

Formation DBA3

PostgreSQL Sauvegardes et RéPLICATION



23.09

Table des matières

Chers lectrices & lecteurs,	1
À propos de DALIBO	1
Remerciements	1
Licence Creative Commons CC-BY-NC-SA	2
Marques déposées	2
Sur ce document	2
1/ PostgreSQL : Politique de sauvegarde	5
1.1 Introduction	6
1.1.1 Au menu	6
1.2 Définir une politique de sauvegarde	7
1.2.1 Objectifs	8
1.2.2 Différentes approches	8
1.2.3 RTO/RPO	9
1.2.4 Industrialisation	10
1.2.5 Documentation	11
1.2.6 Règle 3-2-1	12
1.2.7 Autres points d'attention	13
1.3 Conclusion	15
1.4 Quiz	16
2/ Sauvegarde physique à chaud et PITR	17
2.1 Introduction	18
2.1.1 Au menu	18
2.2 PITR	20
2.2.1 Principes	20
2.2.2 Avantages	21
2.2.3 Inconvénients	22
2.3 Copie physique à chaud ponctuelle avec pg_basebackup	23
2.4 Sauvegarde PITR	26
2.4.1 Méthodes d'archivage	26
2.4.2 Choix du répertoire d'archivage	27
2.4.3 Processus archiver : configuration	27
2.4.4 Processus archiver : lancement	30
2.4.5 Processus archiver : supervision	31
2.4.6 pg_receivewal	33
2.4.7 pg_receivewal - configuration serveur	34
2.4.8 pg_receivewal - redémarrage du serveur	35
2.4.9 pg_receivewal - lancement de l'outil	36
2.4.10 Avantages et inconvénients	37
2.5 Sauvegarde PITR manuelle	38
2.5.1 Sauvegarde manuelle - 1/3 : pg_backup_start	39

2.5.2	Sauvegarde manuelle - 2/3 : copie des fichiers	40
2.5.3	Sauvegarde manuelle - 3/3 : pg_backup_stop	42
2.5.4	Sauvegarde de base à chaud : pg_basebackup	43
2.5.5	Fréquence de la sauvegarde de base	44
2.5.6	Suivi de la sauvegarde de base	44
2.6	Restaurer une sauvegarde PITR	45
2.6.1	Restaurer une sauvegarde PITR (1/5)	45
2.6.2	Restaurer une sauvegarde PITR (2/5)	45
2.6.3	Restaurer une sauvegarde PITR (3/5)	46
2.6.4	Restaurer une sauvegarde PITR (4/5)	47
2.6.5	Restaurer une sauvegarde PITR (5/5)	49
2.6.6	Restauration PITR : durée	50
2.6.7	Restauration PITR : différentes timelines	51
2.6.8	Restauration PITR : illustration des timelines	53
2.6.9	Après la restauration	55
2.7	Pour aller plus loin	56
2.7.1	Compresser les journaux de transactions	56
2.7.2	Outils de sauvegarde PITR dédiés	57
2.7.3	pgBackRest	57
2.7.4	barman	58
2.7.5	pittry	59
2.8	Conclusion	60
2.8.1	Questions	60
2.9	Quiz	61
2.10	Installation de PostgreSQL depuis les paquets communautaires	62
2.10.1	Sur Rocky Linux 8	62
2.10.2	Sur Red Hat 7 / Cent OS 7	64
2.10.3	Sur Debian / Ubuntu	64
2.10.4	Accès à l'instance sur le serveur même	66
2.11	Introduction à pgbench	69
2.11.1	Installation	69
2.11.2	Générer de l'activité	69
2.12	Travaux pratiques	71
2.12.1	pg_basebackup : sauvegarde ponctuelle & restauration	71
2.12.2	pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants	72
2.13	Travaux pratiques (solutions)	74
2.13.1	pg_basebackup : sauvegarde ponctuelle & restauration	74
2.13.2	pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants	78
3/ PostgreSQL : Outils de sauvegarde physique		83
3.1	Introduction	84
3.1.1	Au menu	84
3.1.2	Définition du besoin - Contexte	85
3.2	pg_basebackup - Présentation	86
3.2.1	pg_basebackup - Formats de sauvegarde	86

3.2.2	pg_basebackup - Avantages	87
3.2.3	pg_basebackup - Limitations	91
3.3	pgBackRest - Présentation générale	93
3.3.1	pgBackRest - Fonctionnalités	93
3.3.2	pgBackRest - Sauvegardes	94
3.3.3	pgBackRest - Restauration	94
3.3.4	pgBackRest - Installation	95
3.3.5	pgBackRest - Utilisation	96
3.3.6	pgBackRest - Configuration	97
3.3.7	pgBackRest - Configuration PostgreSQL	97
3.3.8	pgBackRest - Configuration globale	98
3.3.9	pgBackRest - Configuration de la rétention	98
3.3.10	pgBackRest - Configuration SSH	99
3.3.11	pgBackRest - Configuration TLS	100
3.3.12	pgBackRest - Configuration par instance	102
3.3.13	pgBackRest - Exemple configuration par instance	102
3.3.14	pgBackRest - Initialiser le répertoire de stockage des sauvegardes	103
3.3.15	pgBackRest - Effectuer une sauvegarde	103
3.3.16	pgBackRest - Lister les sauvegardes	105
3.3.17	pgBackRest - Planification	106
3.3.18	pgBackRest - Dépôts	106
3.3.19	pgBackRest - bundling et sauvegarde incrémentale en mode block	108
3.3.20	pgBackRest - Restauration	110
3.4	Barman - Présentation générale	111
3.4.1	Barman - Scénario « streaming-only »	111
3.4.2	Barman - Scénario « rsync-over-ssh »	112
3.4.3	Barman - Sauvegardes	113
3.4.4	Barman - Sauvegardes (suite)	113
3.4.5	Barman - Politique de rétention	114
3.4.6	Barman - Restauration	114
3.4.7	Barman - Installation	115
3.4.8	Barman - Utilisation	116
3.4.9	Barman - Configuration	117
3.4.10	Barman - Configuration utilisateur	117
3.4.11	Barman - Configuration SSH	118
3.4.12	Barman - Configuration PostgreSQL	118
3.4.13	Barman - Configuration globale	119
3.4.14	Barman - Configuration sauvegardes	119
3.4.15	Barman - Configuration réseau	120
3.4.16	Barman - Configuration rétention	120
3.4.17	Barman - Configuration des hooks	121
3.4.18	Barman - Configuration d'un dépôt synchronisé	121
3.4.19	Barman - Configuration par instance	122
3.4.20	Barman - Exemple configuration par instance	122
3.4.21	Barman - Exemple configuration Streaming Only	123

3.4.22	Barman - Vérification de la configuration	124
3.4.23	Barman - Statut	126
3.4.24	Barman - Diagnostiquer	128
3.4.25	Barman - Nouvelle sauvegarde	128
3.4.26	Barman - Lister les sauvegardes	129
3.4.27	Barman - Détail d'une sauvegarde	129
3.4.28	Barman - Suppression d'une sauvegarde	130
3.4.29	Barman - Conserver une sauvegarde	131
3.4.30	Barman - Tâches de maintenance	131
3.4.31	Barman - Restauration	132
3.4.32	Barman - Options de restauration	132
3.4.33	Barman - Exemple de restauration à distance	133
3.5	pitrery - Présentation générale	135
3.6	Autres outils de l'écosystème	136
3.6.1	WAL-G - présentation	136
3.7	Conclusion	137
3.8	Quiz	138
3.9	Travaux pratiques	139
3.9.1	Utilisation de pgBackRest (Optionnel)	139
3.9.2	Utilisation de barman (Optionnel)	139
3.9.3	Utilisation de barman (Optionnel)	140
3.10	Travaux pratiques (solutions)	141
3.10.1	Utilisation de pgBackRest (Optionnel)	141
3.10.2	Utilisation de barman (Optionnel)	143
4/	Solutions de réPLICATION	151
4.1	Préambule	152
4.1.1	Au menu	152
4.1.2	Objectifs	153
4.2	Rappels théoriques	154
4.2.1	Cluster, primaire, secondaire, <i>standby</i>	154
4.2.2	RéPLICATION asynchrone asymétrique	155
4.2.3	RéPLICATION asynchrone symétrique	156
4.2.4	RéPLICATION synchrone asymétrique	157
4.2.5	RéPLICATION synchrone symétrique	158
4.2.6	Diffusion des modifications	159
4.3	RéPLICATION interne physique	161
4.3.1	Log Shipping	162
4.3.2	Streaming replication	163
4.3.3	<i>Warm Standby</i>	163
4.3.4	<i>Hot Standby</i>	164
4.3.5	Exemple	165
4.3.6	RéPLICATION interne	166
4.3.7	RéPLICATION en cascade	167

4.4	RéPLICATION interne logique	168
4.4.1	RéPLICATION logique - Fonctionnement	168
4.5	RéPLICATION externe	170
4.6	Sharding	171
4.7	RéPLICATION bas niveau	173
4.7.1	RAID	173
4.7.2	DRBD	174
4.7.3	SAN Mirroring	174
4.8	Conclusion	175
4.8.1	Questions	175
4.9	Quiz	176
5/ RéPLICATION physique : fondamentaux		177
5.1	Introduction	178
5.1.1	Au menu	178
5.1.2	Objectifs	178
5.2	Concepts / principes	179
5.2.1	Principales évolutions de la réPLICATION	179
5.2.2	Avantages	181
5.2.3	Inconvénients	181
5.3	Mise en place de la réPLICATION par streaming	183
5.3.1	Serveur primaire (1/2) - configuration	183
5.3.2	Serveur primaire (2/2) - authentification	185
5.3.3	Serveur secondaire (1/3) - copie des données	186
5.3.4	Serveur secondaire (2/3) - configuration	187
5.3.5	Serveur secondaire (3/3) - démarrage	188
5.3.6	Processus	189
5.4	Promotion	190
5.4.1	Attention au split-brain !	190
5.4.2	Vérification avant promotion	191
5.4.3	Promotion du standby : méthode	192
5.4.4	Promotion du standby : déroulement	192
5.4.5	Opérations après promotion du standby	193
5.4.6	Retour à l'état stable	194
5.4.7	Retour à l'état stable, suite	195
5.5	Conclusion	197
5.6	Quiz	198
5.7	Travaux pratiques	199
5.7.1	RéPLICATION asynchrone en flux avec un seul secondaire	199
5.7.2	Promotion de l'instance secondaire	199
5.7.3	Retour à la normale	200
5.8	Travaux pratiques (solutions)	201
5.8.1	RéPLICATION asynchrone en flux avec un seul secondaire	201
5.8.2	Promotion de l'instance secondaire	204
5.8.3	Retour à la normale	205

