Module W2B

Réplication Physique avancée

24.04

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ Réplication physique avancée **5**
1.1 Introduction . . . 6
1.1.1 Au menu . . . 6
1.2 Supervision (streaming) . . . 7
1.2.1 Utilitaires pour le *streaming* . . . 7
1.2.2 pg_stat_replication . . . 8
1.2.3 Autres vues pour le streaming . . . 10
1.3 Supervision (log shipping) . . . 12
1.4 Conflits de réplication . . . 13
1.4.1 Détection des conflits de réplication . . . 13
1.4.2 Prévenir les conflits de réplication . . . 14
1.5 Contrôle de la réplication . . . 16
1.6 Réplication synchrone . . . 17
1.6.1 Secondaires synchrones . . . 18
1.6.2 Niveau de synchronicité & performances . . . 19
1.7 Réplication en cascade . . . 24
1.8 Décrochage d'un secondaire . . . 25
1.8.1 Sécurisation par log shipping . . . 26
1.8.2 Slot de réplication : mise en place . . . 27
1.8.3 Slot de réplication : avantages & risques . . . 28
1.9 Synthèse des paramètres . . . 31
1.9.1 Serveur primaire . . . 31
1.9.2 Serveur secondaire . . . 32
1.10 Conclusion . . . 33
1.10.1 Questions . . . 33
1.11 Quiz . . . 34
1.12 Travaux pratiques . . . 35
1.12.1 Réplication asynchrone en flux avec deux secondaires . . . 35
1.12.2 Slots de réplication . . . 35
1.12.3 Log shipping . . . 36
1.12.4 Réplication synchrone en flux avec trois secondaires . . . 36
1.12.5 Réplication synchrone : cohérence des lectures (optionnel) . . . 37

1.13 Travaux pratiques (solutions) 39
1.13.1 Réplication asynchrone en flux avec deux secondaires 39
1.13.2 Slots de réplication 40
1.13.3 Log shipping 43
1.13.4 Réplication synchrone en flux avec trois secondaires 46
1.13.5 Réplication synchrone : cohérence des lectures (optionnel) 51

Les formations Dalibo **55**
Cursus des formations 55
Les livres blancs 56
Téléchargement gratuit 56

Sur ce document

Formation	Module W2B
Titre	Réplication Physique avancée
Révision	24.04
PDF	https://dali.bo/w2b_pdf
EPUB	https://dali.bo/w2b_epub
HTML	https://dali.bo/w2b_html
Slides	https://dali.bo/w2b_slides
TP	https://dali.bo/w2b_tp
TP (solutions)	https://dali.bo/w2b_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

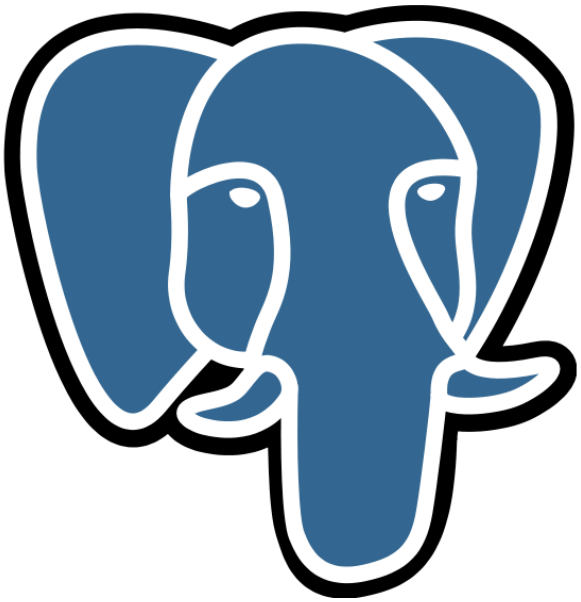
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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

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

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

1/ Réplication physique avancée



1.1 INTRODUCTION

- Supervision
- Fonctionnalités avancées

En complément du module précédent, il est important de bien superviser un cluster en réplication, d'en connaître les limites mais aussi d'appréhender toutes les possibilités offertes par la réplication physique.

1.1.1 Au menu

- Supervision
- Gestion des conflits
- Asynchrone ou synchrone
- Réplication en cascade
- Slot de réplication
- Log shipping

1.2 SUPERVISION (STREAMING)

- Quelles vues et fonctions utilitaires ?
- Comment voir et calculer le retard des secondaires ?

1.2.1 Utilitaires pour le *streaming*

- `pg_is_in_recovery()` : instance en réplication ?
- Calcul du retard en octets :

```
-- primaire
SELECT pg_wal_lsn_diff ( pg_current_wal_lsn(), '0/73D3C1F0' );
```

- et en temps

```
-- secondaire
SELECT now() - pg_last_xact_replay_timestamp() ; -- si activité
```

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

Retard en octets :

Le calcul de la différence de volumétrie de données entre le primaire et ses secondaires (*lag*) peut-être effectué avec la fonction `pg_wal_lsn_diff()`. La fonction `pg_current_wal_lsn()` fournit la position dans le flux de données du primaire. En récupérant la position sur le secondaire (au choix, dernière réception avec `pg_last_wal_receive_lsn()` ou dernier rejeu avec `pg_last_wal_replay_lsn()`), le calcul du *lag* en octet devient :

```
-- sur le secondaire
SELECT  pg_last_wal_replay_lsn () ;

 pg_last_wal_replay_lsn
------------------------
 13/A7DD670

-- sur le primaire
SELECT pg_size_pretty (
```

```
  pg_wal_lsn_diff( pg_current_wal_lsn(), '13/A7DD670')
  ) ;

 pg_size_pretty
----------------
 1939 kB
```

Mais nous allons voir qu'il y a plus pratique.

Retard en durée :

Quand le retard d'un serveur secondaire sur son primaire est exprimé en octets, il n'est pas simple d'en appréhender l'amplitude. Le retard en durée est plus parlant. La fonction `pg_last_xact_replay_timestamp()` indique 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 au rejeu d'un serveur secondaire sous la forme d'une durée.

```
SELECT now() - pg_last_xact_replay_timestamp() ; -- si activité
```

Attention, si le primaire ne reçoit que des transactions en lecture, le flux de journaux n'est pas forcément complètement vide, mais `pg_last_xact_replay_timestamp()` ne s'incrémente alors pas sur le secondaire ! `now() - pg_last_xact_replay_timestamp()` donnera alors une durée croissante dans le temps, même si le serveur secondaire n'a aucun retard.

1.2.2 pg_stat_replication

Type de réplication & *lag* des secondaires :

```
SELECT * FROM pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
pid              | 286511
usesysid         | 10
usename          | postgres
application_name | secondaire2
client_addr      | 192.168.0.55
client_hostname  |
client_port      | -1
backend_start    | 2023-12-19 10:41:47.431471+01
backend_xmin     |
state            | streaming
sent_lsn         | 14/C402A000
write_lsn        | 14/C402A000
flush_lsn        | 14/C402A000
replay_lsn       | 14/C311D460
write_lag        | 00:00:00.032183
flush_lag        | 00:00:00.032601
replay_lag       | 00:00:02.984354
sync_priority    | 1
sync_state       | sync
reply_time       | 2023-12-19 11:05:37.903584+01
```

Pour connaître l'état des différents serveurs secondaires connectés au serveur primaire, le plus simple est de récupérer les informations provenant de la vue `pg_stat_replication` du primaire. Elle permet de connaître l'état de tous les serveurs secondaires connectés en *streaming* (mais pas ceux déconnectés !). Il y a une ligne pour chacun d'entre eux, l'exemple ci-dessus porte donc sur un seul secondaire. La plupart des colonnes se comprennent aisément.

L'adresse IP du serveur est l'information principale pour distinguer les secondaires s'il y en a plusieurs.

`application_name` peut être fourni par le secondaire dans sa chaîne de connexion `primary_conninfo`. Il est conseillé de le renseigner pour la supervision, (ou bien `cluster_name`).

`state` est à `streaming` quand tout va bien. Quand un secondaire vient de se connecter, `state` affiche `catchup` le temps de revenir au moins une fois à un *lag* nul.

`sync_state` vaut `async` dans le cas d'une réplication asynchrone. Avec une réplication synchrone, ce sera `sync`, `potential` ou `quorum`. Si la connection a échoué, la ligne n'existe simplement pas.

`backend_start` indique l'heure de connexion, et `reply_time` l'heure du dernier message envoyé par le secondaire.

Les quatre LSN permettent de suivre la réception, l'enregistrement et le rejeu sur le secondaire, grâce aux fonctions évoquées plus haut :

```
SELECT application_name,
  pg_size_pretty(pg_wal_lsn_diff( pg_current_wal_lsn(), replay_lsn )) AS retard_rejeu
FROM pg_stat_replication ;
```

```
 application_name | retard_rejeu
------------------+----------------
 secondaire3      | 15 MB
 secondaire2      | 0 bytes
```

Le service **streaming_delta** de la sonde check_pgactivity[1] ne fait pas autrement pour suivre les volumétries à recevoir, à appliquer et à rejouer.

Les différents champs `*_lag` indiquent le retard temporel des secondaires, ce qui est très pratique pour repérer un secondaire en retard ou en pause.

`write_lag` mesure le délai entre l'enregistrement dans les journaux en local et la notification de l'enregistrement dans le cache disque du secondaire (ce délai est important en mode synchrone avec `synchronous_commit` à `remote_write`) mais sans attendre l'écriture physique (`sync`).

`flush_lag` mesure le délai jusqu'à confirmation que les données modifiées soient bien écrites sur disque au niveau du serveur *standby* (ce délai est celui à suivre en mode synchrone avec `synchronous_commit` à `on`).

