Module W1

Solutions de réplication

24.04

Table des matières

Sur ce document

Formation	Module W1
Titre	Solutions de réplication
Révision	24.04
PDF	https://dali.bo/w1_pdf
EPUB	https://dali.bo/w1_epub
HTML	https://dali.bo/w1_html
Slides	https://dali.bo/w1_slides

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

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

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

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

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

1/ Solutions de réplication

Source de la photo : epSos.de[1] via Wikimedia[2], licence CC-BY-2.0.

[1] https://www.flickr.com/photos/36495803@N05/3574411866/
[2] https://commons.wikimedia.org/wiki/File:Green_Elephants_Garden_Sculptures.jpg

1.1 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.

1.1.1 Au menu

- Rappels théoriques
- Réplication interne
  - réplication physique
  - réplication logique
- Quelques 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*.

Depuis la version 10 existe aussi une réplication logique, basée sur le transfert du résultat des ordres.

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

1.1.2 Objectifs

- Identifier les différences entre les solutions de réplication 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.

1.2 RAPPELS THÉORIQUES

- Termes
- 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.

1.2.1 Cluster, primaire, secondaire, *standby*…

- **Cluster** : ambiguïté !
  - groupe de bases de données = 1 instance (PostgreSQL)
  - groupe de serveurs (haute disponibilité et/ou réplication)
- Pour désigner les membres :
  - Primaire/*primary*
  - Secondaire/*standby*

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*. En français, on évitera tout ambiguïté en parlant d'« instance ».

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 primaire et de ses deux serveurs secondaires compose un cluster de réplication.

Le serveur, généralement unique, ouvert en écriture est désigné comme « **primaire** » (*primary*), parfois « principal ». Les serveurs connectés dessus sont « **secondaires** » (*secondary*) ou « *standby* » (prêts à prendre le relai). Les termes « maître/esclave » sont à éviter mais encore très courants. On trouvera dans le présent cours aussi bien « primaire/secondaire » que « principal/*standby* ».

1.2.2 Réplication asynchrone asymétrique

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

Dans la réplication asymétrique, seul le serveur primaire accepte des écritures, et les serveurs secondaires ne sont accessibles qu'en lecture.

Dans la réplication asynchrone, les écritures sont faites sur le primaire et le client reçoit une validation de l'écriture avant même qu'elles ne soient poussées vers le secondaire. La mise à jour des tables répliquées **est différée** (asynchrone). Le délai de réplication dépend de la technique et de la charge.

L'inconvénient de ce délai est que certaines données validées sur le primaire pourraient ne pas être disponibles sur les secondaires si le primaire est détruit avant que toutes les données, déjà validées auprès des clients, ne soient poussées sur les secondaires.

Autrement dit, il existe une fenêtre de temps, plus ou moins longue, de perte de données (qui entre dans le calcul du RPO).

1.2.3 Réplication asynchrone symétrique

- **Symétrique**
  - « multi-maîtres »
  - écritures sur les différents primaires
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents primaires
- **Asynchrone**
  - la réplication des écritures est différées
  - perte de données possible en cas de crash du serveur primaire
  - **risque d'incohérences !**
- Exemples :
  - BDR (EDB) : réplication logique
  - Bucardo : réplication par triggers

Dans la réplication symétrique, tous les serveurs sont accessibles aussi bien en lecture qu'en écriture. On parle souvent de « multi-maîtres ».

La réplication asynchrone, comme indiqué précédemment, envoie des modifications vers les autres membres du cluster mais n'attend aucune validation de leur part. Il y a donc toujours un risque de perte de données déjà validées si le serveur tombe sans avoir eu le temps de les répliquer vers au moins un autre serveur du cluster.

Ce mode de réplication ne respecte généralement pas les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) car si une copie échoue sur l'autre primaire 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, pour gérer ces cas de figure.

PostgreSQL ne supporte pas la réplication symétrique nativement. Plusieurs projets ont tenté de remplir ce vide.

BDR[3], de 2nd Quadrant (à présent EDB), se base sur la réplication logique et une extension propre au-dessus de PostgreSQL. BDR assure une réplication symétrique de toutes les données concernées de plusieurs instances toutes liées aux autres, et s'adresse notamment au cas de bases dont on a besoin dans des lieux géographiquement très éloignés. La gestion des conflits est automatique, mais les verrous ne sont pas propagés pour des raisons de performance, ce qui peut donc poser des problèmes d'intégrité, selon les options choisies. Les données peuvent différer entre les nœuds. L'application

[3] https://www.enterprisedb.com/docs/bdr/latest/

doit tenir compte de tout cela. Par exemple, il vaut mieux privilégier les UUID pour éviter les conflits. Les premières versions de BDR étaient sous licence libre, mais ne sont plus supportées, et la version actuelle (BDR4) est propriétaire et payante.