6/ RéPLICATION PHYSIQUE AVANCÉE	209
6.1 Introduction	210
6.1.1 Au menu	210
6.2 Supervision (streaming)	211
6.3 Supervision (log shipping)	214
6.4 Gestion des conflits	215
6.5 Contrôle de la réPLICATION	217
6.6 Asynchrone ou synchrone	218
6.7 RéPLICATION en cascade	223
6.8 Décrochage d'un secondaire	224
6.8.1 Sécurisation par log shipping	225
6.8.2 Slot de réPLICATION : mise en place	227
6.8.3 Slot de réPLICATION : avantages & risques	228
6.9 Synthèse des paramètres	230
6.9.1 Serveur primaire	230
6.9.2 Serveur secondaire	231
6.10 Conclusion	232
6.10.1 Questions	232
6.11 Quiz	233
6.12 Travaux pratiques	234
6.12.1 RéPLICATION asynchrone en flux avec deux secondaires	234
6.12.2 Slots de réPLICATION	234
6.12.3 Log shipping	235
6.12.4 RéPLICATION synchrone en flux avec trois secondaires	235
6.12.5 RéPLICATION synchrone : cohérence des lectures (optionnel)	236
6.13 Travaux pratiques (solutions)	238
6.13.1 RéPLICATION asynchrone en flux avec deux secondaires	238
6.13.2 Slots de réPLICATION	239
6.13.3 Log shipping	242
6.13.4 RéPLICATION synchrone en flux avec trois secondaires	245
6.13.5 RéPLICATION synchrone : cohérence des lectures (optionnel)	250
7/ LES OUTILS DE RÉPLICATION	255
7.1 Introduction	256
7.1.1 Au menu	256
7.2 (Re)construire un secondaire	257
7.2.1 (Re)construction d'un secondaire : pg_basebackup	257
7.2.2 (Re)construction d'un secondaire : script rsync	258
7.2.3 (Re)construction d'un secondaire : outil PITR	260
7.2.4 pg_rewind	261
7.3 Log shipping & PITR	263
7.3.1 pgBackRest	263
7.3.2 barman	264
7.4 Promotion automatique	265
7.4.1 Patroni	266

7.4.2	repmgr	267
7.4.3	Pacemaker	267
7.4.4	PAF	268
7.5	Conclusion	269
7.5.1	Questions	269
7.6	Travaux pratiques	270
7.6.1	(Pré-requis) Environnement	270
7.6.2	Promotion d'une instance secondaire	271
7.6.3	Suivi de timeline	272
7.6.4	pg_rewind	272
7.6.5	pgBackRest	273
7.7	Travaux pratiques (solutions)	274
7.7.1	(Pré-requis) Environnement	274
7.7.2	Promotion d'une instance secondaire	275
7.7.3	Suivi de timeline	277
7.7.4	pg_rewind	278
7.7.5	pgBackRest	281
8/	RéPLICATION LOGIQUE	285
8.1	Objectifs	286
8.1.1	Au menu	286
8.2	Principes de la réPLICATION logique native	287
8.2.1	RéPLICATION physique vs. logique	287
8.2.2	Quelques termes essentiels	288
8.2.3	RéPLICATION en streaming	289
8.2.4	Granularité	290
8.2.5	Possibilités sur les tables répliquées	291
8.2.6	Limitations de la réPLICATION logique	292
8.3	Mise en place	294
8.3.1	Configurer le serveur origine	295
8.3.2	Configuration du serveur destination	296
8.3.3	Créer une publication	297
8.3.4	Souscrire à une publication	298
8.3.5	Options de la souscription (1/2)	299
8.3.6	Options de la souscription (2/2)	300
8.4	Mise en place : exemple	301
8.4.1	Serveurs et schéma	301
8.4.2	RéPLICATION complète	302
8.4.3	Configuration du serveur origine (1/2)	302
8.4.4	Configuration du serveur origine (2/2)	303
8.4.5	Configuration des 4 serveurs destinations	303
8.4.6	Créer une publication complète	304
8.4.7	Souscrire à la publication	304
8.4.8	Tests de la réPLICATION complète	305
8.4.9	RéPLICATION partielle	306

8.4.10	RéPLICATION croisée	307
8.4.11	RéPLICATION de t3_1 de s1 vers s4	308
8.4.12	RéPLICATION de t3_2 de s4 vers s1	309
8.4.13	Tests de la réPLICATION croisée	309
8.5	Administration	311
8.5.1	Processus	311
8.5.2	Synthèse des paramètres (serveur origine)	313
8.5.3	Synthèse des paramètres indispensables	314
8.5.4	Synthèse des paramètres utiles	314
8.5.5	Fichiers (serveur origine)	314
8.5.6	Empêcher les écritures sur un serveur destination	315
8.5.7	Que faire pour les DDL ?	317
8.5.8	Que faire pour les nouvelles tables ?	318
8.5.9	Gérer les opérations de maintenance	319
8.5.10	Gérer les sauvegardes & restaurations logiques	320
8.5.11	Gérer les bascules & les restaurations physiques	322
8.6	Supervision	324
8.6.1	Catalogues systèmes - métadonnées	324
8.6.2	Vues statistiques	326
8.6.3	Outils de supervision	328
8.7	Migration majeure par réPLICATION logique	330
8.8	Rappel des limitations de la réPLICATION logique native	332
8.9	Outils de réPLICATION logique externe	334
8.9.1	Slony : Carte d'identité	334
8.9.2	Slony : Fonctionnalités	334
8.9.3	Slony : Technique	335
8.9.4	Slony : Points forts	335
8.9.5	Slony : Limites	336
8.9.6	Slony : Utilisations	336
8.9.7	Bucardo : Carte d'identité	337
8.9.8	Bucardo : Limites	338
8.10	Conclusion	339
8.10.1	Questions	339
8.11	Quiz	340
8.12	Travaux pratiques	341
8.12.1	Pré-requis	341
8.12.2	RéPLICATION complète d'une base	342
8.12.3	RéPLICATION partielle d'une base	343
8.12.4	RéPLICATION croisée	344
8.12.5	RéPLICATION et partitionnement	344
8.13	Travaux pratiques (solutions)	346
8.13.1	Pré-requis	346
8.13.2	RéPLICATION complète d'une base	347
8.13.3	RéPLICATION partielle d'une base	350
8.13.4	RéPLICATION croisée	351

8.13.5 RéPLICATION et partitionnement	353
Les formations Dalibo	357
Cursus des formations	357
Les livres blancs	358
Téléchargement gratuit	358

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 oubliés, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse [formation@dalibo.com¹](mailto:formation@dalibo.com) !

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>

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 Tachoirs, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

¹<mailto:formation@dalibo.com>

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA²**. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

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

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

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques.

Cette formation peut également contenir quelques images dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées³ par PostgreSQL Community Association of Canada.

Sur ce document

Formation	Formation DBA3
Titre	PostgreSQL Sauvegardes et RéPLICATION
Révision	23.09
ISBN	N/A
PDF	https://dali.bo/dba3_pdf

²<http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

³<https://www.postgresql.org/about/policies/trademarks/>

EPUB	https://dali.bo/dba3_epub
-------------	---

HTML	https://dali.bo/dba3_html
-------------	---

Slides	https://dali.bo/dba3_slides
---------------	---

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

1/ PostgreSQL : Politique de sauvegarde



Photo de l'incendie du datacenter OVHcloud à Strasbourg du 10 mars 2021 fournie gracieusement par l'ITBR67¹.

Cet incendie a provoqué de nombreux arrêts et pertes de données dans toute la France et ailleurs².

¹<https://itbr67.fr/>

²https://fr.wikipedia.org/wiki/Incendie_du_centre_de_donn%C3%A9es_d%27OVHcloud_%C3%A0_Strasbourg

1.1 INTRODUCTION



- Le pire peut arriver
- Politique de sauvegarde

1.1.1 Au menu



- Objectifs
- Approche
- Points d'attention

1.2 DÉFINIR UNE POLITIQUE DE SAUVEGARDE



- Pourquoi établir une politique ?
- Que sauvegarder ?
- À quelle fréquence sauvegarder les données ?
- Quels supports ?
- Quels outils ?
- Vérifier la restauration des sauvegardes

Afin d'assurer la sécurité des données, il est nécessaire de faire des sauvegardes régulières.

Ces sauvegardes vont servir, en cas de problème, à restaurer les bases de données dans un état le plus proche possible du moment où le problème est survenu.

Cependant, le jour où une restauration sera nécessaire, il est possible que la personne qui a mis en place les sauvegardes ne soit pas présente. C'est pour cela qu'il est essentiel d'écrire et de maintenir un document qui indique la mise en place de la sauvegarde et qui détaille comment restaurer une sauvegarde.

En effet, suivant les besoins, les outils pour sauvegarder, le contenu de la sauvegarde, sa fréquence ne seront pas les mêmes.

Par exemple, il n'est pas toujours nécessaire de tout sauvegarder. Une base de données peut contenir des données de travail, temporaires et/ou faciles à reconstruire, stockées dans des tables standards. Il est également possible d'avoir une base dédiée pour stocker ce genre d'objets. Pour diminuer le temps de sauvegarde (et du coup de restauration), il est possible de sauvegarder partiellement son serveur pour ne conserver que les données importantes.