`replay_lag` mesure le délai jusqu'au rejeu des transactions sur le secondaire, les rendant visibles aux requêtes des utilisateurs (ce délai est à surveiller si `synchronous_commit` est à `remote_apply`)

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

La sortie d'écran plus haut indique que la réception des données sur le secondaire est rapide, mais le rejeu a 3 secondes de retard.

1.2.3 Autres vues pour le streaming

- S'il y un slot
  - `pg_replication_slots`
- Sur le secondaire
  - `pg_stat_wal_receiver`

pg_replication_slots :

Toujours depuis le primaire, pour savoir où en sont les serveurs secondaires, éventuellement déconnectés, utilisant un slot de réplication, consulter aussi la vue `pg_replication_slots` :

```
SELECT * FROM pg_replication_slots ;
```

```
-[ RECORD 1 ]-------+--------------
slot_name           | secondaire1
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         | 0/A5F8310
confirmed_flush_lsn |
safe_wal_size       | 5374099280
two_phase           | f
conflicting         |
-[ RECORD 2 ]-------+--------------
slot_name           | secondaire2
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 29287
xmin                |
catalog_xmin        |
restart_lsn         | 0/AEC3B40
```

```
confirmed_flush_lsn |
safe_wal_size       | 5374099280
two_phase           | f
conflicting         |
```

pg_stat_wal_receiver :

Sur le secondaire, on peut consulter aussi la vue `pg_stat_wal_receiver` pour voir la connexion en cours :

```
SELECT * FROM pg_stat_wal_receiver \gx
```

```
-[ RECORD 1 ]---------+------------------------------------------------------------
pid                   | 696088
status                | streaming
receive_start_lsn     | 14/AC000000
receive_start_tli     | 1
written_lsn           | 15/78CB1F8
flushed_lsn           | 15/78CB1F8
received_tli          | 1
last_msg_send_time    | 2023-12-19 12:05:45.275257+01
last_msg_receipt_time | 2023-12-19 12:05:45.275532+01
latest_end_lsn        | 15/78CB1F8
latest_end_time       | 2023-12-19 12:05:45.273271+01
slot_name             | secondaire3
sender_host           | /var/run/postgresql
sender_port           | 5432
conninfo              | user=postgres passfile=/var/lib/postgresql/.pgpass
 ↪  channel_binding=prefer dbname=replication host=/var/run/postgresql port=5432
 ↪  application_name=secondaire3 fallback_application_name=16/secondaire3
 ↪  sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
 ↪  ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres
 ↪  gssdelegation=0 target_session_attrs=any load_balance_hosts=disable
```

Noter que le `primary_conninfo` d'origine est complété de nombreux paramètres par défaut.

1.3 SUPERVISION (LOG SHIPPING)

- Le primaire ne sait rien
- Supervision de l'archivage comme pour du PITR
  - `pg_stat_archiver`
- Secondaire :
  - `pg_wal_lsn_diff()`
  - traces
  - calcul du retard manuel (`pg_last_wal_replay_lsn()`)

Si le secondaire est en *log shipping* (par choix ou parce que le secondaire a trop de retard et a basculé dans ce mode), la supervision est plus compliquée.

Le primaire étant déconnecté du secondaire, `pg_stat_replication` ne contiendra rien sur ce secondaire.

Côté primaire, on vérifiera que l'archivage se fait correctement, notamment avec la vue `pg_stat_archiver`.

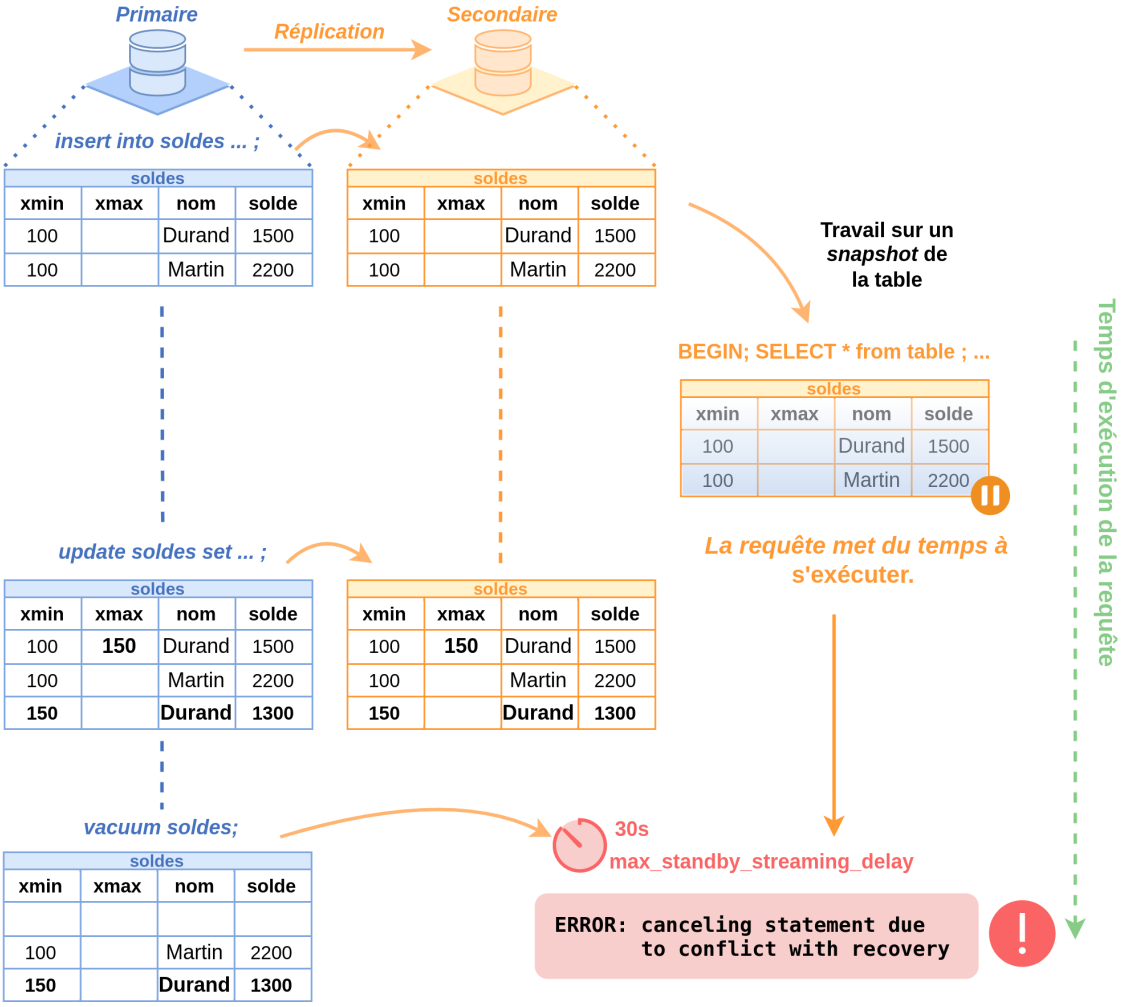
Côté secondaire, les traces permettent de vérifier que les journaux sont récupérés et appliqués, ou de connaître la cause des erreurs : `restore_command` mal paramétrée, problème d'accès aux journaux, etc.

Le calcul du retard ci-dessus reste possible, mais il faudra aller chercher où est le secondaire dans le flux des WAL en y exécutant la fonction `pg_last_wal_replay_lsn()`.

Le service **hot_standby_delta** de la sonde check_pgactivity[2] facilite cela : elle se connecte au primaire et au secondaire, et calcule l'écart, pour lever une alerte au besoin. Il peut être utile de la déployer même sur une instance habituellement en *streaming* pour suivre un rattrapage long.

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

1.4 CONFLITS DE RÉPLICATION



1.4.1 Détection des conflits de réplication

- Une requête en lecture pose des verrous
  - conflit possible avec changements répliqués !
- Vue `pg_stat_database_conflicts` (secondaires)
- Traces :
  - `log_recovery_conflict_waits` (v14+)

Source des conflits :

Sur un primaire, le MVCC garantit qu'une requête ne sera pas gênée si elle lit des lignes dans des blocs qu'une autre requête est en train de modifier.

Mais le primaire ne sait à priori rien des requêtes sur un secondaire. Sur ce dernier, un conflit peut survenir entre l'application des modifications provenant du primaire d'une part, et l'exécution d'une requête (en lecture seule) d'autre part.

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 ce serveur secondaire.

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.

Suivi des conflits :

La table `pg_stat_database_conflicts` du catalogue système n'est renseignée que sur les serveurs secondaires d'une réplication. Elle contient le nombre de conflits détectés sur ce secondaire par type de conflit (conflit sur un tablespace, conflit sur un verrou, etc.). Elle contient une ligne par base de données :

```
SELECT * FROM pg_stat_database_conflicts
WHERE datname='postgres' ;
```

```
-[ RECORD 1 ]----+---------
datid            | 12857
datname          | postgres
confl_tablespace | 0
confl_lock       | 0
confl_snapshot   | 3
confl_bufferpin  | 2
confl_deadlock   | 0
```

Le total se retrouve dans `pg_stat_database`.

En version 14 apparaît le paramètre `log_recovery_conflict_waits`. Son activation est conseillée. Il permet de tracer toute attente due à un conflit de réplication. Il n'est donc valable et pris en compte que sur un serveur secondaire.

1.4.2 Prévenir les conflits de réplication

- `wal_standby_streaming_delay`
- `hot_standby_feedback` à `on` + `wal_receiver_status_interval` (10s)
- gênent le vacuum !

Gestion fine des délais pour réduire les conflits :

Avant d'appliquer un journal, ou extrait de journal, qui entre en conflit avec des requêtes en cours sur le secondaire, PostgreSQL attend un certain délai. `max_standby_archive_delay` et `max_standby_streaming_delay` sont des délais d'attente provenant respectivement du *log shipping* ou du *streaming*, Par défaut, c'est 30 secondes. Monter l'un ou l'autre de ces paramètres peut être suffisant si l'on peut tolérer que la réplication soit brièvement bloquée.

