectionnel (multi-maître) pour une même table. Cependant, il est tout à fait possible de faire en sorte qu'un ensemble de tables soit répliqué du serveur s1 (origine) vers le serveur s4 et qu'un autre ensemble de tables soit répliqué du serveur s4 (origine) vers le serveur s1 (destination).

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` le serveur s1 et dans la partition `t3_2` sur le serveur s2, 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.

5.5.10 RÉPLICATION DE T3_1 DE S1 VERS S4

- Créer la publication partielle sur s1

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

- Souscrire sur s4 à cette nouvelle publication de s1

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

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.

5.5.11 CONFIGURATION DE S4

- s4 devient aussi un serveur origine
  - il doit en avoir la configuration
- Fichier `postgresql.conf`

```
wal_level = logical
```

- Fichier `pg_hba.conf`

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

- Redémarrer le serveur s4

Le serveur s4 n'est pas qu'un serveur destination, il devient aussi un serveur origine. Il est donc nécessaire de le configurer pour ce nouveau rôle. Cela passe par la configuration des fichiers `postgresql.conf` et `pg_hba.conf` comme indiqué dans le slide. Ceci fait, il est nécessaire de redémarrer le serveur PostgreSQL sur s4.

5.5.12 RÉPLICATION DE T3_2 DE S4 VERS S1

- Créer la publication partielle sur s4

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

- Souscrire sur s1 à cette publication de s4

```
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 souscript à cette publication sur le serveur s1.

5.5.13 TESTS DE LA RÉPLICATION CROISÉE

- Insertion, modification, suppression sur t3, pour la partition 1, du serveur s1
- Vérifications sur s4, les nouvelles données doivent être présentes
- Insertion, modification, suppression sur t3, pour la partition 2, du serveur s4
- Vérifications sur s1, les nouvelles données doivent être présentes

Sur s1 :

```
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
(1 row)
```

Sur s4 :

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
(1 row)

b1=# INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 2);
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 |               2
(2 rows)
```

Sur s1 :

```
b1=# 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)
```

5.6 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 du 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.

5.6.1 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.

Pour commencer, un serveur en version 10 dispose d'un processus supplémentaire, le `logical replication launcher`. Ce processus a pour but de demander le lancement d'un `logical replication 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 (on a aussi le terme de `apply worker`, notamment dans certains messages des traces). 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: stats collector process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication 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 worker`). Il a donc les processus suivant :

```
postmaster -D /opt/postgresql/datas/s1
postgres: checkpointer process
postgres: writer process
postgres: wal writer process
postgres: autovacuum launcher process
postgres: stats collector process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication 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
```

5.6.2 FICHIERS

- 2 répertoires importants
- `pg_replslot`
  - slots de réplication
  - 1 répertoire par slot
  - 1 fichier `state` dans le répertoire
- `pg_logical`
  - méta-données
  - snapshots

La réplication logique maintient des méta-données dans deux répertoires : `pg_replslot` et `pg_logical`. Ce dernier contient notamment les snapshots des transactions longues et peut donc avoir une taille assez importante si le serveur exécute beaucoup de transactions longues avec du volume en écriture.

Il est donc important de surveiller la place prise par ce répertoire.

5.6.3 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
- Problème
  - 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 :

```
b1=# CREATE ROLE u1 LOGIN;
CREATE ROLE
b1=# GRANT ALL ON ALL TABLES IN SCHEMA public TO u1;
GRANT
```

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

```
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
```

```
b1=> INSERT INTO t1 VALUES (103, 't1 sur s2, ligne 103');
INSERT 0 1
```

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 :

```
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# REVOKE INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM u1;
REVOKE
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> INSERT INTO t1 VALUES (104);
ERROR:  permission denied for relation t1
```

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

Si cette interdiction n'est pas réalisée, on peut arriver à des problèmes très gênants. Par exemple, nous avons inséré dans la table `t1` de s2 la valeur 103 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
(1 row)
```

Cette ligne n'apparaît pas sur s1 :

```
b1=# 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 :

```
b1=> INSERT INTO t1 VALUES (103, 't1 sur s1, ligne 103');
INSERT 0 1
```

Et maintenant, on se trouve avec deux serveurs désynchronisés :

- sur s1 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
```

```
   103 | t1 sur s1, ligne 103
(1 row)
```

- sur s2 :

```
b1=# 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.