La fréquence peut aussi varier. Un utilisateur peut disposer d'un serveur PostgreSQL pour un entrepôt de données, serveur qu'il n'alimente qu'une fois par semaine. Dans ce cas, il est inutile de sauvegarder tous les jours. Une sauvegarde après chaque alimentation (donc chaque semaine) est suffisante. En fait, il faut déterminer la fréquence de sauvegarde des données selon :

- le volume de données à sauvegarder et/ou restaurer ;
- la criticité des données ;
- la quantité de données qu'il est « acceptable » de perdre en cas de problème.

Le support de sauvegarde est lui aussi très important. Il est possible de sauvegarder les données sur un disque réseau (à travers SMB/CIFS ou NFS), sur des disques locaux dédiés, sur des bandes ou tout autre support adapté. Dans tous les cas, il est fortement déconseillé de stocker les sauvegardes sur les disques utilisés par la base de données.

Ce document doit aussi indiquer comment effectuer la restauration. Si la sauvegarde est composée de plusieurs fichiers, l'ordre de restauration des fichiers peut être essentiel. De plus, savoir où se trouvent les sauvegardes permet de gagner un temps important, qui évitera une immobilisation trop longue.

De même, vérifier la restauration des sauvegardes de façon régulière est une précaution très utile.

1.2.1 Objectifs



- Sécuriser les données
- Mettre à jour le moteur de données
- Dupliquer une base de données de production
- Archiver les données

L'objectif essentiel de la sauvegarde est la sécurisation des données. Autrement dit, l'utilisateur cherche à se protéger d'une panne matérielle ou d'une erreur humaine (un utilisateur qui supprimerait des données essentielles). La sauvegarde permet de restaurer les données perdues. Mais ce n'est pas le seul objectif d'une sauvegarde.

Une sauvegarde peut aussi servir à dupliquer une base de données sur un serveur de test ou de pré-production. Elle permet aussi d'archiver des tables. Cela se voit surtout dans le cadre des tables partitionnées où l'archivage de la table la plus ancienne permet ensuite sa suppression de la base pour gagner en espace disque.

Un autre cas d'utilisation de la sauvegarde est la mise à jour majeure de versions PostgreSQL. Il s'agit de la solution historique de mise à jour (export/import). Historique, mais pas obsolète.

1.2.2 Différentes approches



- Sauvegarde à froid des fichiers (ou physique)
- Sauvegarde à chaud en SQL (ou logique)
- Sauvegarde à chaud des fichiers (PITR)

À ces différents objectifs vont correspondre différentes approches de la sauvegarde.

La sauvegarde au niveau système de fichiers permet de conserver une image cohérente de l'intégralité des répertoires de données d'une instance arrêtée. C'est la sauvegarde à froid. Cependant, l'utilisation d'outils de snapshots pour effectuer les sauvegardes peut accélérer considérablement les temps de sauvegarde des bases de données, et donc diminuer d'autant le temps d'immobilisation du système.

La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données.

La sauvegarde à chaud des fichiers est possible avec le *Point In Time Recovery*.

Suivant les prérequis et les limitations de chaque méthode, il est fort possible qu'une seule de ces solutions soit utilisable. Par exemple :

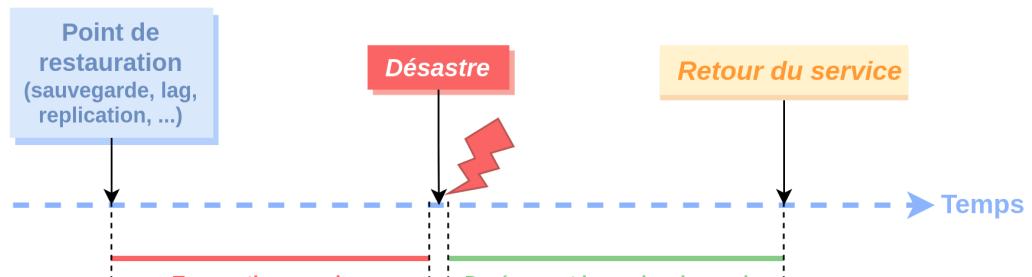
- si le serveur ne peut pas être arrêté, la sauvegarde à froid est exclue d'office ;
- si la base de données est très volumineuse, la sauvegarde logique devient très longue ;
- si l'espace disque est limité et que l'instance génère beaucoup de journaux de transactions, la sauvegarde PITR sera difficile à mettre en place.

1.2.3 RTO/RPO



La politique de sauvegarde découle du :

- **RPO** (*Recovery Point Objective*) : Perte de Données Maximale Admissible
 - faible ou importante ?
- **RTO** (*Recovery Time Objective*) : Durée Maximale d'Interruption Admissible
 - courte ou longue ?



Le RPO et RTO sont deux concepts déterminants dans le choix des politiques de sauvegardes.

La **RPO** (ou PDMA³) est la perte de données maximale admissible, ou quantité de données que l'on peut tolérer de perdre lors d'un sinistre majeur, souvent exprimée en heures ou minutes.

Pour un système mis à jour épisodiquement ou avec des données non critiques, ou facilement récupérables, le RPO peut être important (par exemple une journée). Peuvent alors s'envisager des solutions comme :

- les sauvegardes logiques (dump) ;
- les sauvegardes des fichiers à froid.

³https://fr.wikipedia.org/wiki/Perte_de_donn%C3%A9es_maximale_admissible

Dans beaucoup de cas, la perte de données admissible est très faible (heures, quelques minutes), voire nulle. Il faudra s'orienter vers des solutions de type :

- sauvegarde à chaud ;
- sauvegarde d'instantané à un point donnée dans le temps (PITR) ;
- réPLICATION asynchrone, voire synchrone.

La **RTO** (ou DMIA⁴) est la durée maximale d'interruption du service.

Dans beaucoup de cas, les utilisateurs peuvent tolérer une indisponibilité de plusieurs heures, voire jours. La durée de reprise du service n'est alors pas critique, on peut utiliser des solutions simples comme :

- la restauration des fichiers ;
- la restauration d'une sauvegarde logique (dump).

Si elle est plus courte, le service doit très vite remonter. Cela nécessite des procédures avec un minimum d'acteurs et de manipulation :

- réPLICATION ;
- solutions HA (Haute Disponibilité).

Plus le besoin en RTO/RPO sera court, plus les solutions seront complexes à mettre en œuvre — et chères. Inversement, pour des données non critiques, un RTO/RPO long permet d'utiliser des solutions simples et peu coûteuses.

1.2.4 Industrialisation



- Évaluer les coûts humains et matériels
- Intégrer les méthodes de sauvegardes avec le reste du SI
 - sauvegarde sur bande centrale
 - supervision
 - plan de continuité et de reprise d'activité

Les moyens nécessaires pour la mise en place, le maintien et l'intégration de la sauvegarde dans le SI ont un coût financier qui apporte une contrainte supplémentaire sur la politique de sauvegarde.

Du point de vue matériel, il faut disposer principalement d'un volume de stockage qui peut devenir conséquent. Cela dépend de la volumétrie à sauvegarder, il faut considérer les besoins suivants :

⁴https://fr.wikipedia.org/wiki/Dur%C3%A9e_maximale_d%27interruption_admissible#RTO

- Stocker plusieurs sauvegardes. Même avec une rétention d'une sauvegarde, il faut pouvoir stocker la suivante durant sa création : il vaut mieux purger les anciennes sauvegardes une fois qu'on est sûr que la sauvegarde s'est correctement déroulée.
- Avoir suffisamment de place pour restaurer sans avoir besoin de supprimer la base ou l'instance en production. Un tel espace de travail est également intéressant pour réaliser des restaurations partielles. Cet espace peut être mutualisé. On peut utiliser également le serveur de pré-production s'il dispose de la place suffisante.

Avec une rétention d'une sauvegarde unique, il est bon de prévoir 3 fois la taille de la base ou de l'instance. Pour une faible volumétrie, cela ne pose pas de problèmes, mais quand la volumétrie devient de l'ordre du téraoctet, les coûts augmentent significativement.

L'autre poste de coût est la mise en place de la sauvegarde. Une équipe de DBA peut tout à fait décider de créer ses propres scripts de sauvegarde et restauration, pour diverses raisons, notamment :

- maîtrise complète de la sauvegarde, maintien plus aisés du code ;
- intégration avec les moyens de sauvegardes communs au SI (bandes, externalisation...) ;
- adaptation au PRA/PCA plus fine.

Enfin, le dernier poste de coût est la maintenance, à la fois des scripts et par le test régulier de la restauration.

1.2.5 Documentation



- Documenter les éléments clés de la politique :
 - perte de données
 - rétention
 - temps de référence
- Documenter les processus de sauvegarde et restauration
- Imposer des révisions régulières des procédures

Comme pour n'importe quelle procédure, il est impératif de documenter la politique de sauvegarde, les procédures de sauvegarde et de restauration ainsi que les scripts.

Au strict minimum, la documentation doit permettre à un DBA non familier de l'environnement de comprendre la sauvegarde, retrouver les fichiers et restaurer les données le cas échéant, le plus rapidement possible et sans laisser de doute. En effet, en cas d'avarie nécessitant une restauration, le service aux utilisateurs finaux est généralement coupé, ce qui génère un climat de pression propice aux erreurs qui ne font qu'empirer la situation.

L'idéal est de réviser la documentation régulièrement en accompagnant ces révisions de tests de restauration : avoir un ordre de grandeur de la durée d'une restauration est primordial. On demandera toujours au DBA qui restaure une base ou une instance combien de temps cela va prendre.

1.2.6 Règle 3-2-1



- 3 exemplaires des données
- 2 sur différents médias
- 1 hors site (et hors ligne)
- Un RAID n'est pas une sauvegarde !
- Le *cloud* n'est pas une solution magique !

L'un des points les plus importants à prendre en compte est l'endroit où sont stockés les fichiers des sauvegardes. Laisser les sauvegardes sur la même machine n'est pas suffisant : si une défaillance matérielle se produisait, les sauvegardes pourraient être perdues en même temps que l'instance sauvegardée, rendant ainsi la restauration impossible.

Il est conseillé de suivre au moins la règle 3-2-1. Les données elles-mêmes sont le premier exemplaire.