Avant la version 16, il existait un paramètre `vacuum_defer_cleanup_age` qui demande au `VACUUM` d'attendre un certain nombre de transactions avant de recycler des lignes mortes. Ce paramètre a été supprimé car le calcul était délicat, et il générait de la fragmentation. Il était désactivé par défaut.

hot_standby_feedback :

De manière plus fine (et plus simple), les serveurs secondaires peuvent envoyer des informations au serveur primaire sur les requêtes en cours d'exécution, pour tenter de prévenir au moins les conflits lors du nettoyage des enregistrements (action effectuée par le `VACUUM`). Il faut pour cela activer le paramètre `hot_standby_feedback` (à `off` par défaut). Le serveur secondaire envoie alors des informations au serveur primaire à une certaine fréquence, configurée par le paramètre `wal_receiver_status_interval`, soit 10 secondes par défaut.

Ces paramètres doivent être maniés avec précaution, car ils peuvent causer une fragmentation des tables sur le serveur primaire, certes pas forcément plus importante que si les requêtes sur le secondaire avaient été lancées sur le primaire. Attention notamment s'il y a un slot de réplication et `hot_standby_feedback` à `on` (voir plus bas) !

Gestion des déconnexions du secondaire

Grâce à cet envoi d'informations, PostgreSQL peut savoir si un serveur secondaire est indisponible, par exemple suite à une coupure réseau ou à un arrêt brutal du serveur secondaire. Rappelons que si jamais le serveur secondaire est indisponible, le primaire coupe la connexion avec le secondaire après un temps déterminé par le paramètre `wal_sender_timeout` (1 minute par défaut), Pour éviter des coupures intempestives, il faut donc conserver `wal_receiver_status_interval` à une valeur inférieure à celle de `wal_sender_timeout`.

1.5 CONTRÔLE DE LA RÉPLICATION

- `pg_wal_replay_pause()` : mettre en pause le **rejeu**
- `pg_wal_replay_resume()` : reprendre
- `pg_is_wal_replay_paused()` : statut
- Utilité :
  - requêtes longues
  - `pg_dump` depuis un secondaire

Lancer un `pg_dump` depuis un serveur secondaire est souvent utile pour ne pas charger le primaire, mais ce 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 (notamment `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 avec les fonctions suivantes :

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

Ces fonctions s'exécutent uniquement sur les serveurs secondaires et la réplication n'est en pause que sur le serveur secondaire où la fonction est exécutée. Il est donc possible de laisser la réplication en exécution sur certains secondaires et de la mettre en pause sur d'autres.

Plus généralement, cette technique est applicable pour figer des secondaires et y effectuer de très longues requêtes qui n'ont pas besoin des données les plus récentes enregistrées sur le primaire.

Noter qu'il s'agit bien de figer le *rejeu* des journaux, pas leur transmission. Le serveur secondaire ainsi figé stocke les journaux et pourra les réappliquer plus tard. Même une réplication synchrone, dans sa version la moins stricte, reste ainsi possible.

1.6 RÉPLICATION SYNCHRONE

Le primaire attend l'enregistrement sur le secondaire.

- Comment configurer ?
- Comment limiter l'impact sur les performances ?

La réplication synchrone est fréquemment demandée sur tous les moteurs de bases de données.

En réplication **a**synchrone, quand une transaction est validée, le serveur primaire rend la main à l'utilisateur lorsqu'il a fini d'enregistrer les données dans ses journaux de transactions sur disque. Il n'attend donc pas de savoir si le serveur secondaire a reçu les données, et encore moins si elles sont enregistrées sur son disque. Le problème survient quand le serveur primaire s'interrompt soudainement et qu'il faut basculer le serveur secondaire en serveur primaire. Les dernières données enregistrées sur le serveur primaire n'ont peut-être pas eu le temps d'arriver sur le serveur secondaire. Par conséquent, on peut se trouver dans une situation où le serveur indique une transaction comme enregistrée, alors qu'après le *failover* elle n'est plus visible.

Avec une réplication synchrone, le serveur primaire ne valide la transaction auprès de l'utilisateur qu'à partir du moment où le serveur secondaire synchrone a lui aussi reçu/écrit/rejoué la donnée sur disque (selon le mode). Le premier intérêt de la réplication synchrone est donc de s'assurer qu'en cas de *failover*, aucune donnée ne soit perdue. Le second intérêt peut être d'avoir des serveurs secondaires renvoyant exactement la même chose au même moment que le primaire.

L'immense inconvénient de la réplication synchrone est la latence supplémentaire due aux échanges entre les serveurs pour chaque transaction. En effet, il ne faut pas seulement attendre que le serveur primaire fasse l'écriture, il faut aussi attendre l'écriture sur le serveur secondaire sans parler des interactions et des latences réseau. Même si le coût semble minime, il reste cependant présent, et dépend aussi de la qualité du réseau : la durée d'un aller-retour réseau est souvent du même ordre de grandeur (milliseconde) que bien des petites transactions, voire plus élevée. Pour des serveurs réalisant beaucoup d'écritures, le coût n'en sera que plus grand.

Même si le mode peut se choisir transaction par transaction, noter que la réplication d'une transaction synchrone doit attendre la réception, voire le rejeu, de toutes les transactions précédentes. Une grosse transaction asynchrone peut donc ralentir la transmission ou le rejeu de transactions synchrones.

Enfin, la réplication synchrone a un autre danger : si le serveur synchrone ne répond pas, la transaction ne sera pas validée sur le primaire. Du point de vue de l'application, un `COMMIT` ne rendra pas la main. PostgreSQL permet de déclarer plusieurs serveurs synchrones pour réduire le risque.

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 intolérable. Nous allons voir les différentes options pour limiter les inconvénients.

1.6.1 Secondaires synchrones

- Par défaut : réplication physique **a**synchrone
- Secondaires synchrones :

```
# s1 synchrone, s2 en dépannage
synchronous_standby_names = 'FIRST 1 (s1, s2)'
# 2 synchrones au moins
synchronous_standby_names = 'ANY 2 (s1,s2,s3)'
# n'importe quel secondaire
synchronous_standby_names = '*'
```

- Plusieurs synchrones simultanés possibles
  - ou un quorum

Par défaut, la réplication fonctionne en asynchrone. La mise en place d'un mode synchrone est très simple.

Sur le(s) secondaire(s) synchrone(s) :

Il n'y a rien à configurer de plus. Par contre il est crucial de sécuriser la disponibilité de l'instance.

Sur le primaire :

Comme le paramètre `synchronous_commit` est déjà à `on` par défaut, il ne reste qu'à déclarer les serveurs secondaires synchrones avec le paramètre `synchronous_standby_names`, en séparant par des virgules les différentes instances secondaires synchrones. Il est possible d'indiquer le nombre de serveurs synchrones simultanés. Les serveurs surnuméraires sont des synchrones potentiels.

Pour que `s1` soit un secondaire synchrone, et que `s2` et `s3` le deviennent si `s1` ne répond pas, on a plusieurs syntaxes au choix :

```
synchronous_standby_names = '1 (s1,s2,s3)'
synchronous_standby_names = 'FIRST 1 (s1, s2, s3)'
-- syntaxe à ne plus utiliser
synchronous_standby_names = 's1,s2,s3'
```

Dans l'exemple suivant, `s1` et `s2` seront tous les deux synchrones, `s3` ne le sera pas, sauf défaillance d'un des premiers.

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

`*` remplace la liste des secondaires :

```
# un secondaire désigné synchrone dans la liste, les autres en secours
synchronous_standby_names = '1 (*)'
```

```
synchronous_standby_names = 'FIRST 1(*)'
synchronous_standby_names = '*'
```

Il est possible de se baser sur un quorum. Par exemple, pour que la transaction synchrone s'achève dès que 2 serveurs sur les 3 indiqués l'ont enregistrée, et quels qu'il soient, on écrira :

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

Si l'on ne veut pas spécifier les secondaires manuellement, cette syntaxe est très pratique :

```
synchronous_standby_names = 'ANY 2 (*)'
```

Il est parfois nécessaire d'utiliser des guillemets droits :

```
synchronous_standby_names = 'ANY 2 (sec1,"sec-2","sec 3")'
```

La comparaison entre l'`application_name` des connexions de réplication et la liste de serveurs spécifiée dans `synchronous_standby_names` n'est pas sensible à la casse, que l'on utilise des guillemets droits ou non. Il n'y a pas de validation des noms. En cas de faute de frappe, PostgreSQL cherchera donc à être synchrone avec un serveur non connecté, ce qui va bloquer les transactions.

S'il existe des serveurs secondaires non listés dans `synchronous_standby_names`, ils seront implicitement répliqués de manière asynchrone, donc sans impact sur les performances.

Mais comment indiquer le nom d'un serveur secondaire ? Ce nom dépend d'un paramètre de connexion appelé `application_name`, que le client définit librement. Il doit apparaître dans la chaîne de connexion du serveur secondaire au serveur primaire, c'est-à-dire `primary_conninfo`, et différer pour chaque secondaire. Par exemple :

```
primary_conninfo = 'user=user_repli host=prod application_name=s2'
```

Sur le primaire, le nom apparaîtra dans la vue `pg_stat_replication`, champ `application_name`. Ce nom est indépendant de l'éventuel slot de réplication (`primary_slot_name`), même s'ils sont souvent identiques.

1.6.2 Niveau de synchronicité & performances

- Niveau de synchronicité :

```
SET synchronous_commit = off / local / remote_write / on / remote_apply
```

- Ajustable par base/utilisateur/session/transaction
- Risque de blocage du primaire à cause des secondaires !