Le processus de réplication logique n'arrive plus à appliquer les données sur s2, d'où les messages suivants 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 :

```
b1=# DELETE FROM t1 WHERE id_t1=103;
DELETE 1
b1=# 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 :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

5.6.4 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.

5.6.5 QUE FAIRE POUR LES NOUVELLES TABLES

- Publication complète
  - rafraîchir les souscriptions concernées
- Publication partielle
  - ajouter la nouvelle table dans les souscriptions concernées

La création d'une table est une opération DDL. Elle est donc ignorée dans le contexte de la réplication logique. Il est tout à fait entendable qu'on ne veuille pas la répliquer, auquel cas il n'est rien besoin de faire. Mais si on souhaite répliquer son contenu, deux cas se présentent : la publication a été déclarée `FOR ALL TABLES` ou elle a été déclarée pour certaines tables uniquement.

Dans le cas où la publication ne concerne qu'un sous-ensemble de tables, il faut ajouter la table à la publication avec l'ordre `ALTER PUBLICATION...ADD TABLE`.

Dans le cas où elle a été créé avec la clause `FOR ALL TABLES`, la nouvelle table est immédiatement prise en compte dans la publication. Cependant, pour que les serveurs des-

tinataires gèrent aussi cette nouvelle table, il va falloir leur demander de rafraîchir leur souscription avec l'ordre `ALTER SUBSCRIPTION...REFRESH PUBLICATION`.

Voici un exemple de ce deuxième cas.

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

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
CREATE TABLE
b1=# GRANT SELECT ON TABLE t4 TO logrepli;
GRANT
b1=# INSERT INTO t4 VALUES (1);
INSERT 0 1
```

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

```
b1=# 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 :

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
CREATE TABLE
b1=# 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 :

```
b1=# ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;
ALTER SUBSCRIPTION
b1=# SELECT * FROM t4;
 id_t4
-------
     1
(1 row)
```

5.6.6 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.

5.6.7 GÉRER LES SAUVEGARDES

- `pg_dumpall` et `pg_dump`
  - sauvegardent publications et souscriptions
  - nouvelles options `--no-publications` et `--no-subscriptions`
- Sauvegarde PITR
  - sauvegardent publications et souscriptions

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;
```

Enfin, pour une souscription, l'ordre SQL 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 que nous avons exécuté, 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). Une fois la sauvegarde restaurée et que les vérifications nécessaires ont été effectuées, il est possible d'activer la souscription et de la rafraîchir :

```
b1=# ALTER SUBSCRIPTION subscr_complete ENABLED;
ALTER SUBSCRIPTION
b1=# ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

Ces opérations sont obligatoirement manuelles.

Quant aux sauvegardes PITR, ce sont des sauvegardes intégrales, les souscriptions et les publications sont elles aussi conservées.

5.7 SUPERVISION

- Méta-données
- Statistiques
- Outils

5.8 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

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

```
b1=# SELECT * FROM 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
(3 rows)
```

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

```
b1=# SELECT * FROM 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
(9 rows)
```

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 1
ORDER BY 1;
```

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

- la liste des publications par table :

```
SELECT tablename, array_agg(pubname ORDER BY pubname) AS publications_list
FROM pg_publication_tables
GROUP BY 1
```

```
ORDER BY 1;

 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}
(7 rows)
```

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

```
b1=# \x
Expanded display is on.
b1=# 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}
```

5.9 VUES STATISTIQUES

- `pg_stat_replication`
  - statut de réplication
- `pg_stat_subscription`
  - état des souscriptions
- `pg_replication_origin_status`
  - statut des origines de réplication

Comme pour la réplication physique, le retard de réplication est calculable en utilisant les informations de la vue `pg_stat_replication` :

```
b1=# 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
```

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

```
b1=# 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
```

5.10 OUTILS

- check_pgactivity
  - replication_slots
- check_postgres
  - same_schema

Peu d'outils ont déjà été mis à jour pour ce type de réplication.

Néanmoins, 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[27]). 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=-1
Perfdata        : subscr_partielle=-1
Perfdata        : subscr_t3_1=-1
```

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

```
b1=# INSERT INTO t1 SELECT generate_series(1000000, 2000000);
INSERT 0 1000001
```

L'outil détecte bien que le slot subscr_partielle a un retard conséquent (7 journaux de transactions) :