Les deux copies doivent se trouver sur des supports physiques différents (et de préférence sur un autre serveur) pour parer à la destruction du support original (notamment une perte de disques durs). La première copie peut être à proximité pour faciliter une restauration.



Des disques en RAID ne sont pas une sauvegarde ! Ils peuvent parer à la défaillance d'un disque, pas à une fausse manipulation (`rm -rf /` ou TRUNCATE malheureux). La perte de la carte contrôleur peut entraîner la perte de toute la grappe. Un conseil courant est d'ailleurs de choisir des disques de séries différentes pour éviter des défaillances simultanées.

Le troisième exemplaire doit se trouver à un autre endroit, pour parer aux scénarios les plus catastrophiques (cambriolage, incendie...). Selon la criticité, le délai nécessaire pour remonter rapidement un système fonctionnel, et le budget, ce troisième exemplaire peut être une copie manuelle sur un disque externe stocké dans un coffre, ou une infrastructure répliquée complète avec sa copie des sauvegardes à l'autre bout de la ville, voire du pays.

Pour limiter la consommation d'espace disque des copies multiples, les durées de rétention peuvent différer. La dernière sauvegarde peut résider sur la machine, les 5 dernières sur un serveur distant, des bandes être déposées dans un site sécurisé tous les mois.



Stocker vos données dans le *cloud* n'est pas une solution miracle. Un *datacenter* peut aussi brûler⁵, être sujet à une attaque, ou perdre durablement alimentation ou électricité. Dans la formule choisie, il faut bien vérifier que le fournisseur sauvegarde les données sur un autre site, ou s'assurer de le faire soi-même.

Il ne faut pas négliger le risque d'une attaque (classique ou par *ransomware*...), qui s'en prendra aussi aux sauvegardes accessibles en ligne. Une copie physique hors ligne est donc chaudement recommandée pour les données les plus critiques.

1.2.7 Autres points d'attention



- Sauvegarder les fichiers de configuration
- **Tester la restauration**
 - De nombreuses catastrophes auraient pu être évitées avec un test
 - Estimation de la durée

La sauvegarde ne concerne pas uniquement les données. Il est également fortement conseillé de sauvegarder les fichiers de configuration du serveur et les scripts d'administration. L'idéal est de les copier avec les sauvegardes. On peut également déléguer cette tâche à une sauvegarde au niveau système, vu que ce sont de simples fichiers. Les principaux fichiers de PostgreSQL à prendre en compte sont : `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`, ainsi que `recovery.conf` (pour les serveurs répliqués antérieurs à la version 12). Cette liste n'est en aucun cas exhaustive.

Il s'agit donc de recenser l'ensemble des fichiers et scripts nécessaires si l'on désirait recréer le serveur depuis zéro.

Enfin, même si les sauvegardes se déroulent correctement, il est indispensable de tester si elles se restaurent sans erreur. Une erreur de copie lors de l'externalisation peut, par exemple, rendre la sauvegarde inutilisable.



Just that backup tapes are seen to move, backup scripts are run for a lengthy period of time, should not be construed as verifying that data backups are properly being performed.

Que l'on voit bouger les bandes de sauvegardes, ou que les scripts de sauvegarde fonctionnent pendant une longue période, ne doit pas être interprété comme une validation que les sauvegardes sont faites.

NASA, *Lessons Learned*, Lesson 1781⁶

Le test de restauration permet de vérifier l'ensemble de la procédure : ensemble des objets sauvegardés, intégrité de la copie, commandes à passer pour une restauration complète (en cas de stress, vous en oublierez probablement une partie). De plus, cela permet d'estimer la durée nécessaire à la restauration.

Nous rencontrons régulièrement en clientèle des scripts de sauvegarde qui ne fonctionnent pas, et jamais testés. Vous trouverez sur Internet de nombreuses histoires de catastrophes qui auraient été évitées par un simple test. Entre mille autres :

- une base disparaît à l'arrêt et ses sauvegardes sont vides⁷ : erreur de manipulation, script ne vérifiant pas qu'il a bien fait sa tâche, monitoring défaillant ;
- perte de 6 heures de données chez Gitlab en 2017⁸ : procédures de sauvegarde et de réPLICATION incomplètes, complexes et peu claires, outils mal choisis ou mal connus, sauvegardes vides et jamais testées, distraction d'un opérateur — Gitlab a le mérite d'avoir tout soigneusement documenté⁹.

Restaurer régulièrement les bases de test ou de préproduction à partir des sauvegardes de la production est une bonne idée. Cela vous évitera de découvrir la procédure dans l'urgence, le stress, voire la panique, alors que vous serez harcelé par de nombreux utilisateurs ou clients bloqués.

⁷http://www.pgdba.org/post/restoring_data/#a-database-horror-story

⁸<https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/>

⁹<https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/>

1.3 CONCLUSION



- Les techniques de sauvegarde de PostgreSQL sont :
 - complémentaires
 - automatisables
- La maîtrise de ces techniques est indispensable pour assurer un service fiable.
- **Testez vos sauvegardes !**

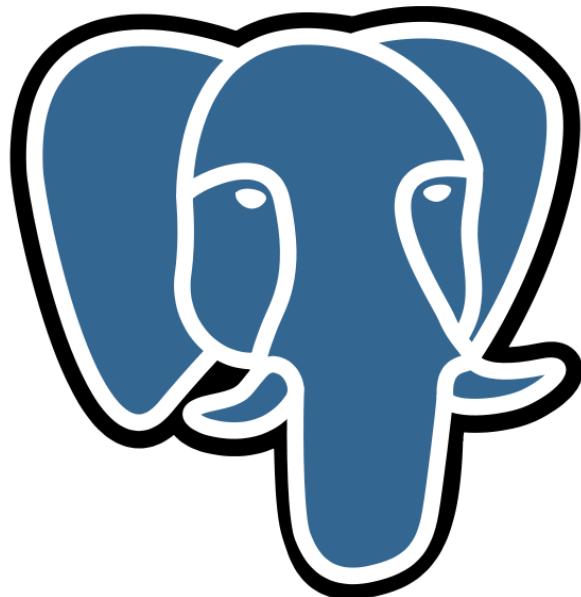
L'écosystème de PostgreSQL offre tout le nécessaire pour effectuer des sauvegardes fiables. Le plan de sauvegarde doit être fait sérieusement, et les sauvegardes testées. Cela a un coût, mais un désastre détruisant toutes vos données sera incommensurablement plus ruineux.

1.4 QUIZ



https://dali.bo/i0_quiz

2/ Sauvegarde physique à chaud et PITR



2.1 INTRODUCTION



- Sauvegarde traditionnelle
 - sauvegarde pg_dump à chaud
 - sauvegarde des fichiers à froid
- Insuffisant pour les grosses bases
 - long à sauvegarder
 - encore plus long à restaurer
- Perte de données potentiellement importante
 - car impossible de réaliser fréquemment une sauvegarde
- Une solution : la sauvegarde PITR

La sauvegarde traditionnelle, qu'elle soit logique ou physique à froid, répond à beaucoup de besoins. Cependant, ce type de sauvegarde montre de plus en plus ses faiblesses pour les gros volumes : la sauvegarde est longue à réaliser et encore plus longue à restaurer. Et plus une sauvegarde met du temps, moins fréquemment on l'exécute. La fenêtre de perte de données devient plus importante.

PostgreSQL propose une solution à ce problème avec la sauvegarde physique à chaud. On peut l'utiliser comme un simple mode de sauvegarde supplémentaire, mais elle permet bien d'autres possibilités, d'où le nom de PITR (*Point In Time Recovery*).

2.1.1 Au menu



- Mettre en place la sauvegarde PITR
 - sauvegarde : manuelle, ou avec pg_basebackup
 - archivage : manuel, ou avec pg_receivewal
- Restaurer une sauvegarde PITR
- Des outils

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (de manière manuelle ou avec l'outil pg_receivewal) à la sauvegarde des fichiers (là aussi, en manuel, ou avec l'outil pg_basebackup). Il discute aussi de la restauration d'une telle sauvegarde. Nous évoquerons très rapidement quelques outils externes pour faciliter ces sauvegardes.

NB : pg_recvwal s'appelait pg_receivexlog avant PostgreSQL 10.

2.2 PITR



- *Point In Time Recovery*
- À chaud
- En continu
- Cohérente

PITR est l'acronyme de *Point In Time Recovery*, autrement dit restauration à un point dans le temps.

C'est une sauvegarde à chaud et surtout en continu. Là où une sauvegarde logique du type pg_dump se fait au mieux une fois toutes les 24 h, la sauvegarde PITR se fait en continu grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

2.2.1 Principes



- Les journaux de transactions contiennent toutes les modifications
- Il faut les archiver
 - ...et avoir une image des fichiers à un instant t
- La restauration se fait en restaurant cette image
 - ...et en rejouant les journaux
 - dans l'ordre
 - entièrement
 - ou partiellement (*i.e.* jusqu'à un certain moment)

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t. Toutes les actions effectuées après cet instant t pourront être rejouées en demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec pg_dump n'est pas utilisable.

Il est donc nécessaire de conserver ces journaux de transactions. Or PostgreSQL les recycle dès qu'il n'en a plus besoin. La solution est de demander au moteur de les archiver ailleurs avant ce recyclage. On doit aussi disposer de l'ensemble des fichiers qui composent le répertoire des données (incluant les tablespaces si ces derniers sont utilisés).

La restauration a besoin des journaux de transactions archivés. Il ne sera pas possible de restaurer et éventuellement revenir à un point donné avec la sauvegarde seule. En revanche, une fois la sauvegarde des fichiers restaurée et la configuration réalisée pour rejouer les journaux archivés, il sera possible de les rejouer tous ou seulement une partie d'entre eux (en s'arrêtant à un certain moment). Ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom).

2.2.2 Avantages



- Sauvegarde à chaud
- Rejou d'un grand nombre de journaux
- Moins de perte de données

Tout le travail est réalisé à chaud, que ce soit l'archivage des journaux ou la sauvegarde des fichiers de la base. En effet, il importe peu que les fichiers de données soient modifiés pendant la sauvegarde car les journaux de transactions archivés permettront de corriger toute incohérence par leur application.