Pour définir le mode de fonctionnement exact, `synchronous_commit` peut prendre plusieurs valeurs. En ordre croissant de sécurité, ce sont les suivantes :

off :

La transaction est directement validée dans le cache du serveur primaire, mais elle sera être écrite plus tard dans les journaux et sur le disque. Évidement, les secondaires ne sont pas synchrones non plus.

Ce paramétrage peut causer la perte des transactions non encore écrites dans les journaux si le serveur se crashe. La durée d'activité potentiellement perdue est d'au maximum 3 fois la valeur de `wal_writer_delay` (soit au total 0,6 s par défaut). Par contre, il n'y a pas de risque de corruption.

Même sans réplication, `synchronous_commit = off` offre de gros gains de performance dans le cas de nombreuses petites transactions,

C'est à savoir pour tous les cas où la perte des dernières transactions validées ne porte pas à conséquence grave (au milieu d'un batch que l'on relancera par exemple). On réduit en effet l'impact en performance de l'opération de synchronisation sur disque des journaux, sans risquer de corruption de données.

local :

On force le mode asynchrone. La transaction est validée 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.

S'il n'y a pas de secondaire synchrone, `on` et `local` sont équivalents.

Si le primaire disparaît, il peut y avoir perte de transactions validées et non reçues par un secondaire.

remote_write :

Le primaire synchronise ses journaux, bien sûr, et attend que les journaux soient écrits sur le disque du secondaire via le système d'exploitation, mais sans avoir demandé le vidage du cache système sur disque (`fsync`). Les informations sont donc écrites sur le disque du primaire, mais uniquement dans la mémoire système du secondaire.

Il est donc possible de perdre des données si l'instance secondaire crashe en même temps que le primaire.

`remote_write` impacte beaucoup moins les performances que la valeur `on`, et la fenêtre de perte de données est bien moins importante que le mode asynchrone, mais toujours présente.

L'instance primaire ne s'assure pas non plus que le secondaire a **rejoué** la transaction. Le rejeu des journaux peut effectivement durer un certain temps. Deux requêtes exécutées au même moment sur

le primaire et un secondaire peuvent renvoyer des résultats différents. Ce peut être important dans certains cas.

Le délai que `remote_write` impose se mesure dans `pg_stat_replication`, champ `write_lag`.

on (défaut) :

Sans réplication synchrone, il s'agit du fonctionnement normal, où les journaux de transaction sont synchronisés sur disque (`fsync`) avant que la transaction soit considérée comme validée.

Avec des secondaires synchrones, PostgreSQL attend que l'enregistrement associé au COMMIT soit écrit durablement dans les journaux de transactions des instances primaire **et** secondaire(s). L'impact en performances est donc assez lourd.

Il n'y a donc pas de perte de données en cas de crash (sauf pertes des disques des **deux**, ou plus, ou des machines).

La sécurité étant assurée par l'enregistrement des journaux, le primaire n'attend pas que le secondaire ait réellement rejoué les données pour rendre la main à son client. Le secondaire peut accuser un certain retard (voire avoir mis le rejeu de pause). Là encore, deux requêtes exécutées au même moment sur le primaire et un secondaire peuvent renvoyer des résultats différents.

Le délai que `synchronous_commit` à `on` impose se mesure dans `pg_stat_replication`, champ `flush_lag`.

remote_apply :

C'est le mode de synchronisation le plus poussé. Non seulement les modifications doivent être enregistrées dans les journaux du secondaire, et synchronisées sur son disque, mais le secondaire doit les avoir rejouées pour que PostgreSQL confirme la validation de la transaction au client.

Cette méthode est la seule garantissant qu'une transaction validée sur le serveur primaire sera visible sur le secondaire. Évidemment, elle rajoute encore une latence supplémentaire.

`remote_apply` n'est pas une garantie absolue que les serveurs primaire et secondaires renverront tous la même information au même moment : si un secondaire ne répond pas ou a du retard, la session sera bloquée sur le primaire, et son résultat n'y sera pas encore visible ; mais les secondaires qui fonctionnent bien auront déjà rejoué les données modifiées et les afficheront ! Il n'y a aucune synchronisation entre différents secondaires, et un secondaire ne peut pas savoir que le primaire attend un autre secondaire avant de valider la transaction. Ce problème est rare car une réplication synchrone est à éviter sur une liaison instable.

Le délai que `remote_apply` entraîne se mesure dans `pg_stat_replication`, champ `replay_lag`.

Tableau récapitulatif :

Table 1/ .1: Durée de retour du `COMMIT` de synchronisation des données lors de la validation d'une transaction, et facteur principal de cette durée, selon le paramètre `synchronous_commit`

	Durée	Facteur contraignant (primaire)	Facteur contraignant (secondaire)
off	0	Aucun	Aucun
local	Selon disque	Écriture dans `pg_wal`	Aucun
remote_write	`write_lag`	(idem)	Écriture dans la RAM du secondaire
on	`flush_lag`	(idem)	Écriture dans `pg_wal` du secondaire
remote_apply	`write_lag`	(idem)	Rejeu des données en RAM du secondaire

Les valeurs en `*_lag` sont des champs de `pg_stat_replication`.

Synchronicité différente suivant les cas :

`synchronous_commit` peut être défini dans `postgresql.conf` bien sûr, mais aussi par utilisateur, par base, par utilisateur, par session, voire par transaction :

```
ALTER ROLE batch_user SET synchronous_commit TO off ;
ALTER DATABASE audit  SET synchronous_commit TO local ;
SET synchronous_commit TO on ;        -- dans la session
SET LOCAL synchronous_commit TO remote_apply ;  -- dans la transaction
```

Il est conseillé de n'utiliser la synchronisation que pour les modifications les plus importantes et vitales, et la désactiver pour les cas où la performance en écriture prime, ou si vous pouvez relancer l'opération en cas de crash. À vous de définir la bonne valeur par défaut pour `synchronous_commit`, selon les données, les utilisateurs, les applications, et bien sûr l'impact sur les performances.

Par contre, pour modifier `synchronous_standby_names`, il vous faudra modifier `postgresql.conf` ou passer par `ALTER SYSTEM`, puis recharger la configuration.

En cas de problème :

Il faut savoir qu'en cas d'indisponibilité du ou des secondaire(s) synchrone(s), pour que des transactions synchrones bloquées puissent se terminer, le plus simple est de retirer le secondaire problématique de `synchronous_standby_names` depuis une autre session :

```
SHOW synchronous_standby_names ;

 synchronous_standby_names
---------------------------
 2 (s2,s3)

-- s2 ne répond plus
ALTER SYSTEM SET synchronous_standby_names TO 's3';
SELECT pg_reload_conf();
```

Une alternative est de débrayer le mode synchrone. Cela désactivera aussi le mode synchrone vers d'autres secondaires encore en place.

```
ALTER SYSTEM SET synchronous_commit TO 'local' ;
SELECT pg_reload_conf();
```

Mais les sessions bloquées ne verront pas tout de suite le changement de configuration. Il faudra leur envoyer un signal pour qu'elles se terminent. Elles seront bien validées, dans les journaux du primaire au moins.

```
SELECT pg_cancel_backend(2868749) ;

 pg_cancel_backend
-------------------
 t
```

Apparaît alors le message suivant dans les traces :

```
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been
 ↪  replicated to the standby.
```

1.7 RÉPLICATION EN CASCADE

- Un secondaire peut fournir les informations de réplication
- Décharger le serveur primaire de ce travail
- Diminuer la bande passante du serveur primaire

Imaginons un système PostgreSQL installé à Paris et un serveur secondaire installé à Marseille. Il s'avère que le site de Marseille devient plus important et qu'un deuxième serveur secondaire doit y être installé.

Si ce deuxième serveur secondaire se connecte directement sur le primaire à Paris, la consommation de la bande passante va doubler. PostgreSQL permet au deuxième serveur secondaire de se connecter au premier (donc en local dans notre exemple) pour récupérer les informations de réplication. La bande passante est ainsi mieux maîtrisée.

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

Si un secondaire est promu et devient primaire, cela n'a pas d'impact sur ses propres secondaires.

1.8 DÉCROCHAGE D'UN SECONDAIRE

- Par défaut, le primaire n'attend **pas** les secondaires pour recycler ses WAL
  - risque de « décrochage » !
- 3 solutions :
- archivage en plus du *streaming*
  - bascule automatique
  - mutualisation avec sauvegarde PITR
- Slot de réplication
- Garder des journaux
  - `wal_keep_size` (v13+) / `wal_keep_segments` (<13)

Par défaut, le primaire n'attend **pas** que le serveur secondaire ait obtenu tous les journaux avant de recycler ses journaux.

Le secondaire peut donc se retrouver à demander au principal des informations que celui-ci n'a même plus à disposition car il a recyclé les journaux concernés. Cela peut arriver si la liaison est mauvaise, instable, ou si le secondaire a peiné à réappliquer les journaux pour une raison ou une autre, voire s'il a été déconnecté un certain temps. Le secondaire ne peut alors plus continuer la réplication : il « décroche » (tout comme après la perte d'un journal en *log shipping*).

Ce phénomène peut intervenir même sur un serveur fraîchement copié, si le maître évolue trop vite.

Il faut reconstruire le secondaire, ce qui est peut être très gênant avec une base volumineuse.

Une réplication synchrone ne protège pas de ce problème, car toutes les transactions ne sont pas forcément synchrones. De plus, l'impact en performance est sévère. `hot_standby_feedback` et `vacuum_defer_cleanup_age` (<= v15) ne protègent pas non plus.

Il existe plusieurs moyens pour éviter le décrochage :

L'archivage comme sécurisation du *streaming*

Une réplication par *log shipping* peut être configurée en plus de la réplication par flux. Comme une sauvegarde PITR du principal est très souvent en place, il ne coûte pas grand-chose de permettre au secondaire d'y puiser les journaux manquants.