```
$ ./check_pgactivity -s replication_slots -p 5441 -F human
Service         : POSTGRES_REPLICATION_SLOTS
Returns         : 0 (OK)
Message         : Replication slots OK
Perfdata        : subscr_complete=-1
Perfdata        : subscr_partielle=7
Perfdata        : subscr_t3_1=-1
```

Il est aussi possible d'utiliser l'action same_schema avec l'outil check_postgres (disponible aussi sur github[28]) pour détecter des différences de schémas entre deux serveurs (l'origine et une destination).

[27] https://github.com/OPMDG/check_pgactivity
[28] https://github.com/bucardo/check_postgres/

5.11 RAPPEL DES LIMITATIONS

- Pas de réplication des requêtes DDL
  - et donc pas de `TRUNCATE`
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Contraintes d'unicité obligatoires pour les `UPDATE`/`DELETE`

5.12 CONCLUSION

- Enfin une réplication logique
- Réplication complète ou partielle
  - par objet (table)
  - par opération (insert/update/delete)

5.12.1 QUESTIONS

N'hésitez pas, c'est le moment !

5.13 TRAVAUX PRATIQUES

5.13.1 ÉNONCÉS

Pré-requis

La réplication logique n'étant disponible qu'en version 10, vous devez tout d'abord installer cette version.

Vous devez disposer de quatre instances PostgreSQL. Soit sur un seul serveur (les instances utiliseront alors des numéros de port différents), soit sur 4 serveurs (physiques ou virtuels).

Le schéma de la base b1 de l'instance origine est le suivant :

```
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_1 ADD PRIMARY KEY(id_t3, clepartition_t3);
ALTER TABLE t3_2 ADD PRIMARY KEY(id_t3, clepartition_t3);
ALTER TABLE t3_3 ADD PRIMARY KEY(id_t3, clepartition_t3);
ALTER TABLE t3_4 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

Réplication complète d'une base

Répliquer toute la base b1 sur le serveur s2.

Réplication partielle d'une base

Répliquer uniquement les tables t1 et t2 de la base b1 sur le serveur s3.

Réplication croisée

Répliquer la partition t3_1 du serveur s1 vers le serveur s4. Répliquer la partition t3_2 du serveur s4 vers le serveur s2.

5.13.2 SOLUTIONS

Réplication complète d'une base

> Sur s1, créer l'utilisateur de réplication et lui donner les droits de lecture sur les tables

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

> Sur s1, modifier la configuration du paramètre wal_level dans le fichier postgresql.conf

```
wal_level = logical
```

> Sur s1, modifier la configuration des connexions dans le fichier pg_hba.conf

```
host replication logrepli 192.168.10.0/24 trust
```

> Sur s1, redémarrer le serveur PostgreSQL

> Sur s2, créer l'utilisateur de réplication

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

> Sur s2, créer la base b1

```
createdb -h s2 b1
```

> Sur s2, ajouter dans la base b1 les tables répliquées (sans contenu)

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

> Sur s1, créer la publication pour toutes les tables

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

> Sur s2, créer la souscription

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

Vérifier sur s1, dans la vue pg_stat_replication l'état de la réplication logique. Sur s2, consulter pg_stat_subcription, vérifier que les tables ont le même contenu que sur s1 et que les modifications sont également répliquées.

Réplication partielle d'une base

> Sur s1, créer la publication pour t1 et t2

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

> Sur s3, créer la base s1, créer les tables à répliquer, puis souscrire à la nouvelle publication de s1

```
createdb -h s3 b1
```

```
pg_dump -h s1 -s -t t1 -t t2   b1 | psql -h s3 b1
```

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

Réplication croisée d'une base

> Sur s1, créer la publication pour t3_1

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

> Sur s4, souscrire à cette nouvelle publication de s1. Pour créer la table t3_1 il faut aussi créer mère t3.

```
createdb -h s4 b1
```

```
pg_dump -h s1 -s -t t3 -t t3_1  b1| psql -h s4 b1
```

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

> Sur s4, modifier la valeur du paramètre wal_level dans le fichier postgresql.conf

```
wal_level = logical
```

> Sur s4, modifier le contenu du fichier pg_hba.conf

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

> Sur s4, redémarrer le serveur s4

> Sur s4, créer la publication pour t3_4. Il faudra donner les droits de lecture à logrepli :

```
GRANT SELECT ON t3_4 TO logrepli ;
```

```
pg_dump -h s1 -s -t t3_4  b1| psql -h s4 b1
```

```
CREATE PUBLICATION publi_t3_4
  FOR TABLE t3_4;
```

> Sur s1, souscrire à cette nouvelle publication de s4. Il faudra donner les droits nécessaires à logrepli.

```
GRANT SELECT,DELETE,INSERT ON t3_4  TO logrepli;
```

```
CREATE SUBSCRIPTION subscr_t3_4
  CONNECTION 'host=192.168.10.4 user=logrepli dbname=b1'
  PUBLICATION publi_t3_4;
```

Insérer des données dans t3_4 sur s4 et vérifier que la réplication se fait de s4 à s1. Vérifier que la réplication de t3_4 ne s'est pas faite spontanément vers s2. Vérifier qu'elle se fait avec :

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