Il est possible de rejouer un très grand nombre de journaux (une journée, une semaine, un mois, etc.). Évidemment, plus il y a de journaux à appliquer, plus cela prendra du temps. Mais il n'y a pas de limite au nombre de journaux à rejouer.

Dernier avantage, c'est le système de sauvegarde qui occasionnera le moins de perte de données. Généralement, une sauvegarde pg_dump s'exécute toutes les nuits, disons à 3 h du matin. Supposons qu'un gros problème survienne à midi. S'il faut restaurer la dernière sauvegarde, la perte de données sera de 9 h. Le volume maximum de données perdu correspond à l'espacement des sauvegardes. Avec l'archivage continu des journaux de transactions, la fenêtre de perte de données va être fortement réduite. Plus l'activité est intense, plus la fenêtre de temps sera petite : il faut changer de fichier de journal pour que le journal précédent soit archivé et les fichiers de journaux sont de taille fixe.

Pour les systèmes n'ayant pas une grosse activité, il est aussi possible de forcer un changement de journal à intervalle régulier, ce qui a pour effet de forcer son archivage, et donc dans les faits de pouvoir s'assurer une perte correspondant au maximum à cet intervalle.

2.2.3 Inconvénients



- Sauvegarde de l'instance complète
- Nécessite un grand espace de stockage (données + journaux)
- Risque d'accumulation des journaux
 - dans pg_wal en cas d'échec d'archivage... avec arrêt si il est plein !
 - dans le dépôt d'archivage si échec des sauvegardes
- Restauration de l'instance complète
- Impossible de changer d'architecture (même OS conseillé)
- Plus complexe

Certains inconvénients viennent directement du fait qu'on copie les fichiers : sauvegarde et restauration complète (impossible de ne restaurer qu'une seule base ou que quelques tables), restauration sur la même architecture (32/64 bits, *little/big endian*). Il est même fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance (différence de définition des locales notamment).

Elle nécessite en plus un plus grand espace de stockage car il faut sauvegarder les fichiers (dont les index) ainsi que les journaux de transactions sur une certaine période, ce qui peut être volumineux (en tout cas beaucoup plus que des pg_dump).

En cas de problème dans l'archivage et selon la méthode choisie, l'instance ne voudra pas effacer les journaux non archivés. Il y a donc un risque d'accumulation de ceux-ci. Il faudra donc surveiller la taille du pg_wal. En cas de saturation, PostgreSQL s'arrête !

Enfin, la sauvegarde PITR est plus complexe à mettre en place qu'une sauvegarde pg_dump. Elle nécessite plus d'étapes, une réflexion sur l'architecture à mettre en œuvre et une meilleure compréhension des mécanismes internes à PostgreSQL pour en avoir la maîtrise.

2.3 COPIE PHYSIQUE À CHAUD PONCTUELLE AVEC PG_BASEBACKUP



- Réalise les différentes étapes d'une sauvegarde
 - via 1 ou 2 connexions de réPLICATION + slots de réPLICATION
 - base backup + journaux
- Copie intégrale,
 - image de la base à la **fin** du backup
- Fichier manifeste (v13+)
- Compresse la sauvegarde côté serveur ou client (v15+)
- Pas d'incrémental
- Configuration : *streaming* (rôle, droits, slots)

```
$ pg_basebackup --format=tar --wal-method=stream \
--checkpoint=fast --progress -h 127.0.0.1 -U sauve \
-D /var/lib/postgresql/backups/
```

Description :

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut depuis la version 10 le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de la base, à distance, via deux connexions de réPLICATION : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie. Sa compression permet d'éviter une durée de transfert ou une place disque occupée trop importante. Cela a évidemment un coût, notamment au niveau CPU, sur le serveur ou sur le client suivant le besoin. Il est simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes que nous verrons par la suite.

Par contre, il ne permet pas de réaliser une sauvegarde incrémentale, et ne permet pas de continuer à archiver les journaux, contrairement aux outils de PITR classiques. Cependant, ceux-ci peuvent l'utiliser pour réaliser la première copie des fichiers d'une instance.

Mise en place :

`pg_basebackup` nécessite des connexions de réPLICATION. Il peut utiliser un slot de réPLICATION, une technique qui fiabilise la sauvegarde ou la réPLICATION en indiquant à l'instance quels journaux elle doit conserver. Par défaut, tout est en place pour cela dans PostgreSQL 10 et suivants pour une connexion en local :

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Ensuite, il faut configurer le fichier pg_hba.conf pour accepter la connexion du serveur où est exécutée pg_basebackup. Dans notre cas, il s'agit du même serveur avec un utilisateur dédié :

```
host replication sauve 127.0.0.1/32 scram-sha-256
```

Enfin, il faut créer un utilisateur dédié à la réPLICATION (ici sauve) qui sera le rôle créant la connexion et lui attribuer un mot de passe :

```
CREATE ROLE sauve LOGIN REPLICATION;  
\password sauve
```

Dans un but d'automatisation, le mot de passe finira souvent dans un fichier .pgpass ou équivalent.

Il ne reste plus qu'à :

- lancer pg_basebackup, ici en lui demandant une archive au format tar ;
- archiver les journaux en utilisant une connexion de réPLICATION par *streaming* ;
- forcer le *checkpoint*.

Cela donne la commande suivante, ici pour une sauvegarde en local :

```
$ pg_basebackup --format=tar --wal-method=stream \  
--checkpoint=fast --progress -h 127.0.0.1 -U sauve \  
-D /var/lib/postgresql/backups/
```

```
644320/644320 kB (100%), 1/1 tablespace
```

Le résultat est ici un ensemble des deux archives : les journaux sont à part et devront être dépaquetés dans le pg_wal à la restauration.

```
$ ls -l /var/lib/postgresql/backups/  
total 4163772  
-rw----- 1 postgres postgres 659785216 Oct  9 11:37 base.tar  
-rw----- 1 postgres postgres   16780288 Oct  9 11:37 pg_wal.tar
```

La cible doit être vide. En cas d'arrêt avant la fin, il faudra tout recommencer de zéro, c'est une limite de l'outil.

Restauration :

Pour restaurer, il suffit de remplacer le PGDATA corrompu par le contenu de l'archive, ou de créer une nouvelle instance et de remplacer son PGDATA par cette sauvegarde. Au démarrage, l'instance repérera qu'elle est une sauvegarde restaurée et réappliquera les journaux. L'instance contiendra les données telles qu'elles étaient à la **fin** du pg_basebackup.

Noter que les fichiers de configuration ne sont PAS inclus si ils ne sont pas dans le PGDATA, notamment sur Debian et ses versions dérivées.

Differences entre les versions :

À partir de la v10, un slot temporaire sera créé par défaut pour garantir que le serveur gardera les journaux jusqu'à leur copie intégrale.

À partir de la version 13, la commande pg_basebackup crée un fichier manifeste contenant la liste des fichiers sauvegardés, leur taille et une somme de contrôle. Cela permet de vérifier la sauvegarde

avec l'outil `pg_verifybackup` (ce dernier ne fonctionne hélas que sur une sauvegarde au format plain, ou décompressée).

Lisez bien la documentation de `pg_basebackup`¹ pour votre version précise de PostgreSQL, des options ont changé de nom au fil des versions.



Même avec un serveur un peu ancien, il est possible d'installer un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

L'outil est développé plus en détail dans notre module I4².

¹<https://docs.postgresql.fr/current/app-pgbasebackup.html>

²https://dali.bo/i4_html

2.4 SAUVEGARDE PITR



2 étapes :

- Archivage des journaux de transactions
 - archivage interne
 - ou outil pg_receivewal
- Sauvegarde des fichiers
 - pg_basebackup
 - ou manuellement (outils de copie classiques)

Même si la mise en place est plus complexe qu'un pg_dump, la sauvegarde PITR demande peu d'étapes. La première chose à faire est de mettre en place l'archivage des journaux de transactions. Un choix est à faire entre un archivage classique et l'utilisation de l'outil pg_receivewal.

Lorsque cette étape est réalisée (et fonctionnelle), il est possible de passer à la seconde : la sauvegarde des fichiers. Là-aussi, il y a différentes possibilités : soit manuellement, soit pg_basebackup, soit son propre script ou un outil extérieur.

2.4.1 Méthodes d'archivage



- Deux méthodes :
 - processus interne archiver
 - outil pg_receivewal (flux de réPLICATION)

La méthode historique est la méthode utilisant le processus archiver. Ce processus fonctionne sur le serveur à sauvegarder et est de la responsabilité du serveur PostgreSQL. Seule sa (bonne) configuration incombe au DBA.

Une autre méthode existe : pg_receivewal. Cet outil livré aussi avec PostgreSQL se comporte comme un serveur secondaire. Il reconstitue les journaux de transactions à partir du flux de réPLICATION.

Chaque solution a ses avantages et inconvénients qu'on étudiera après avoir détaillé leur mise en place.

2.4.2 Choix du répertoire d'archivage



- À faire quelle que soit la méthode d'archivage
- Attention aux droits d'écriture dans le répertoire
 - la commande configurée pour la copie doit pouvoir écrire dedans
 - et potentiellement y lire

Dans le cas de l'archivage historique, le serveur PostgreSQL va exécuter une commande qui va copier les journaux à l'extérieur de son répertoire de travail :

- sur un disque différent du même serveur ;
- sur un disque d'un autre serveur ;
- sur des bandes, un CDROM, etc.

Dans le cas de l'archivage avec `pg_recvwal`, c'est cet outil qui va écrire les journaux dans un répertoire de travail. Cette écriture ne peut se faire qu'en local. Cependant, le répertoire peut se trouver dans un montage NFS.