Ainsi, si la réplication par *streaming* décroche, le secondaire bascule sur la restauration par *log shipping* et va puiser dans le dépôt d'archives, dont l'historique couvre généralement plusieurs jours, voire semaines. Une fois le retard rattrapé, le secondaire ne trouvera plus de nouveaux journaux et rebasculera sur la réplication par *streaming*, qui fonctionnera à nouveau.

Un inconvénient est qu'il faut bien penser à tester les deux modes de réplication pour ne pas avoir de mauvaise surprise le jour où le *streaming* décroche.

Cette configuration est très fréquente, et même recommandée, surtout avec une sauvegarde PITR déjà en place.

Slots de réplication

Un secondaire peut informer son primaire de là où il en est au moyen d'un « slots de réplication ». Le primaire sait ainsi quels journaux sont encore nécessaires à ses secondaires et ne les recycle pas. Cette technique est également très courante. Nous allons la voir plus bas.

Garder des journaux

La dernière méthode est moins recommandée mais peut être utile : elle consiste à paramétrer `wal_keep_size` sur le primaire, par exemple :

```
wal_keep_size = '16GB'
```

Les journaux de transaction bons à recycler seront en fait conservés temporairement à hauteur de la volumétrie indiquée. Un secondaire en retard a alors plus de chances que le primaire n'ait pas déjà effacé les journaux dont il a besoin.

C'est le moyen le plus simple, mais il gaspille du disque de façon permanente. Surtout, il ne garantit pas d'éviter un décrochage si la quantité à conserver a été sous-estimée.

1.8.1 Sécurisation par log shipping

- `archive_command` / `restore_command`
  - script par l'outil PITR
  - ou `cp`, `scp`, `lftp`, `rsync`, script…
- Nettoyage
  - rétention des journaux si outil PITR
  - ou outil dédié :

```
archive_cleanup_command = '/usr/pgsql-14/bin/pg_archivecleanup -d
↪  rep_archives/ %r'
```

La sécurisation par l'archivage consiste donc à permettre au serveur secondaire de rattraper son retard avant de redémarrer sa connexion de réplication.

Manuellement :

Il suffit qu'`archive_command` et `restore_command` soient correctement configurés et indiquent où copier les archives, et comment les récupérer. La mise en place est la même que lors de la mise en place d'une sauvegarde physique[3]. La `restore_command` est ignorée si le secondaire a rebasculé en *streaming*.

Les serveurs secondaires ont cependant la responsabilité de supprimer les journaux devenus inutiles pour ne pas saturer l'espace disque. Afin d'automatiser ce nettoyage, on définit sur le secondaire le paramètre `archive_cleanup_command`.

La commande qui s'y trouve est appelée périodiquement (même si le *streaming* fonctionne), après chaque *restartpoint* (l'équivalent d'un *checkpoint* sur un secondaire), afin de supprimer les archives devenues inutiles pour le secondaire. Généralement, on se contente d'appeler un outil dédié, nommé `pg_archivecleanup`[4] :

```
archive_cleanup_command = '/usr/pgsql-16/bin/pg_archivecleanup  depot_archives/ %r'
```

La situation se complique si un même dépôt d'archives est partagé par plusieurs secondaires…

Avec un outil de sauvegarde PITR :

La situation est plus simple s'il existe déjà une sauvegarde PITR par un outil comme pgBackRest ou barman[5] : `archive_command` comme `restore_command` sont fournies dans leur documentation.

La purge des journaux étant aussi gérée par cet outil on ne configurera bien sûr pas `archive_cleanup_command` !

1.8.2 Slot de réplication : mise en place

- Slot de réplication sur le primaire :
  - `max_replication_slots`
  - NB : non répliqué !
  - création manuelle :

```
SELECT pg_create_physical_replication_slot ('nomsecondaire') ;
```

- Secondaire :
  - dans `postgresql.conf`

```
primary_slot_name = 'nomsecondaire'
```

  - redémarrage de l'instance (<v13) ou rechargement (v13+)

[3] https://dali.bo/i2_html
[4] https://docs.postgresql.fr/current/pgarchivecleanup.html
[5] https://dali.bo/i4_html

Le paramètre `max_replication_slots` doit être supérieur à 0. Par défaut il vaut 10, ce qui suffit souvent. S'il faut le modifier, un redémarrage est nécessaire.

Un slot de réplication se crée sur le primaire par un appel de fonction et en lui attribuant un nom :

```
SELECT pg_create_physical_replication_slot ('nomsecondaire');
```

Traditionnellement le nom est celui du secondaire qui va l'utiliser. Cela facilite la supervision mais n'a rien d'obligatoire.

Sur le secondaire, on ajoute dans `postgresql.conf` la mention du slot à utiliser :

```
primary_slot_name = 'nomsecondaire'
```

Un slot est propre à l'instance et ne fait pas partie des objets répliqués. Lors d'une restauration PITR ou une bascule, il doit fréquemment être recréé manuellement.

En cas de réplication en cascade, un secondaire peut avoir ses propres slots de réplication dédiés à ses propres secondaires.

1.8.3 Slot de réplication : avantages & risques

- Avantages :
  - plus de risque de décrochage des secondaires
  - supervision facile : `pg_replication_slots`
  - utilisable par pg_basebackup
- Risque : accumulation des journaux
  - danger pour le primaire !
  - sécurité : `max_slot_wal_keep_size` (v13+)
- Risque : vacuum bloqué
  - `hot_standby_feedback` ?

Le slot de réplication garantit au secondaire que son primaire ne recyclera pas les journaux dont il aura encore besoin. Le secondaire peut donc prendre un retard conséquent sans risque de décrochage.

Il est facile de voir à quel point se trouve un secondaire avec la vue `pg_replication_slots` (noter les champs `active` et `restart_lsn`), qui complète `pg_stat_replication` :

```
SELECT * FROM pg_replication_slots ;
```

```
-[ RECORD 1 ]+-----------
slot_name           | s3
plugin              | ø
slot_type           | physical
datoid              | ø
database            | ø
temporary           | f
active              | t
active_pid          | 3951267
xmin                | 3486363
catalog_xmin        | ø
restart_lsn         | 14/ACDD9E10
confirmed_flush_lsn | ø
wal_status          | reserved
safe_wal_size       | 5370962416
two_phase           | f
conflicting         | ø
```

Avec pg_basebackup :

pg_basebackup, déjà évoqué plus haut, utilise les slots pour garantir que sa sauvegarde sera complète. Ses options exactes varient suivant les versions. pg_basebackup est capable de créer ce slot (option `--create-slot`) qui sera conservé ensuite.

Risque d'accumulation des journaux :

Par contre, les slots ont un grave inconvénient : en cas de problème prolongé sur un secondaire, les journaux vont s'accumuler sur le primaire, sans limitation de taille ni de durée, déclenchant une saturation de la partition de `pg_wal` dans le pire des cas — et l'arrêt du primaire.

Certes, la supervision de l'espace disque fait partie des bases de la supervision, mais les journaux s'accumulent parfois très vite lors d'une mise à jour massive.

Il est donc important de détruire tout slot dont le secondaire a été désactivé ou est hors ligne pour un certain temps (quitte à devoir reconstruire le secondaire) :

```
SELECT pg_drop_replication_slot ('nomsecondaire');
```

Les plus prudents se limiteront donc à une réplication par *streaming* classique sans slot, couplée au *log shipping* pour éviter le décrochage. Rappelons que l'archivage peut lui aussi laisser les journaux s'accumuler en cas de problème sur le serveur cible de l'archivage.

À partir de PostgreSQL 13, le paramètre `max_slot_wal_keep_size` permet de limiter la quantité de WAL conservés par les slots de réplication. Au-delà, le primaire ne garantit plus la conservation. Le secondaire risque à nouveau de décrocher, mais une longue indisponibilité ne risque plus de saturer le disque du primaire.

En production, il est conseillé de toujours définir `max_slot_wal_keep_size` (à une valeur élevée au besoin) si l'on crée un slot de réplication. En effet, l'expérience montre que les slots de réplication sont souvent oubliés.

Risque sur le vacuum sur le primaire :

Le slot permet au primaire de mémoriser durablement la transaction où s'est arrêté le secondaire (`pg_replication_slots.xmin` est renseigné), à condition que `hot_standby_feedback` soit à `on`.

Avec un slot de réplication actif et `hot_standby_feedback` à `on`, si le secondaire est durablement déconnecté, non seulement les journaux de transaction vont s'accumuler sur le primaire, mais le vacuum y sera inefficace jusqu'au rétablissement de la réplication de ce secondaire ou la destruction du slot !
Ce problème de vacuum persiste même si l'on a paramétré `max_slot_wal_keep_size` pour éviter la saturation des journaux…

Selon l'utilisation, on peut donc préférer monter `max_standby_streaming_delay` plutôt que de laisser `hot_standby_feedback` à `on`.

1.9 SYNTHÈSE DES PARAMÈTRES

1.9.1 Serveur primaire

Log shipping	Streaming
`wal_level = replica` *	`wal_level = replica` *
`archive_mode = on` *	
`archive_command` *	
`archive_library`	
`archive_timeout`	`wal_sender_timeout`
	`max_wal_senders`
	`max_replication_slots`
	`wal_keep_size`
	`max_slot_wal_keep_size` *

(*) paramètres indispensables, généralement modifiés par rapport à l'installation par défaut, ou d'utilisation fortement conseillés

Ne figurent pas les paramètres disparus dans les toutes dernières versions, généralement inutilisés auparavant.