Bucardo[4] se base, lui, sur de la réplication par triggers. Il sera évoqué plus loin.

Si l'application le permet, il est possible de se rabattre sur un modèle où chaque instance est le point d'entrée unique de certaines données (par exemple selon la géographie), et les autres n'en ont que des copies en lecture, obtenues d'une manière ou d'une autre, ou doivent accéder au serveur responsable en écriture. Il s'agit alors plus d'une forme de *sharding* que de véritable réplication symétrique.

1.2.4 Réplication synchrone asymétrique

- **Asymétrique**
  - écritures sur un serveur primaire unique
  - lectures sur le serveur primaire et/ou les secondaires
- **Synchrone**
  - les écritures sur les secondaires sont immédiates
  - le client sait si sa commande a réussi sur plusieurs serveurs

Dans la réplication asymétrique, seul le serveur primaire accepte des écritures, les secondaires ne sont accessibles qu'en lecture.

Dans la réplication synchrone, le client envoie sa requête en écriture sur le serveur primaire, le serveur primaire l'écrit sur son disque, il envoie les données au serveur secondaire 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. Concrètement, un ordre `COMMIT` rend la main une fois l'écriture validée sur plusieurs serveurs, généralement au moins deux (un primaire et un secondaire). 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 serveur primaire s'arrête brutalement et qu'il est nécessaire de basculer sur un serveur secondaire. L'inconvénient majeur est que cela ralentit fortement les écritures.

PostgreSQL permet d'ajuster ce ralentissement à la criticité. Par défaut (paramètre `synchronous_commit` à `on`), la réplication synchrone garantit que le serveur secondaire a bien écrit la transaction dans ses journaux et qu'elle a été synchronisée sur son disque (`fsync`), en plus de celui du primaire

[4] https://www.bucardo.org/Bucardo/

bien sûr. En revanche, elle ne garantit pas forcément que le secondaire a bien rejoué la transaction : il peut se passer un laps de temps où une lecture sur le secondaire serait différente du serveur primaire (le temps que le secondaire rejoue la transaction). PostgreSQL dispose d'un mode (`synchronous_commit` à `remote_apply`) qui permet d'avoir la garantie que les modifications sont rejouées sur le secondaire, au prix d'une latence supplémentaire. À l'inverse, on peut estimer qu'il est suffisant que les données soient juste écrites dans la mémoire cache du serveur secondaire, et pas forcément sur disque, pour être considérées comme répliquées (`synchronous_commit` à `remote_write`). La synchronicité peut être désactivée pour les requêtes peu critiques (`synchronous_commit` à `local`, voire `off`). Ce paramétrage peut être ajusté selon les besoins, requête par requête.

PostgreSQL permet aussi de disposer de plusieurs secondaires synchrones.

1.2.5 Réplication synchrone symétrique

- **Symétrique**
  - écritures sur les différents serveurs primaires
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents serveurs
- **Synchrone**
  - les écritures sur les autres serveurs sont immédiates
  - le client sait si sa commande est validée sur plusieurs serveurs
  - 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 risque de perte de données, 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.

PostgreSQL ne supporte pas la réplication symétrique, donc encore moins la symétrique synchrone. Certains projets évoqués essaient ou ont essayé d'apporter cette fonctionnalité.

Le besoin d'une architecture « multi-maîtres » revient régulièrement, mais il faut s'assurer qu'il est réel. Avant d'envisager une architecture complexe, et donc source d'erreurs, optimisez une installation asy-

métrique simple et classique, quitte à multiplier les serveurs secondaires, et testez ses performances : PostgreSQL pourrait bien vous surprendre !

Selon les cas d'utilisation, la réplication logique peut aussi être utile.

1.2.6 Diffusion des modifications

- 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. En effet, 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 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 ne sont généralement pas gérées. Autrement dit, l'ajout d'une table, l'ajout d'une colonne demande une administration plus poussée, non automatisable.

La deuxième solution (par journaux de transactions) est bien plus intéressante car les journaux contiennent toutes les modifications, données comme structures. Il suffit au secondaire de réappli-

quer tous les journaux provenant du primaire pour être à l'image exacte de celui-ci. De ce fait, une fois mise en place, cette méthode requiert peu de maintenance. PostgreSQL offre nativement depuis longtemps deux variantes de cette solution : par journaux entiers (*log shipping*) ou par flux (*streaming replication*).