L'exemple pris ici utilise le répertoire `/mnt/nfs1/archivage` comme répertoire de copie. Ce répertoire est en fait un montage NFS. Il faut donc commencer par créer ce répertoire et s'assurer que l'utilisateur Unix (ou Windows) `postgres` peut écrire dedans :

```
# mkdir /mnt/nfs1/archivage  
# chown postgres:postgres /mnt/nfs1/archivage
```

2.4.3 Processus archiver : configuration



- configuration (`postgresql.conf`)
 - `wal_level = replica`
 - `archive_mode = on (ou always)`
 - `archive_command = '... une commande ...'`
 - ou: `archive_library = '... une bibliothèque ...'` (v15+)
 - `archive_timeout = '... min'`
- Ne pas oublier de forcer l'écriture de l'archive sur disque
- Code retour de l'archivage entre 0 (ok) et 125

Après avoir créé le répertoire d'archivage, il faut configurer PostgreSQL pour lui indiquer comment archiver.

Niveau d'archivage :

La valeur par défaut de `wal_level` est adéquate :

```
wal_level = replica
```

Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal`, les journaux ne servent qu'à garantir la cohérence des fichiers de données en cas de crash. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où la nécessité du niveau `replica` (défaut à partir de PostgreSQL 10).

Le niveau `logical`, nécessaire à la réPLICATION logique³, convient également.

Mode d'archivage :

Il s'active ainsi sur une instance seule ou primaire :

```
archive_mode = on
```

(La valeur `always` permet d'archiver depuis un secondaire). Le changement nécessite un redémarrage !

Enfin, une commande d'archivage doit être définie par le paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel. PostgreSQL l'appelle une fois pour chaque fichier WAL, impérativement dans l'ordre des fichiers. En cas d'échec, elle est répétée indéfiniment jusqu'à réussite, avant de passer à l'archivage du fichier suivant. C'est la technique encore la plus utilisée.

(Noter qu'à partir de la version 15, il existe une alternative, avec l'utilisation du paramètre `archive_library`. Il est possible d'indiquer une bibliothèque partagée qui fera ce travail d'archivage. Une telle bibliothèque, écrite en C, devrait être plus puissante et performante. Un module basique est fourni avec PostgreSQL : `basic_archive`⁴). Notre blog présente un exemple fonctionnel de module d'archivage⁵ utilisant une extension en C pour compresser les journaux de transactions. Mais en production, il vaudra mieux utiliser une bibliothèque fournie par un outil PITR reconnu. Cependant, à notre connaissance (en août 2023), aucun outil connu n'utilise encore cette fonctionnalité.)

Exemples d'`archive_command` :

PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transactions suivant son contexte. Si vous utilisez un outil de sauvegarde, la commande vous sera probablement fournie. Une simple commande de copie suffit dans la plupart des cas. La directive `archive_command` peut alors être positionnée comme suit :

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

Le joker `%p` est remplacé par le chemin complet vers le journal de transactions à archiver, alors que le joker `%f` correspond au nom du journal de transactions une fois archivé.

³https://dali.bo/w5_html

⁴<https://docs.postgresql.fr/current/basic-archive.html>

⁵<https://blog.dalibo.com/2023/07/28/hackingpg2.html>

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

La commande d'archivage suivante est donnée dans la documentation officielle à titre d'exemple :

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p
                   ↵ /mnt/server/archivedir/%f'
```



Cette commande a deux inconvénients. Elle ne garantit pas que les données seront synchronisées sur disque. De plus si le fichier existe ou a été copié partiellement à cause d'une erreur précédente, la copie ne s'effectuera pas.

Cette protection est une bonne chose. Cependant, il faut être vigilant lorsque l'on rétablit le fonctionnement de l'archiver suite à un incident ayant provoqué des écritures partielles dans le répertoire d'archive, comme une saturation de l'espace disque.

Il est aussi possible d'y placer le nom d'un script bash, perl ou autre. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque. Il peut aussi être intéressant de tracer l'action de l'archivage par exemple, ou encore de compresser le journal avant archivage.



Il faut s'assurer d'une seule chose : la commande d'archivage doit retourner 0 en cas de réussite et surtout une valeur différente de 0 en cas d'échec.

Si le code retour de la commande est compris entre 1 et 125, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0).

Tant qu'un fichier journal n'est pas considéré comme archivé avec succès, PostgreSQL ne le supprimera ou recyclera pas !

Il ne cherchera pas non plus à archiver les fichiers suivants.



De plus si le code retour de la commande est supérieur à 125, le processus archiver redémarrera, et l'erreur ne sera pas comptabilisée dans la vue pg_stat_archiver ! Ce cas de figure inclut les erreurs de type command not found associées aux codes retours 126 et 127, ou le cas de rsync, qui renvoie un code retour 255 en cas d'erreur de syntaxe ou de configuration du ssh.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur. Les causes d'échec sont nombreuses : problème réseau, montage inaccessible, erreur de

paramétrage de l'outil, droits insuffisants ou expirés, génération de journaux trop rapide...

À titre d'exemple encore, les commandes utilisées par pgBackRest ou barman ressemblent à ceci :

```
# pgBackRest
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'

# barman
archive_command='/usr/bin/barman-wal-archive backup prod %p'
```



Enfin, le paramétrage suivant archive « dans le vide ». Cette astuce est utilisée lors de certains dépannages, ou pour éviter le redémarrage que nécessiterait la désactivation de archive_mode.

```
archive_mode = on
archive_command = '/bin/true'
```

Période minimale entre deux archivages :

Si l'activité en écriture est très réduite, il peut se passer des heures entre deux archivages de journaux. Il est alors conseillé de garantir qu'un archivage aura lieu périodiquement en indiquant un délai maximum entre deux archivages :

```
archive_timeout = '5min'
```

(La valeur par défaut, 0, désactive ce comportement.) Une conséquence sera l'archivage de journaux de transactions partiellement remplis. Comme la taille reste fixe (16 Mo par fichier par défaut), la consommation en terme d'espace disque sera donc plus importante (la compression par l'outil d'archivage compense généralement cela), et le temps de restauration plus long.

2.4.4 Processus archiver : lancement



- Redémarrage de PostgreSQL
 - si modification de wal_level et/ou archive_mode
 - ou recharge de la configuration

Il ne reste plus qu'à indiquer à PostgreSQL de recharger sa configuration pour que l'archivage soit en place (avec SELECT pg_reload_conf(); ou la commande reload adaptée au système). Dans le cas où l'un des paramètres wal_level et archive_mode a été modifié, il faudra relancer PostgreSQL.

2.4.5 Processus archiver : supervision



- Vue pg_stat_archiver
- pg_wal/archive_status/
- Taille de pg_wal
 - si saturation : Arrêt !
- Traces

PostgreSQL archive les journaux impérativement dans l'ordre.



S'il y a un problème d'archivage d'un journal, les suivants ne seront pas archivés non plus, et vont s'accumuler dans pg_wal ! De plus, une saturation de la partition portant pg_wal mènera à l'arrêt de l'instance PostgreSQL !

La supervision se fait de quatre manières complémentaires.

Taille :

Si le répertoire pg_wal commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions : c'est un indicateur d'une commande d'archivage n'arrivant pas à faire son travail pour une raison ou une autre. Ce peut être temporaire si l'archivage est juste lent. Si l'archivage est bloqué, ce répertoire grossira indéfiniment.

Vue pg_stat_archiver :

La vue système pg_stat_archiver indique les derniers journaux archivés et les dernières erreurs. Dans l'exemple suivant, il y a eu un problème pendant quelques secondes, d'où 6 échecs, avant que l'archivage reprenne :

```
# SELECT * FROM pg_stat_archiver \gx
-[ RECORD 1 ]-----+
archived_count | 156
last_archived_wal | 0000000200000001000000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count | 6
last_failed_wal | 0000000200000001000000D7
last_failed_time | 2020-01-17 18:24:24.463038+00
stats_reset | 2020-01-17 16:08:37.980214+00
```

Comme dit plus haut, pour que cette supervision soit fiable, la commande exécutée doit renvoyer un code retour inférieur ou égal à 125. Dans le cas contraire, le processus archiver redémarre et l'erreur n'apparaît pas dans la vue !

Traces :

On trouvera la sortie et surtout les messages d'erreurs du script d'archivage dans les traces (qui dépendent bien sûr du script utilisé) :

```
2020-01-17 18:24:18.427 UTC [15431] LOG: archive command failed with exit code 3
2020-01-17 18:24:18.427 UTC [15431] DETAIL: The failed archive command was:
  rsync pg_wal/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:19.456 UTC [15431] LOG: archive command failed with exit code 3
2020-01-17 18:24:19.456 UTC [15431] DETAIL: The failed archive command was:
  rsync pg_wal/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:20.463 UTC [15431] LOG: archive command failed with exit code 3
```

C'est donc le premier endroit à regarder en cas de souci ou lors de la mise en place de l'archivage.

pg_wal/archive_status :

Enfin, on peut monitorer les fichiers présents dans pg_wal/archive_status. Les fichiers .ready, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester faible et retomber rapidement à 0 ou 1. Le service ready_archives de la sonde Nagios check_pgactivity⁶ se base sur ce répertoire.