1.9.2 Serveur secondaire

Log shipping	Streaming
`wal_level = replica` *	`wal_level = replica` *
`restore_command` *	
`archive_cleanup_command`	
(selon outil)	`primary_conninfo` *
	`wal_receiver_timeout`
	`hot_standby`
	`primary_slot_name` *
`max_standby_archive_delay`	`max_standby_streaming_delay`
	`hot_standby_feedback`
	`wal_receiver_status_interval`

(*) paramètres indispensables, généralement modifiés par rapport à l'installation par défaut, ou d'utilisation fortement conseillés

Ne figurent pas les paramètres disparus dans les toutes dernières versions, généralement inutilisés auparavant.

1.10 CONCLUSION

- Système de réplication fiable…
  - et très complet

PostgreSQL possède de nombreuses fonctionnalités de réplication très avancées, telle que le choix du synchronisme de la réplication à la transaction près, ce qui en fait un système aujourd'hui très complet.

1.10.1 Questions

N'hésitez pas, c'est le moment !

1.11 QUIZ

https://dali.bo/w2b_quiz

1.12 TRAVAUX PRATIQUES

1.12.1 Réplication asynchrone en flux avec deux secondaires

But : Mettre en place une réplication asynchrone en flux avec deux secondaires.

Créer un deuxième serveur secondaire **instance3** avec l'outil `pg_basebackup`, en *streaming replication*.

S'assurer que la réplication fonctionne bien et que les processus adéquats sont bien présents.

1.12.2 Slots de réplication

But : Mettre en place un slot de réplication.

Les slots de réplication permettent au serveur principal de connaître les journaux encore nécessaires aux serveurs secondaires.

- Créer un slot de réplication sur le primaire.
- Configurer le deuxième secondaire pour utiliser ce slot.
- Contrôler que le slot est bien actif.

- Arrêter le deuxième secondaire et générer de l'activité (créer une table d'un million de lignes, par exemple).
- Où sont conservés les journaux de transaction ?
- Forcer un `CHECKPOINT`. Quel est le journal le plus ancien sur le serveur principal ?

- Que se serait-il passé sans slot de réplication ?

- Démarrer le deuxième secondaire.
- Contrôler que les deux secondaires sont bien en réplication par *streaming* avec la vue système adéquate.

- Détruire le slot de réplication.

1.12.3 Log shipping

But : Mettre en place une réplication par *log shipping*.

- Plutôt que d'utiliser un slot de réplication, configurer l'instance primaire pour qu'elle archive ses journaux de transactions dans `/var/lib/pgsql/14/archives`.
- Vérifier que l'archivage fonctionne.

- Configurer les instances secondaires pour utiliser ces archives en cas de désynchronisation. Ne pas oublier de redémarrer.

- Simuler un décrochage de la deuxième instance secondaire.
- Vérifier les WALs au niveau du serveur principal ainsi que les archives.

- Re-démarrer l'instance secondaire et observer ses traces pour suivre ce qu'il se passe.

- 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 ?

- Si les deux secondaires puisaient leur journaux depuis le même répertoire d'archive, quel pourrait être le risque ?

1.12.4 Réplication synchrone en flux avec trois secondaires

But : Mettre en place une réplication synchrone en flux avec trois secondaires.

- Créer un troisième serveur secondaire **instance4** avec l'outil `pg_basebackup`, en *streaming replication*.

- Passer la réplication en synchrone pour un seul secondaire.

- Arrêter le secondaire synchrone.
- Modifier des données sur le principal. Que se passe-t-il ?

- Redémarrer le secondaire synchrone.

- Passer le deuxième secondaire comme également synchrone, en indiquant sur le primaire :

```
synchronous_standby_names = 'FIRST 1 (instance2,instance3)'
```

- Arrêter le premier secondaire synchrone.
- Modifier des données sur le principal. Que se passe-t-il ?

- Quel paramètre modifier pour avoir deux secondaires synchrones simultanément ?
- Vérifier que les deux secondaires sont synchrones.

- Pour la suite du TP, configurer le paramètre `application_name` de l'instance **instance4**.
- Ensuite, se baser sur la notion de quorum pour obtenir deux serveurs synchrones parmi les trois instances secondaires en utilisant :

```
synchronous_standby_names = ANY 2 (instance2,instance3,instance4)
```

1.12.5 Réplication synchrone : cohérence des lectures (optionnel)

But : Découvrir les mécanismes de mise en pause et de reprise de réplication.

- 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 (utiliser la vue `pg_stat_replication`) ?
- Relancer le rejeu et contrôler la présence des enregistrements sur les trois instances.

- Quel paramètre modifier pour obtenir les mêmes résultats sur les trois instances ?

- Appliquer ce paramètre et effectuer la même opération (pause du rejeu puis insertion d'enregistrements sur le principal). Que se passe-t-il ?

1.13 TRAVAUX PRATIQUES (SOLUTIONS)

1.13.1 Réplication asynchrone en flux avec deux secondaires

> Créer un deuxième serveur secondaire **instance3** avec l'outil `pg_basebackup`, en *streaming replication*.

À part les appels à `systemctl`, les opérations se font toutes en tant que **postgres**.

Nous allons utiliser la même méthode que précédemment pour créer le deuxième secondaire :

```
$ pg_basebackup -D /var/lib/pgsql/14/instance3 -P -h 127.0.0.1 -U repli -R -c fast
104425/104425 kB (100%), 1/1 tablespace
```

Créer le fichier `standby.signal` s'il n'existe pas déjà et adapter le port de l'instance primaire dans `postgresql.auto.conf` créé par la commande précédente :

```
$ touch /var/lib/pgsql/14/instance3/standby.signal
$ cat /var/lib/pgsql/14/instance3/postgresql.auto.conf
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass'' host=127.0.0.1
 ↪  port=5432 sslmode=prefer sslcompression=0 gssencmode=prefer krbsrvname=postgres
 ↪  target_session_attrs=any'
```

Il est nécessaire de modifier le numéro de port de l'instance dans `/var/lib/pgsql/14/instance3/postgresql.conf`

```
port = 5434
```

Et on peut enfin démarrer le deuxième secondaire :

```
# systemctl start instance3
```

> S'assurer que la réplication fonctionne bien et que les processus adéquats sont bien présents.

Vérifions la liste des processus :

```
$ ps -o pid,cmd fx

  PID CMD
 6845 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance3/
 6847  \_ postgres: logger
 6848  \_ postgres: startup   recovering 00000003000000000000000E
 6849  \_ postgres: checkpointer
 6850  \_ postgres: background writer
 6851  \_ postgres: stats collector
 6852  \_ postgres: walreceiver   streaming 0/E000060

 5841 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
 5843  \_ postgres: logger
 5844  \_ postgres: startup   recovering 00000003000000000000000E
 5845  \_ postgres: checkpointer
 5846  \_ postgres: background writer
 5847  \_ postgres: stats collector
```

```
 5848  \_ postgres: walreceiver   streaming 0/E000060

 5684 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
 5686  \_ postgres: logger
 5688  \_ postgres: checkpointer
 5689  \_ postgres: background writer
 5690  \_ postgres: stats collector
 5792  \_ postgres: walwriter
 5793  \_ postgres: autovacuum launcher
 5794  \_ postgres: logical replication launcher
 5849  \_ postgres: walsender repli 127.0.0.1(48230) streaming 0/E000060
 6853  \_ postgres: walsender repli 127.0.0.1(48410) streaming 0/E000060
```

L'instance principale (`/var/lib/pgsql/14/instance1/`) a bien deux processus `walsender` et chaque instance secondaire, son `walreceiver`.

1.13.2 Slots de réplication

- Créer un slot de réplication sur le primaire.
- 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 :

```
postgres=# SHOW max_replication_slots;

 max_replication_slots
-----------------------
 10
```

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

```
postgres=# SELECT pg_create_physical_replication_slot('slot_instance3');

 pg_create_physical_replication_slot
-------------------------------------
 (slot_instance3,)
```

Il faut ensuite spécifier le slot dans le fichier `postgresql.conf` (ou ici `postgresql.auto.conf`, utilisé par `pg_basebackup` pour créer le paramétrage initial de la réplication) :

```
primary_slot_name = 'slot_instance3'
```

Puis redémarrer le serveur secondaire.

```
# systemctl restart instance3
```

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_instance3
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 6900
xmin                |
catalog_xmin        |
restart_lsn         | 0/E000148
confirmed_flush_lsn |
```

- Arrêter le deuxième secondaire et générer de l'activité (créer une table d'un million de lignes, par exemple).
- Où sont conservés les journaux de transaction ?
- Forcer un `CHECKPOINT`. Quel est le journal le plus ancien sur le serveur principal ?

```
# systemctl stop instance3
```

Pour générer de l'activité :

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

Cette table fait 35 Mo, et va donc nécessiter l'écriture ou la création d'au moins 3 fichiers WAL de 16 Mo.

On les voit en regardant les journaux au niveau du serveur principal :

```
$ ls -alh /var/lib/pgsql/14/instance1/pg_wal/
```

```
(...)
-rw-------.  1 postgres postgres  16M Nov 27 16:40 00000003000000000000000E
-rw-------.  1 postgres postgres  16M Nov 27 16:40 00000003000000000000000F
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000030000000000000010
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000030000000000000011
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000020000000000000012
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000020000000000000013
```

(Au niveau SQL, la fonction `SELECT * FROM pg_ls_waldir() ORDER BY 1 ;` fonctionne aussi.)

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

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

```
CHECKPOINT
```

Le journal le plus ancien (ici `00000003000000000000000E`) est toujours présent.

- Que se serait-il passé sans slot de réplication ?

Le deuxième secondaire n'aurait pas pu récupérer des journaux indispensables à la réplication et les aurait attendu indéfiniment. Le serveur primaire aurait recyclé ses journaux inutiles après le *checkpoint* suivant (ils auraient été renommés).

- Démarrer le deuxième secondaire.
- Contrôler que les deux secondaires sont bien en réplication par *streaming* avec la vue système adéquate.