PostgreSQL permet, depuis la version 10, le décodage logique des modifications de données correspondant aux blocs modifiés dans les journaux de transactions. L'objectif est de permettre l'extraction logique des données écrites permettant la mise en place d'une réplication logique des résultats entièrement intégrée, donc sans triggers. Les modifications de structures doivent être gérées à la main.

1.3 RÉPLICATION INTERNE PHYSIQUE

- Réplication
  - asymétrique
  - asynchrone (défaut) ou synchrone (et selon les transactions)
- Secondaires
  - non disponibles (*Warm Standby*)
  - disponibles en lecture seule (*Hot Standby*)
  - cascade
  - retard programmé

La réplication physique de PostgreSQL est par défaut **asynchrone** et **asymétrique**. Il est possible de sélectionner le mode synchrone/asynchrone pour chaque serveur secondaire individuellement, et séparément pour chaque transaction, en modifiant le paramètre `synchronous_commit`.

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

Un secondaire peut récupérer les informations de réplication depuis un autre serveur secondaire (fonctionnement en cascade).

Le serveur secondaire peut aussi n'appliquer les informations de réplication qu'après un délai configurable.

1.3.1 Log Shipping

- But :
  - envoyer les journaux de transactions à un secondaire
- Première solution disponible
- Gros inconvénients :
  - perte possible de plusieurs journaux
  - latence à la réplication
  - penser à `archive_timeout` ou `pg_receivewal`

Le *log shipping* permet d'envoyer les journaux de transactions terminés sur un autre serveur. Ce dernier peut être un serveur secondaire, 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 le secondaire. Or, un journal de 16 Mo peut contenir plusieurs centaines de transactions ! Si l'archivage a du retard (grosse charge, réseau saturé…), plusieurs journaux peuvent même être en attente. Le retard du secondaire par rapport au primaire peut donc devenir important, ce qui est gênant dans le cas d'un *standby* utilisé en lecture seule, par exemple dans le cadre d'une répartition de la charge de lecture.

En conséquence il est possible de perdre toutes les transactions contenues dans le journal de transactions en cours, voire tous ceux en retard, en cas de *failover* et de destruction physique des journaux sur le primaire.

On peut cependant moduler le risque de trois façons:

- sauf avarie très grave sur le serveur primaire, les journaux de transactions en attente peuvent être récupérés et appliqués 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. Attention toutefois, les journaux archivés font toujours 16 Mo, qu'ils soient archivés remplis ou non ;
- on peut utiliser l'outil `pg_receivewal` (nommé `pg_receivexlog` jusqu'en 9.6) qui crée à distance les journaux de transactions à partir d'un flux de réplication.

1.3.2 Streaming replication

- But
  - avoir un retard moins important sur le serveur secondaire
- Rejouer **les enregistrements de transactions** du serveur primaire par **paquets**
  - paquets plus petits qu'un journal de transactions

L'objectif du mécanisme de la *streaming replication* est d'avoir un secondaire qui accuse moins de retard. En effet, comme on vient de le voir, le *log shipping* exige d'attendre qu'un journal soit complètement rempli avant qu'il ne soit envoyé au serveur secondaire.

La réplication par *streaming* 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 du contenu des WAL entre le serveur primaire et le serveur secondaire. Ce flux est totalement indépendant de l'archivage du WAL. Ainsi, en cas de perte du serveur primaire, sauf retard à cause d'une saturation quelconque, la perte de données est très faible.

Les délais de réplication entre le serveur primaire et le serveur secondaire sont très courts : une modification sur le serveur primaire sera en effet très rapidement répliqué sur un secondaire.

C'est une solution éprouvée et au point depuis des années. Néanmoins, elle a ses propres inconvénients : réplication de l'instance complète, architecture matérielle et version majeure de PostgreSQL forcément identiques entre les serveurs du cluster, etc.

1.3.3 *Warm Standby*

- Serveur de secours
  - prêt à prendre le relai du primaire
  - (presque) identique au primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée
- En pratique, préférer le *Hot Standby*

Le *Warm Standby* existe depuis la version 8.2. La robustesse de ce mécanisme simple est prouvée depuis longtemps.

Les journaux de transactions sont répliqués en *log shipping* ou *streaming replication* selon la version, le besoin et les contraintes d'architecture. Le serveur secondaire est en mode *recovery* perpétuel et applique automatiquement les journaux de transaction reçus.

Un serveur en *Warm Standby* n'accepte aucune connexion entrante. Il n'est utile que comme réplicat prêt à être promu en production à la place de l'actuel primaire en cas d'incident. Les serveurs secondaires sont donc généralement paramétrés directement en *Hot Standby*.