```
postgres=# SELECT * FROM pg_ls_dir ('pg_wal/archive_status') ORDER BY 1;

pg_ls_dir
-----
0000000200000001000000DE.done
0000000200000001000000DF.done
0000000200000001000000E0.done
0000000200000001000000E1.ready
0000000200000001000000E2.ready
0000000200000001000000E3.ready
0000000200000001000000E4.ready
0000000200000001000000E5.ready
0000000200000001000000E6.ready
00000002.history.done
```

⁶https://github.com/OPMDG/check_pgactivity

2.4.6 pg_receivewal



- Archivage via le protocole de réPLICATION
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
 - toujours au plus près du primaire
- Slots de réPLICATION obligatoires

`pg_receivewal` est un outil permettant de se faire passer pour un serveur secondaire utilisant la réPLICATION en flux (*streaming replication*) dans le but d'archiver en continu les journaux de transactions. Il fonctionne habituellement sur un autre serveur, où seront archivés les journaux. C'est une alternative à l'archiver.

Comme il utilise le protocole de réPLICATION, les journaux archivés ont un retard bien inférieur à celui induit par la configuration du paramètre `archive_command` ou du paramètre `archive_library`, les journaux de transactions étant écrits au fil de l'eau avant d'être complets. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le serveur primaire. On peut même utiliser une réPLICATION synchrone (paramètres `synchronous_commit` et `synchronous_standby_names`) pour ne perdre aucune transaction, si l'on accepte un impact certain sur la latence des transactions.

Cet outil utilise les mêmes options de connexion que la plupart des outils PostgreSQL, avec en plus l'option `-D` pour spécifier le réPERTOIRE où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs LOGIN et REPLICATION.

Comme il s'agit de conserver toutes les modifications effectuées par le serveur dans le cadre d'une sauvegarde permanente, il est nécessaire de s'assurer qu'on ne puisse pas perdre des journaux de transactions. Il n'y a qu'un seul moyen pour cela : utiliser la technologie des slots de réPLICATION. En utilisant un slot de réPLICATION, `pg_receivewal` s'assure que le serveur ne va pas recycler des journaux dont `pg_receivewal` n'aurait pas reçu les enregistrements. On retrouve donc le risque d'accumulation des journaux sur le serveur principal si `pg_receivewal` ne fonctionne pas.

Voici l'aide de cet outil en v15 :

```
$ pg_receivewal --help
pg_receivewal reçoit le flux des journaux de transactions PostgreSQL.
```

Usage :
`pg_receivewal [OPTION]...`

Options :
`-D, --directory=RÉPERTOIRE` reçoit les journaux de transactions dans ce réPERTOIRE
`-E, --endpos=LSN` quitte après avoir reçu le LSN spécifié
`--if-not-exists` ne pas renvoyer une erreur si le slot existe

```

-n, --no-loop                               déjà lors de sa création
    --no-sync                                ne boucle pas en cas de perte de la connexion
                                              n'attend pas que les modifications soient
                                              proprement écrites sur disque
-s, --status-interval=SECS                durée entre l'envoi de paquets de statut au
                                              (par défaut 10)
-S, --slot=NOMREP                           slot de réPLICATION à utiliser
    --synchroNous                          vide le journal de transactions immédiatement
                                              après son écriture
-v, --verbose                                affiche des messages verbeux
-V, --version                                 affiche la version puis quitte
-Z, --compress=METHOD[:DETAiL]               compresse comme indiqué
                                              affiche cette aide puis quitte

Options de connexion :
-d, --dbname=CHAÎNE_CONNEX                chaîne de connexion
-h, --host=HÔTE                            hôte du serveur de bases de données ou
                                              répertoire des sockets
-p, --port=PORT                            numéro de port du serveur de bases de données
-U, --username=UTILISATEUR                  se connecte avec cet utilisateur
-w, --no-password                         ne demande jamais le mot de passe
-W, --password                            force la demande du mot de passe (devrait
                                              survenir automatiquement)

Actions optionnelles :
--create-slot                               crée un nouveau slot de réPLICATION
                                              (pour le nom du slot, voir --slot)
--drop-slot                                 supprime un nouveau slot de réPLICATION
                                              (pour le nom du slot, voir --slot)

```

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
Page d'accueil de PostgreSQL : <<https://www.postgresql.org/>>

2.4.7 pg_receivewal - configuration serveur



- `postgresql.conf` (si < v10):

```
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf`:

```
host replication repli_user 192.168.0.0/24 scram-sha-256
```

- Utilisateur de réPLICATION :

```
CREATE ROLE repli_user LOGIN REPLICATION PASSWORD 'supersecret'
```

Le paramètre `max_wal_senders` indique le nombre maximum de connexions de réPLICATION sur le serveur. Logiquement, une valeur de 1 serait suffisante, mais il faut compter sur quelques soucis réseau qui pourraient faire perdre la connexion à `pg_receivewal` sans que le serveur primaire n'en soit mis au courant, et du fait que certains autres outils peuvent utiliser la réPLICATION. `max_replication_slots` indique le nombre maximum de slots de réPLICATION. Pour ces deux paramètres, le défaut est 10 à partir de PostgreSQL 10, mais 0 sur les versions précédentes, et il faudra les modifier.

Si l'on modifie un de ces paramètres, il est nécessaire de redémarrer le serveur PostgreSQL.

Les connexions de réPLICATION nécessitent une configuration particulière au niveau des accès. D'où la modification du fichier `pg_hba.conf`. Le sous-réseau (192.168.0.0/24) est à modifier suivant l'adressage utilisé. Il est d'ailleurs préférable de n'indiquer que le serveur où est installé `pg_receivewal` (plutôt que l'intégralité d'un sous-réseau).

L'utilisation d'un utilisateur de réPLICATION n'est pas obligatoire mais fortement conseillée pour des raisons de sécurité.

2.4.8 pg_receivewal - redémarrage du serveur



- Redémarrage de PostgreSQL
- Slot de réPLICATION

```
SELECT pg_create_physical_replication_slot('archivage');
```

Pour que les modifications soient prises en compte, nous devons redémarrer le serveur.

Enfin, nous devons créer le slot de réPLICATION qui sera utilisé par `pg_receivewal`. La fonction `pg_create_physical_replication_slot()` est là pour ça. Il est à noter que la liste des slots est disponible dans le catalogue système `pg_replication_slots`.

2.4.9 pg_receivewal - lancement de l'outil



- Exemple de lancement

```
pg_receivewal -D /data/archives -S archivage
```

- Journaux créés en temps réel dans le répertoire de stockage
- Mise en place d'un script de démarrage
- S'il n'arrive pas à joindre le serveur primaire
 - Au démarrage de l'outil : pg_receivewal s'arrête
 - En cours d'exécution : pg_receivewal tente de se reconnecter
- Nombreuses options

Une fois le serveur PostgreSQL redémarré, on peut alors lancer pg_receivewal :

```
pg_receivewal -h 192.168.0.1 -U repli_user -D /data/archives -S archivage
```

Les journaux de transactions sont alors créés en temps réel dans le répertoire indiqué (ici, /data/archives) :

```
-rwx----- 1 postgres postgres 16MB juil. 27 000000010000000000000000E*
-rwx----- 1 postgres postgres 16MB juil. 27 000000010000000000000000F*
-rwx----- 1 postgres postgres 16MB juil. 27 00000001000000000000000010.partial*
```

En cas d'incident sur le serveur primaire, il est alors possible de partir d'une sauvegarde physique et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension .partial du dernier journal).

Il faut mettre en place un script de démarrage pour que pg_receivewal soit redémarré en cas de redémarrage du serveur.

2.4.10 Avantages et inconvénients



- Méthode archiver
 - simple à mettre en place
 - perte au maximum d'un journal de transactions
- Méthode pg_receivewal
 - mise en place plus complexe
 - perte minimale voire nulle

La méthode archiver est la méthode la plus simple à mettre en place. Elle se lance au lancement du serveur PostgreSQL, donc il n'est pas nécessaire de créer et installer un script de démarrage. Cependant, un journal de transactions n'est archivé que quand PostgreSQL l'ordonne, soit parce qu'il a rempli le journal en question, soit parce qu'un utilisateur a forcé un changement de journal (avec la fonction `pg_switch_wal` ou suite à un `pg_stop_backup`), soit parce que le délai maximum entre deux archivages a été dépassé (paramètre `archive_timeout`). Il est donc possible de perdre un grand nombre de transactions (même si cela reste bien inférieur à la perte qu'une restauration d'une sauvegarde logique occasionnerait).

La méthode `pg_receivewal` est plus complexe à mettre en place. Il faut exécuter ce démon, généralement sur un autre serveur. Un script de démarrage doit donc être configuré. Par contre, elle a le gros avantage de ne perdre pratiquement aucune transaction, surtout en mode synchrone. Les enregistrements de transactions sont envoyés en temps réel à `pg_receivewal`. Ce dernier les place dans un fichier de suffixe `.partial`, qui est ensuite renommé pour devenir un journal de transactions complet.

2.5 SAUVEGARDE PITR MANUELLE



- 3 étapes :
 - fonction de démarrage
 - copie des fichiers par outil externe
 - fonction d'arrêt
- Exclusive : simple... & obsolète ! (< v15)
- Concurrente : plus complexe à scripter
- Aucun impact pour les utilisateurs ; pas de verrou
- Préférer des outils dédiés qu'un script maison

Une fois l'archivage en place, une sauvegarde à chaud a lieu en trois temps :

- l'appel à la fonction de démarrage ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à la fonction d'arrêt.

La fonction de démarrage s'appelle `pg_backup_start()` à partir de la version 15 mais avait pour nom `pg_start_backup()` auparavant. De la même façon, la fonction d'arrêt s'appelle `pg_backup_stop()` à partir de la version 15, mais `pg_stop_backup()` avant.

La sauvegarde exclusive était la plus simple, et cela en faisait le choix par défaut. Il suffisait d'appeler les fonctions concernées avant et après la copie des fichiers. Il ne pouvait y en avoir qu'une à la fois. Elle ne fonctionnait que depuis un primaire.



À cause de ces limites et de différents problèmes, , la sauvegarde exclusive est déclarée obsolète depuis la 9.6, et n'est plus disponible depuis la version 15. Même sur les versions antérieures, il est conseillé d'utiliser dès maintenant des scripts utilisant les sauvegardes concurrentes.

Tout ce qui suit ne concerne plus que la sauvegarde concurrente.

La sauvegarde concurrente, apparue avec PostgreSQL 9.6, peut être lancée plusieurs fois en parallèle. C'est utile pour créer des secondaires alors qu'une sauvegarde physique tourne, par exemple. Elle est nettement plus complexe à gérer par script. Elle peut être exécutée depuis un serveur secondaire, ce qui allège la charge sur le primaire.

Pendant la sauvegarde, l'utilisateur ne verra aucune différence (à part peut-être la conséquence d'I/O saturées pendant la copie). Aucun verrou n'est posé. Lectures, écritures, suppression et création de tables, archivage de journaux et autres opérations continuent comme si de rien n'était.



La description du mécanisme qui suit est essentiellement destinée à la compréhension et l'expérimentation. En production, un script maison reste une possibilité, mais préférez des outils dédiés et fiables : pg_basebackup, pgBackRest...