```
# systemctl start instance3
```

```
$  psql -x -c "SELECT * FROM pg_stat_replication"
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 5849
usesysid         | 16384
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 48230
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/13FFE6D8
write_lsn        | 0/13FFE6D8
flush_lsn        | 0/13FFE6D8
replay_lsn       | 0/13FFE6D8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
-[ RECORD 2 ]----+------------------------------
pid              | 7044
usesysid         | 16384
usename          | repli
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 48434
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/13FFE6D8
write_lsn        | 0/13FFE6D8
flush_lsn        | 0/13FFE6D8
replay_lsn       | 0/13FFE6D8
write_lag        | 00:00:00.942356
flush_lag        | 00:00:00.964213
replay_lag       | 00:00:01.378381
sync_priority    | 0
sync_state       | async
reply_time       | ...
```

La synchronisation a pu se faire rapidement.

Forcer un `CHECKPOINT` et revérifier le répertoire des WALs :

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

```
CHECKPOINT
```

```
$ ls -alh /var/lib/pgsql/14/instance1/pg_wal/
```

```
(...)
-rw-------.  1 postgres postgres  16M Nov 27 16:44 000000030000000000000013
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000030000000000000014
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000030000000000000015
-rw-------.  1 postgres postgres  16M Nov 27 16:40 000000030000000000000016
-rw-------.  1 postgres postgres  16M Nov 27 16:41 000000030000000000000017
-rw-------.  1 postgres postgres  16M Nov 27 16:41 000000030000000000000018
-rw-------.  1 postgres postgres  16M Nov 27 16:41 000000030000000000000019
```

Le deuxième serveur secondaire ayant rattrapé son retard, le primaire peut enfin recycler ses anciens journaux : ceux affichés sont prêts à recevoir des données.

- Détruire le slot de réplication.

Les slots de réplication ont un grave inconvénient : en cas de problème prolongé sur un secondaire, les journaux vont s'accumuler sur le primaire, sans limitation de taille ni de durée, déclenchant une saturation de la partition de `pg_wal` dans le pire des cas — et l'arrêt du primaire.

Il est donc important de détruire tout slot dont le secondaire a été désactivé ou est hors ligne pour un certain temps (quitte à devoir reconstruire le secondaire).

Ici, après avoir supprimé `primary_slot_name` du fichier de configuration et redémarré l'instance **instance3** :

```
$ psql -c "SELECT pg_drop_replication_slot('slot_instance3');"
```

1.13.3 Log shipping

- Plutôt que d'utiliser un slot de réplication, configurer l'instance primaire pour qu'elle archive ses journaux de transactions dans `/var/lib/pgsql/14/archives`.
- Vérifier que l'archivage fonctionne.

Tout d'abord, il faut créer le répertoire d'archivage :

```
$ mkdir /var/lib/pgsql/14/archives
```

L'utilisateur **postgres** doit avoir le droit de lire et écrire dans ce répertoire.

Modifions maintenant le fichier `/var/lib/pgsql/14/instance1/postgresql.conf` pour que PostgreSQL archive les journaux de transactions :

```
archive_mode = on
archive_command = 'rsync %p /var/lib/pgsql/14/archives/%f'
```

La commande `rsync` n'est pas installée par défaut. Le paquet se nomme simplement **rsync** dans toutes les distributions Linux habituelles.

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

```
# systemctl restart instance1
```

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

```
$ psql -c "SELECT pg_switch_wal()"

 pg_switch_wal
---------------
 0/13FFE850

$ ls -l /var/lib/pgsql/14/archives/

total 16384
-rw-------. 1 postgres postgres 16777216 Nov 27 16:59 000000030000000000000013
```

Au fil de l'activité, les journaux vont s'accumuler à cet endroit.

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

Pour superviser le bon déroulement de l'archivage, on peut suivre aussi le vue `pg_stat_archiver`.

- Configurer les instances secondaires pour utiliser ces archives en cas de désynchronisation. Ne pas oublier de redémarrer.

Maintenant que l'archivage fonctionne, configurons nos instances secondaires pour utiliser ces archives en cas de désynchronisation en ajoutant dans leurs `postgresql.conf` (ou ici, `postgresql.auto.conf`) :

```
restore_command = 'cp /var/lib/pgsql/14/archives/%f %p'
```

Penser à redémarrer ensuite les deux instances :

```
# systemctl restart instance2
# systemctl restart instance3
```

- Simuler un décrochage de la deuxième instance secondaire.
- Vérifier les WALs au niveau du serveur principal ainsi que les archives.

Pour simuler un décrochage de la deuxième instance secondaire, arrêter le service, générer de l'activité et forcer un `CHECKPOINT` :

```
# systemctl stop instance3

$ psql -c "INSERT INTO t1 SELECT * FROM generate_series(1,2000000);" b1
$ psql -c "CHECKPOINT;"
```

Vérifier les WALs au niveau du serveur principal ainsi que les archives :

```
$ ls -alh /var/lib/pgsql/14/instance1/pg_wal/

(...)
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000001B
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000001C
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000001D
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000001E
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000001F
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000020
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000021
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000022
-rw-------.  1 postgres postgres  16M Nov 27 17:21 00000003000000000000023

$ ls -alh /var/lib/pgsql/14/archives/

(...)
-rw-------. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000001B
-rw-------. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000001C
-rw-------. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000001D
-rw-------. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000001E
```

> – Re-démarrer l'instance secondaire et observer ses traces pour suivre ce qu'il se passe.

```
# systemctl start instance3

$ tail -f /var/lib/pgsql/14/instance3/log/postgresql-*.log

(...)
LOG:  restored log file "00000003000000000000001B" from archive
LOG:  restored log file "00000003000000000000001C" from archive
LOG:  restored log file "00000003000000000000001D" from archive
LOG:  restored log file "00000003000000000000001E" from archive
cp: cannot stat ‘/var/lib/pgsql/14/archives/00000003000000000000001F’:
No such file or directory
LOG:  started streaming WAL from primary at 0/1F000000 on timeline 3
```

Une fois le retard de réplication rattrapé grâce aux archives, l'instance secondaire se reconnecte automatiquement à l'instance primaire.

> – 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` 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. Il faut évidemment penser à redémarrer les instances après changement de paramétrage.

```
archive_cleanup_command =
     '/usr/pgsql-14/bin/pg_archivecleanup -d /var/lib/pgsql/14/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(1,1000000);" b1
$ psql -c "CHECKPOINT;"
$ psql -p 5433 -c "CHECKPOINT;"
$ psql -p 5434 -c "CHECKPOINT;"
```

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

```
(...)
pg_archivecleanup: keeping WAL file
                   "/var/lib/pgsql/14/archives//000000030000000000000029" and later
pg_archivecleanup: removing file
                   "/var/lib/pgsql/14/archives//000000030000000000000013"
pg_archivecleanup: removing file
                   "/var/lib/pgsql/14/archives//000000030000000000000014"
(...)
```

> – Si les deux secondaires puisaient 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.

Pour éviter cela, chaque secondaire doit posséder ses propres copies des journaux, ou la purge doit être opérée par un outil tiers (sauvegarde PITR généralement).

1.13.4 Réplication synchrone en flux avec trois secondaires

> – Créer un troisième serveur secondaire **instance4** avec l'outil `pg_basebackup`, en *streaming replication*.

Nous allons utiliser la même méthode que précédemment pour créer le deuxième secondaire :

```
$ pg_basebackup -D /var/lib/pgsql/14/instance4 -P -h 127.0.0.1 -U repli -R -c fast

104425/104425 kB (100%), 1/1 tablespace
```

Créer le fichier `standby.signal` s'il n'existe pas déjà et adapter le port de l'instance primaire dans `postgresql.auto.conf` créé par la commande précédente :

```
$ touch /var/lib/pgsql/14/instance4/standby.signal
$ cat /var/lib/pgsql/14/instance4/postgresql.auto.conf
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass'' host=127.0.0.1
 ↪  port=5432 sslmode=prefer sslcompression=0 gssencmode=prefer krbsrvname=postgres
 ↪  target_session_attrs=any'
```

Il est nécessaire de modifier le numéro de port de l'instance dans `/var/lib/pgsql/14/instance4/postgresql.conf`

```
port = 5435
```

Et on peut enfin démarrer le deuxième secondaire :

```
# systemctl start instance4
```

> – Passer la réplication en synchrone pour un seul secondaire.

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` (configuration dans `postgresql.conf` ou `postgresql.auto.conf`) :

```
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass''
                    host=127.0.0.1 port=5432
                    application_name=instance2'
```

Ensuite, nous devons indiquer le serveur `secondaire` dans la liste des serveurs synchrones initialisée par le paramètre `synchronous_standby_names`. Il faut modifier cette valeur dans le fichier `/var/lib/pgsql/14/instance1/postgresql.conf` :

```
synchronous_standby_names = 'instance2'
```

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

```
# systemctl reload instance1
# systemctl restart instance2
```

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 trois secondaires :

```
$ psql -p 5432

psql (14.1)
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    | ...
state            | streaming
sync_state       | async
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sync_state       | sync
-[ RECORD 3 ]----+------------------------------
application_name | walreceiver
```

```
backend_start     | ...
state             | streaming
sync_state        | async
```

Nous avons bien un serveur synchrone et deux serveurs asynchrones.

- Arrêter le secondaire synchrone.
- Modifier des données sur le principal. Que se passe-t-il ?

Exécutons une requête de modification :

```
postgres=# \c b1
You are now connected to database "b1" as user "postgres".

b1=# CREATE TABLE t3(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 :

```
# systemctl stop instance2

$ psql -p 5432 b1

psql (14.1)
Type "help" for help.

b1=# CREATE TABLE t4(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`.

- Redémarrer le secondaire synchrone.

Démarrer le secondaire synchrone à partir d'un autre terminal : la requête sur le principal se termine.

- Passer le deuxième secondaire comme également synchrone, en indiquant sur le primaire :

```
synchronous_standby_names = 'FIRST 1 (instance2,instance3)'
```

Nous allons maintenant passer le deuxième secondaire en synchrone avec le `application_name` positionné à **instance3** afin de les différencier (il est possible d'utiliser le même `application_name`). Ensuite ajoutons **instance3** à la liste des `synchronous_standby_names` sur l'instance principale.