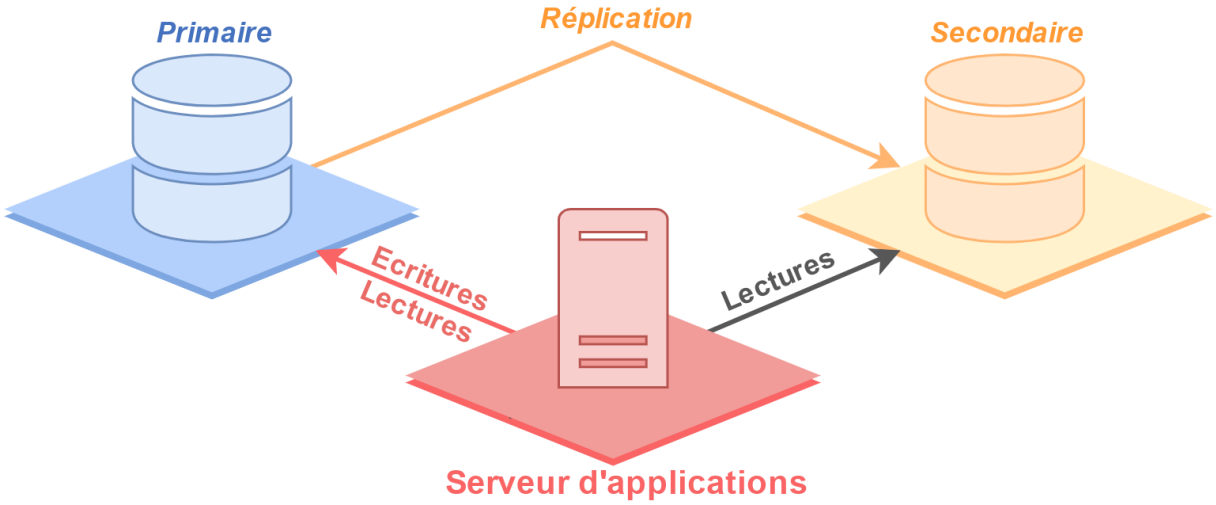
1.3.4 *Hot Standby*

- Serveur secondaire
  - accepte les connexions entrantes
  - requêtes en lecture seule et sauvegardes
  - prêt à prendre le relai du primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée

Le *Hot Standby* est une évolution du *Warm Standby* : il accepte les connexions des utilisateurs et permet d'exécuter des requêtes en lecture seule.

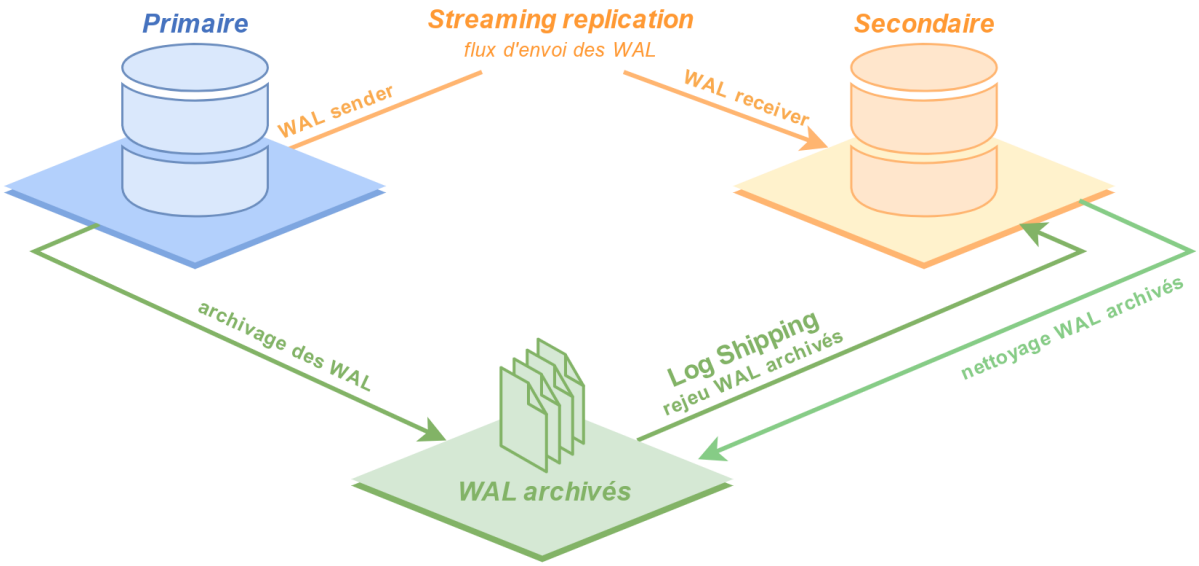
Ce serveur peut toujours remplir le rôle de serveur de secours, tout en étant utilisable pour soulager le primaire : sauvegarde, requêtage en lecture…