```
synchronous_standby_names = 'FIRST 1 (instance2,instance3)'

# systemctl restart instance3
# systemctl reload instance1
```

```
$ psql -x -p 5432 -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sync_state       | potential
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sync_state       | sync
-[ RECORD 3 ]----+------------------------------
application_name | walreceiver
backend_start    | ...
state            | streaming
sync_state       | async
```

```
$ psql -p 5432 -c "SHOW synchronous_standby_names"

     synchronous_standby_names
---------------------------------
 FIRST 1 (instance2,instance3)
```

Cette fois les deux serveurs correspondent au `synchronous_standby_names`, on peut constater qu'un serveur est `sync` et l'autre `potential`. On a demandé 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`.

- Arrêter le premier secondaire synchrone.
- Modifier des données sur le principal. Que se passe-t-il ?

Arrêt du premier secondaire synchrone :

```
# systemctl stop instance2
```

```
$ psql -x -p 5432 -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sync_state       | sync
-[ RECORD 2 ]----+------------------------------
application_name | walreceiver
backend_start    | ...
state            | streaming
sync_state       | async
```

Et faisons une modification sur le principal :

```
$ psql -p 5432 -c "CREATE TABLE t5(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.

```
# systemctl start instance2
```

- Quel paramètre modifier pour avoir deux secondaires synchrones simultanément ?
- Vérifier que les deux secondaires sont synchrones.

Dans notre cas :

```
synchronous_standby_names = 'FIRST 2 (instance2,instance3)'
```

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

```
$ psql -x -c "SELECT application_name, backend_start, state, sync_state FROM
 ↪  pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sync_state       | sync
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sync_state       | sync
-[ RECORD 3 ]----+------------------------------
application_name | walreceiver
backend_start    | ...
state            | streaming
sync_state       | async
```

L'indisponibilité d'un seul des deux secondaires générera une attente lors d'écritures sur le primaire.

- Pour la suite du TP, configurer le paramètre `application_name` de l'instance **instance4**.
- Ensuite, se baser sur la notion de quorum pour obtenir deux serveurs synchrones parmi les trois instances secondaires en utilisant :

```
synchronous_standby_names = ANY 2 (instance2,instance3,instance4)
```

Configurer le `application_name` de l'instance **instance4**.

Ensuite, configurer dans `/var/lib/pgsql/14/instance1/postgresql.conf` :

```
synchronous_standby_names = 'ANY 2 (instance2,instance3,instance4)'
```

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

```
$ psql -x -c "SELECT application_name, backend_start, state, sync_state
FROM pg_stat_replication;"
```

```
-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sync_state       | quorum
-[ RECORD 3 ]----+------------------------------
application_name | instance4
backend_start    | ...
state            | streaming
sync_state       | quorum
```

1.13.5 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

$ psql -p 5433 -c "SELECT * FROM t4;" b1

 id
----
(0 rows)

$ psql -p 5434 -c "SELECT * FROM t4;" b1

 id
----
  1

$ psql -p 5435 -c "SELECT * FROM t4;" b1

 id
----
  1
```

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

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

```
$ psql -x -p 5432 -c "SELECT application_name, backend_start, state, sent_lsn,
write_lsn, flush_lsn, replay_lsn, sync_state FROM pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A0C8
write_lsn        | 0/2D02A0C8
flush_lsn        | 0/2D02A0C8
replay_lsn       | 0/2D02A0C8
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A0C8
write_lsn        | 0/2D02A0C8
flush_lsn        | 0/2D02A0C8
replay_lsn       | 0/2D028030
sync_state       | quorum
-[ RECORD 3 ]----+------------------------------
application_name | instance4
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A0C8
write_lsn        | 0/2D02A0C8
flush_lsn        | 0/2D02A0C8
replay_lsn       | 0/2D02A0C8
sync_state       | quorum
```

Les serveurs secondaires sont bien en réplication synchrone avec la notion de quorum. On constate que tous ont bien reçu les enregistrements mais **instance2** n'a pas 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 -x -p 5432 -c "SELECT application_name, backend_start, state, sent_lsn,
write_lsn, flush_lsn, replay_lsn, sync_state FROM pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
application_name | instance3
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A1B0
write_lsn        | 0/2D02A1B0
flush_lsn        | 0/2D02A1B0
```

```
replay_lsn       | 0/2D02A1B0
sync_state       | quorum
-[ RECORD 2 ]----+------------------------------
application_name | instance2
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A1B0
write_lsn        | 0/2D02A1B0
flush_lsn        | 0/2D02A1B0
replay_lsn       | 0/2D02A1B0
sync_state       | quorum
-[ RECORD 3 ]----+------------------------------
application_name | instance4
backend_start    | ...
state            | streaming
sent_lsn         | 0/2D02A1B0
write_lsn        | 0/2D02A1B0
flush_lsn        | 0/2D02A1B0
replay_lsn       | 0/2D02A1B0
sync_state       | quorum
```

Cette fois, **instance2** a bien rejoué les journaux. Les enregistrements sont bien présents dans la table `t4` :

```
$ psql -p 5433 -c "SELECT * FROM t4;" b1

 id
----
  1
```

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

Par défaut la réplication synchrone garantit 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'enregistrements sur le principal). Que se passe-t-il ?

Dans `/var/lib/pgsql/14/instance1/postgresql.conf` :

```
synchronous_commit = remote_apply
```

Faire un rechargement de la configuration du serveur principal :

```
# systemctl reload instance1
```

```
$ 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 5435 -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 deux secondaires pour qu'elle puisse s'effectuer puisque nous avons configuré.

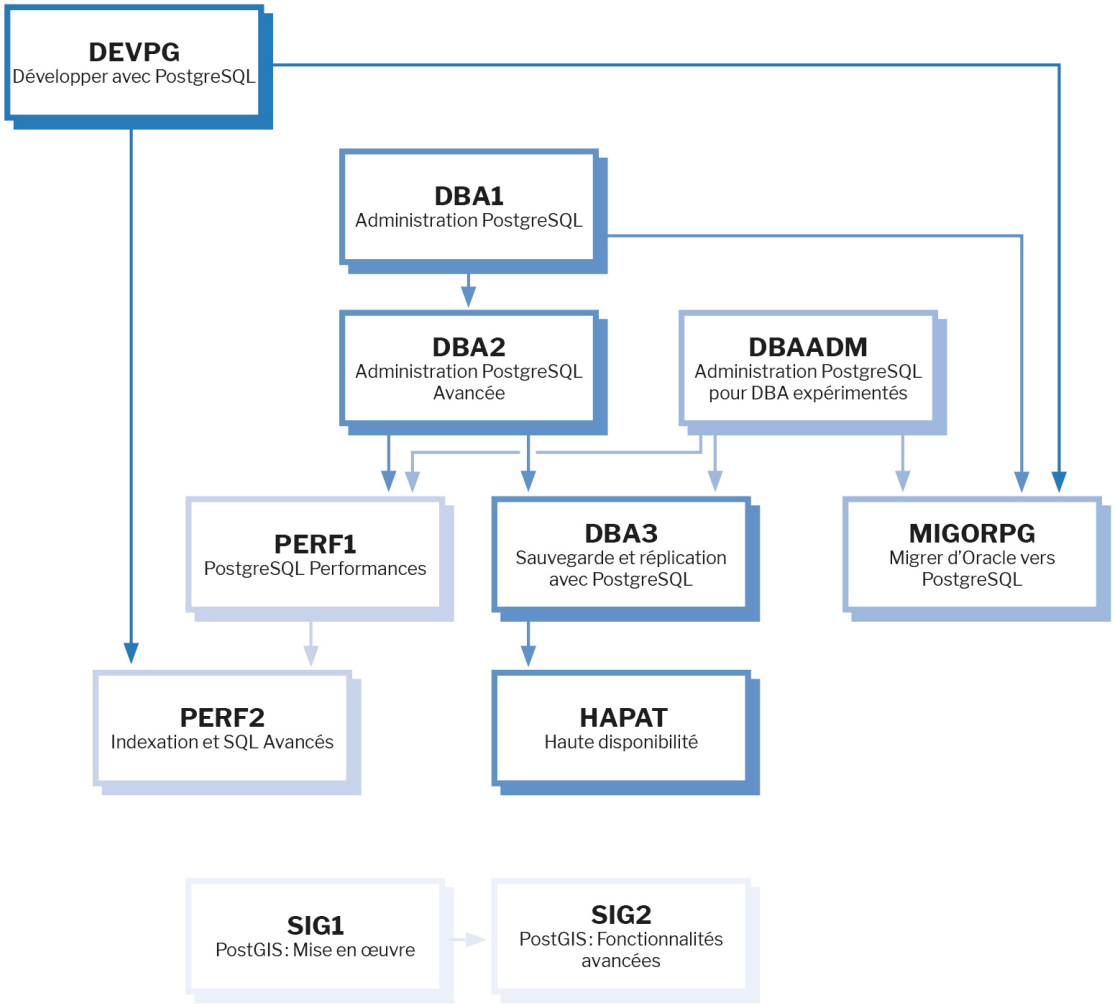
```
synchronous_standby_names = 'ANY 2 (instance2,instance3,instance4)'
```

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.