1.3.5 Exemple

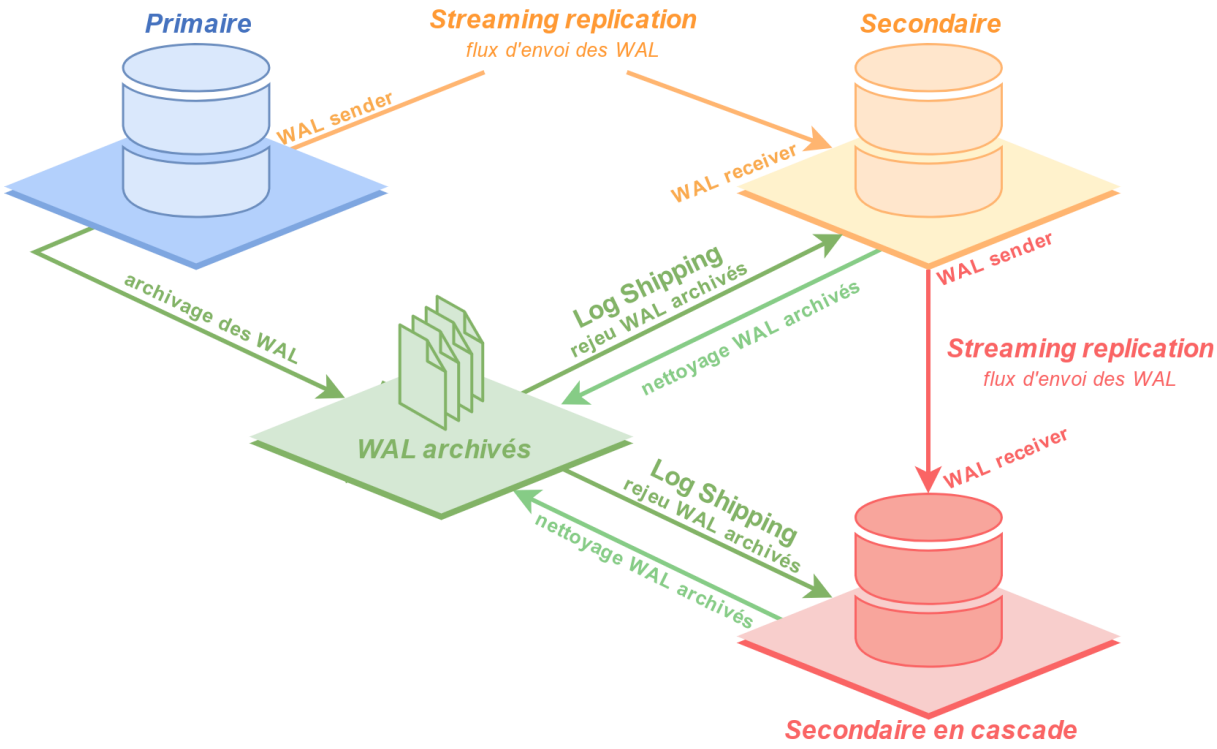


Cet exemple montre un serveur primaire en *streaming replication* vers un serveur secondaire. Ce dernier est configuré en *Hot Standby*. Ainsi, les utilisateurs peuvent se connecter sur le serveur secondaire pour les requêtes en lecture et sur le primaire pour des lectures comme des écritures. Cela permet une forme de répartition de charge des lectures, la répartition étant gérée par le serveur d'applications ou par un outil spécialisé.

1.3.6 Réplication interne



1.3.7 Réplication en cascade



1.4 RÉPLICATION INTERNE LOGIQUE

- Réplique les changements
  - d'une seule base de données
  - d'un ensemble de tables défini
- Principe Éditeur/Abonnés

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 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. 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.

La réplication logique est disponible depuis la version 10 de PostgreSQL. Contrairement à la réplication physique, elle s'effectue entre instances primaires, toutes deux ouvertes en écriture avec leurs tables propres. Rien n'interdit à une instance abonnée pour certaines tables de proposer ses propres publications, même de tables concernées par un abonnement.

1.4.1 Réplication logique - Fonctionnement

- Création d'une publication sur un serveur
- Souscription d'un autre serveur à cette publication
- Limitations :
  - DDL, Large objects, séquences, tables étrangères et vues matérialisées non répliqués
  - peu adaptée pour un *failover*

Une « publication » est créée sur le serveur éditeur et ne concerne que certaines tables. L'abonné souscrit à cette publication, c'est un « souscripteur ». Un processus spécial est lancé : le *logical replication worker*. Il va se connecter à un slot de réplication sur le serveur éditeur. Ce dernier va procéder à un décodage logique de ses propres journaux de transaction pour extraire les résultats des ordres SQL (et non les ordres eux-mêmes). Le flux logique est transmis à l'abonné qui les applique sur les tables.

La réplication logique possède quelques limitations. La principale est que seules les données sont répliquées, c'est-à-dire le résultat des ordres `INSERT`, `DELETE`, `UPDATE`, `TRUNCATE` (sauf en v10 pour ce dernier). Les tables cible doivent être créés manuellement, et il faudra dès lors répliquer manuellement les changements de structure.

Il n'est pas obligatoire de conserver strictement la même structure des deux côtés. Mais, afin de conserver sa cohérence, la réplication s'arrêtera en cas de conflit. Des clés primaires sur toutes les tables concernées sont fortement conseillées. Les *large objects* ne sont pas répliqués. Les séquences non plus, y compris celles utilisées par les colonnes de type `serial`. Notez que pour éviter des effets de bord sur la cible, les triggers des tables abonnées ne seront pas déclenchés par les modifications reçues via la réplication.

En principe, il serait possible d'utiliser la réplication logique en vue d'un *failover* vers un serveur de secours en propageant manuellement les mises à jour de séquences et de schéma. La réplication physique est cependant plus appropriée et plus efficace pour cela.

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

1.5 RÉPLICATION EXTERNE

- Outils les plus connus :
  - Pgpool
  - Slony, Bucardo
  - pgLogical
- Niches

Jusqu'en 2010, PostgreSQL ne disposait pas d'un système de réplication évolué, et plus d'une quinzaine de projets ont tenté de combler ce vide. L'arrivée de la réplication logique en version 10 met à mal les derniers survivants. En pratique, ces outils ne comblent plus que des niches.

La liste exhaustive est trop longue pour que l'on puisse évoquer en détail chacune des solutions, surtout que la plupart sont obsolètes ou ne semblent plus maintenues. Voici les plus connues :

- Slony (réplication par trigger, éprouvé et fiable) ;
- Bucardo (réplication par trigger, toujours maintenu) ;
- Pgpool (réplication d'ordres SQL parmi d'autres fonctionnalités ; toujours actif) ;
- pgLogical (réplication logique, toujours actif) ;
- Londiste ;
- PGCluster ;
- Mammoth Replicator/Replicator ;
- Daffodil Replicator ;
- RubyRep ;
- pg_comparator ;
- Postgres-X2 (ex-Postgres-XC)…

Pour les détails sur ces outils et d'autres, voir le wiki[5] ou cet article : Haute disponibilité avec PostgreSQL[6], Guillaume Lelarge, 2009.

[5] https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
[6] https://public.dalibo.com/exports/conferences/_archives/_2011/201102_haute_disponibilite_avec_postgresql/solutionsha.pdf

1.6 SHARDING

- Répartition des données sur plusieurs instances
- Évolution horizontale en ajoutant des serveurs
- Parallélisation
- Clé de répartition cruciale
- Administration complexifiée
- Sous PostgreSQL :
  - *Foreign Data Wrapper*
  - PL/Proxy
  - Citus (extension), et nombreux *forks*

Le *sharding* n'est pas de la réplication, ce serait même l'inverse. Le principe consiste à répartir les données sur plusieurs instances différentes, chacune étant responsable d'une partie des données, et ouverte en écriture.

La volumétrie peut augmenter, il suffit de rajouter des serveurs. Plusieurs serveurs peuvent travailler en parallèle sur la même requête. On contourne ainsi le principal goulet d'étranglement des performances : les entrées-sorties.

Le problème fondamental est la clé de répartitions des données sur les différents serveurs. Un cas simple est la répartition des données de nombreux clients dans plusieurs instances, chaque client n'étant présent que dans une seule instance. On peut aussi opérer une sorte de *hash* de la clé pour répartir équitablement les données sur les serveurs. Il faut aviser en fonction de la charge prévue, de la nécessité d'éviter les conflits lors des mises à jour, du besoin de croiser les données en masse, des purges à faire de temps à autre, et de la manière de répartir harmonieusement les écritures et lectures entre ces instances. C'est au client ou à une couche d'abstraction de savoir quel(s) serveur(s) interroger.

PostgreSQL n'implémente pas directement le *sharding*. Plusieurs techniques et outils permettent de le mettre en place.

- Les *Foreign Data Wrappers* (et l'extension `postgres_fdw` en particulier) permettent d'accéder à des données présentes sur d'autres serveurs. La capacité de `postgres_fdw` à « pousser » filtres et jointures vers les serveurs distants s'améliore de version en version. Des tables distantes peuvent être montées en tant que partitions d'une table mère. Les insertions dans une table partitionnée peuvent même être redirigées vers une partition distante de manière transparente. Vous trouverez un exemple dans cet article[7] ou à la fin du module de formation V1[8] Le

[7] https://pgdash.io/blog/postgres-11-sharding.html
[8] https://dali.bo/v1_html#tables-distantes-sharding

parcours simultané des partitions distantes est même possible à partir de PostgreSQL 14[9]. La réplication logique peut synchroniser des tables non distribuées sur les instances.

- PL/Proxy est une extension qui permet d'appeler plusieurs hôtes distants à la fois avec un seul appel de fonctions. Elle existe depuis des années. Son inconvénient majeur est la nécessité de réécrire tous les appels à distribuer par des fonctions, on ne peut pas se reposer sur le SQL de manière transparente.
- Citus[10] est une extension libre dont le but est de rendre le *sharding* transparent, permettant de garder la compatibilité avec le SQL habituel. Des nœuds sont déclarés auprès du serveur principal (où les clients se connectent), et quelques fonctions permettent de déclarer les tables comme distribuées selon une clé (et découpées entre les serveurs) ou tables de références (dupliquées partout pour faciliter les jointures). Les requêtes sont redirigées vers le bon serveur selon la clé, ou réellement parallélisées sur les différents serveurs concernés. Le projet est vivant, bien documenté, et a bonne réputation. Depuis son rachat par Microsoft en 2019, Citus assure ses revenus grâce à une offre disponible sur le cloud Azure. Ceci a permis en 2022 de libérer les fonctionnalités payantes et de publier l'intégralité projet en open-source.

Le *sharding* permet d'obtenir des performances impressionnantes, mais il a ses inconvénients :

- plus de serveurs implique plus de sources de problèmes, de supervision, de tâches d'administration (chaque serveur a potentiellement son secondaire, sa réplication…) ;
- le modèle de données doit être adapté au problème, limitant les interactions d'un nœud avec les autres ; le choix de la clé est crucial ;
- les propriétés ACID et la cohérence sont plus difficilement respectées dans ces environnements.

Historiquement, plusieurs *forks* de PostgreSQL ont été développés en partie pour faire du *sharding*, principalement en environnement OLAP/décisionnel, comme PostgreSQL-XC/XL[11], à présent disparu, ou Greenplum[12], qui existe toujours. Ces *forks* ont plus ou moins de difficultés à suivre la rapidité d'évolution de la version communautaire : les choisir implique de se passer de certaines nouveautés. D'où le choix de Citus de la forme d'une extension.

Ce domaine est l'un de ceux où PostgreSQL devrait beaucoup évoluer dans les années à venir, comme le décrivait Bruce Momjian en septembre 2018[13].

[9] https://dali.bo/workshop14_html#lecture-asynchrone-des-tables-distantes
[10] https://www.citusdata.com/product
[11] https://www.postgres-xl.org/
[12] https://greenplum.org/
[13] https://momjian.us/main/writings/pgsql/sharding.pdf

1.7 RÉPLICATION BAS NIVEAU

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

Même si cette présentation est destinée à détailler les solutions logicielles de réplication pour PostgreSQL, on peut tout de même évoquer les solutions de réplication de « bas niveau », voire matérielles.

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

1.7.1 RAID

- Obligatoire
- Fiabilité d'un serveur
- RAID 1 ou RAID 10
- RAID 5 déconseillé (performances)
- Lectures plus rapides
    - dépend du nombre de disques impliqués

Un système RAID 1 ou RAID 10 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 serveurs mais cela contribue à la haute-disponibilité du système.

Le RAID 5 offre les mêmes avantages en gaspillant moins d'espace qu'un RAID 1, mais il est déconseillé pour les bases de données (PostgreSQL comme ses concurrents) à cause des performances en écriture, au quotidien comme lors de la reconstruction d'une grappe après remplacement d'un disque.

Le système RAID 10 est plus intéressant pour les fichiers de données alors qu'un système RAID 1 est suffisant pour les journaux de transactions.

Le RAID 0 (répartition des écritures sur plusieurs disques sans redondance) est évidemment à proscrire.

1.7.2 DRBD

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

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[14] de Guillaume Lelarge dans Linux Magazine Hors Série 45.

DRBD est un système simple à mettre en place. Son gros avantage est la possibilité d'avoir une réplication synchrone. Ses inconvénients sont sa lenteur, la non-disponibilité des secondaires et un volume de données plus important à répliquer (WAL + tables + index + vues matérialisées…).

1.7.3 SAN Mirroring

- Comparable à DRBD
- Solution intégrée
- Manque de transparence

La plupart des constructeurs de baie de stockage proposent des systèmes de réplication automatisés avec des mécanismes de *failover*/*failback* parfois sophistiqués. Ces solutions présentent peu ou prou les mêmes caractéristiques, avantages et inconvéniants que DRBD. Ces technologies ont en revanche le défaut d'être opaques et de nécessiter une main d'œuvre hautement qualifiée.

1.8 CONCLUSION

Quelle que soit la solution envisagée :

- Bien définir son besoin
- Identifier tous les *SPOF*
- Superviser son *cluster*
- Tester régulièrement les procédures de *failover* (Loi de Murphy…)

Bibliographie :

- « Haute disponibilité, répartition de charge et réplication[15] » (documentation officielle)

1.8.1 Questions

N'hésitez pas, c'est le moment !

[15] https://docs.postgresql.fr/current/high-availability.html

1.9 QUIZ

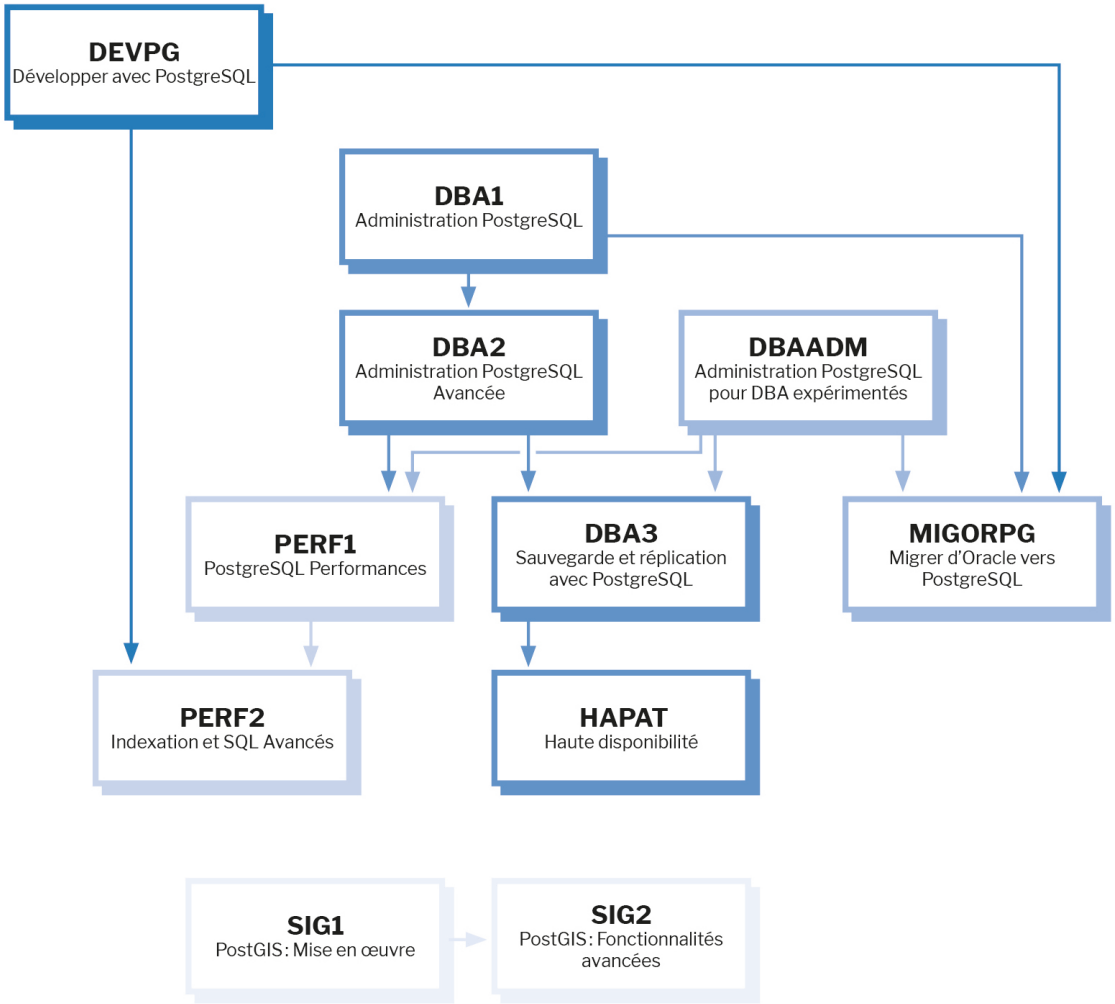
https://dali.bo/w1_quiz

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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