Les sauvegardes manuelles servent cependant encore quand on veut utiliser une sauvegarde par snapshot, ou avec rsync (car pg_basebackup ne sait pas synchroniser vers une sauvegarde interrompue ou ancienne), et quand les outils conseillés ne sont pas utilisables ou disponibles sur le système.

2.5.1 Sauvegarde manuelle - 1/3 : pg_backup_start



```
SELECT pg_backup_start (
    - un_label : texte
    - fast : forcer un checkpoint ?
)
```

L'exécution de pg_backup_start() peut se faire depuis n'importe quelle base de données de l'instance.

(Rappelons que pour les versions avant la 15, la fonction s'appelle pg_start_backup(). Pour effectuer une sauvegarde non-exclusive avec ces versions, il faudra positionner un troisième paramètre⁷ à false.)

Le label (le texte en premier argument) n'a aucune importance pour PostgreSQL (il ne sert qu'à l'administrateur, pour reconnaître le backup).

Le deuxième argument est un booléen qui permet de demander un *checkpoint* immédiat, si l'on est pressé et si un pic d'I/O n'est pas gênant. Sinon il faudra attendre souvent plusieurs minutes (selon la configuration du déclenchement du prochain checkpoint, dépendant des paramètres *checkpoint_timeout* et *max_wal_size* et de la rapidité d'écriture imposée par *checkpoint_completion_target*).

La session qui exécute la commande pg_backup_start() doit être la même que celle qui exécutera plus tard pg_backup_stop(). Nous verrons que cette dernière fonction fournira de quoi créer deux fichiers, qui devront être nommés *backup_label* et *tablespace_map*. Si la connexion est interrompue avant pg_backup_stop(), alors la sauvegarde doit être considérée comme invalide.

En plus de rester connectés à la base, les scripts qui gèrent la sauvegarde concurrente doivent donc récupérer et conserver les informations renvoyées par la commande de fin de sauvegarde.

⁷<https://docs.postgresql.fr/14/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-EXCLUSIVE>

La sauvegarde PITR est donc devenue plus complexe au fil des versions, et il est donc recommandé d'utiliser plutôt pg_basebackup ou des outils la supportant (barman, pgBackRest...).

2.5.2 Sauvegarde manuelle - 2/3 : copie des fichiers



- Cas courant : snapshot
 - cohérence ? redondance ?
- Sauvegarde des fichiers **à chaud**
 - répertoire principal des données
 - tablespaces
- Copie forcément incohérente (la restauration des journaux corrigera)
- rsync et autres outils
- Ignorer :
 - postmaster.pid, log, pg_wal, pg_replslot et quelques autres
- Ne pas oublier : configuration !

La deuxième étape correspond à la sauvegarde des fichiers. Le choix de l'outil dépend de l'administrateur. Cela n'a aucune incidence au niveau de PostgreSQL.

La sauvegarde doit comprendre aussi les tablespaces si l'instance en dispose.

Snapshot :

Il est assez fréquent que les sauvegardes se fassent par snapshot au niveau de la baie, de l'hyperviseur, directement ou via divers outils intégrés (Veeam, Tina...), ou encore de l'OS (LVM, ZFS...). Il est crucial que cette sauvegarde soit bien cohérente, y compris entre les tablespaces. Cela est de la seule responsabilité de l'outil utilisé. Il faut en étudier sa documentation pour savoir exactement ce qu'il fait.

Dans les cas simples, la restauration de ce snapshot équivaldra pour PostgreSQL à un redémarrage brutal, mais pour une sauvegarde PITR, il faudra encadrer le snapshot des appels aux fonctions de démarrage et d'arrêt ci-dessus. La facilité d'implémentation dépend de l'outil utilisé. Il faut aussi vérifier qu'il sait gérer les sauvegardes non exclusives pour utiliser PostgreSQL 15 et supérieurs.



Le point noir de la sauvegarde par snapshot est d'être liée au même système matériel que l'instance PostgreSQL (disque, hyperviseur, datacenter...) Une défaillance grave du matériel peut donc emporter, corrompre ou bloquer la sauvegarde en même temps que la sauvegarde. La sécurité de l'instance est donc reportée sur celle de l'infrastructure sous-jacente. Une copie parallèle des données de manière plus classique est conseillée pour éviter un désastre total.

Copie manuelle :



La sauvegarde se fait à chaud : il est donc possible que pendant ce temps des fichiers changent, disparaissent avant d'être copiés ou apparaissent sans être copiés. Cela n'a pas d'importance en soi car les journaux de transactions corrigent cela (leur archivage doit donc commencer **avant** le début de la sauvegarde et se poursuivre sans interruption jusqu'à la fin).

Il **faut** s'assurer que l'outil de sauvegarde supporte cela, c'est-à-dire qu'il soit capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques. `tar` par exemple convient : il retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique. `rsync` est très courant également.

Sur les plateformes Microsoft Windows, peu d'outils sont capables de copier des fichiers en cours de modification. Assurez-vous d'en utiliser un possédant cette fonctionnalité. À noter : l'outil `tar` (ainsi que d'autres issus du projet GNU) est disponible nativement à travers le projet `unxutils`⁸.

Exclusions :

Des fichiers et répertoires sont à ignorer, pour gagner du temps ou faciliter la restauration. Voici la liste exhaustive (disponible aussi dans la documentation officielle⁹) :

- `postmaster.pid`, `postmaster.opts`, `pg_internal.init`;
- les fichiers de données des tables non journalisées (*unlogged*) ;
- `pg_wal`, ainsi que les sous-répertoires (mais à archiver séparément !) ;
- `pg_replslot` : les slots de réPLICATION seront au mieux périmés, au pire gênants sur l'instance restaurée ;
- `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` ne doivent pas être copiés (ils contiennent des informations propres à l'instance, ou qui ne survivent pas à un redémarrage) ;
- les fichiers et répertoires commençant par `pgsql_tmp` (fichiers temporaires) ;
- les fichiers autres que les fichiers et les répertoires standards (donc pas les liens symboliques).

On n'oubliera pas les fichiers de configuration s'ils ne sont pas dans le PGDATA.

⁸<http://unxutils.sourceforge.net/>

⁹<https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP>

2.5.3 Sauvegarde manuelle - 3/3 : pg_backup_stop



Ne pas oublier !!

```
SELECT * FROM pg_backup_stop (
    - true : attente de l'archivage
)
```

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_backup_stop()`.



N'oubliez pas d'exécuter `pg_backup_stop()`, de vérifier qu'il finit avec succès et de récupérer les informations qu'il renvoie !

Cet oubli trop courant rend vos sauvegardes inutilisables !

PostgreSQL va alors :

- marquer cette fin de backup dans le journal des transactions (étape capitale pour la restauration) ;
- forcer la finalisation du journal de transactions courant et donc son archivage, afin que la sauvegarde (fichiers + archives) soit utilisable même en cas de crash juste l'appel à la fonction : `pg_backup_stop()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès.

La fonction renvoie :

- le *lsn* de fin de backup ;
- un champ destiné au fichier `backup_label` ;
- un champ destiné au fichier `tablespace_map`.

```
# SELECT * FROM pg_stop_backup() \gx
```

```
NOTICE: all required WAL segments have been archived
-[ RECORD 1 ]-----
lsn      | 22/2FE5C788
labelfile | START WAL LOCATION: 22/2B000028 (file 00000001000000220000002B)++
           | CHECKPOINT LOCATION: 22/2B000060
           | BACKUP METHOD: streamed
           | BACKUP FROM: master
           | START TIME: 2019-12-16 13:53:41 CET
           | LABEL: rr
           | START TIMELINE: 1
spcmapfile | 134141 /tbl/froid
```

```
| 134152 /tbl/quota
|
```

Ces informations se retrouvent aussi dans un fichier .backup mêlé aux journaux :

```
# cat /var/lib/postgresql/12/main/pg_wal/00000001000000220000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 00000001000000220000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 00000001000000220000002F)
CHECKPOINT LOCATION: 22/2B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2019-12-16 13:53:41 CET
LABEL: rr
START TIMELINE: 1
STOP TIME: 2019-12-16 13:54:04 CET
STOP TIMELINE: 1
```

Il faudra créer le fichier `tablespace_map` avec le contenu du champ `spcmapfile`:

```
134141 /tbl/froid
34152 /tbl/quota
```

... ce qui n'est pas trivial à scripter.

Ces deux fichiers devront être placés dans la sauvegarde, pour être présent d'entrée dans le PGDATA du serveur restauré.

À partir du moment où `pg_backup_stop()` rend la main, il est possible de restaurer la sauvegarde obtenue puis de rejouer les journaux de transactions suivants en cas de besoin, sur un autre serveur ou sur ce même serveur.

Tous les journaux archivés avant celui précisé par le champ `START WAL LOCATION` dans le fichier `backup_label` ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et peuvent donc être supprimés. Attention, il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite.

2.5.4 Sauvegarde de base à chaud : pg_basebackup



Outil de sauvegarde pouvant aussi servir au base backup

- Backup de base ici **sans** les journaux :

```
$ pg_basebackup --format=tar --wal-method=none \
--checkpoint=fast --progress -h 127.0.0.1 -U sauve \
-D /var/lib/postgresql/backups/
```

`pg_basebackup` a été décrit plus haut. Il a l'avantage d'être simple à utiliser, de savoir quels fichiers ne pas copier, de fiabiliser la sauvegarde par un slot de réPLICATION. Il ne réclame en général pas de configuration supplémentaire.

Si l'archivage est déjà en place, copier les journaux est inutile (`--wal-method=none`). Nous verrons plus tard comment lui indiquer où les chercher.

L'inconvénient principal de `pg_basebackup` reste son incapacité à reprendre une sauvegarde interrompue ou à opérer une sauvegarde différentielle ou incrémentale.

2.5.5 Fréquence de la sauvegarde de base



- Dépend des besoins
- De tous les jours à tous les mois
- Plus elles sont espacées, plus la restauration est longue
 - et plus le risque d'un journal corrompu ou absent est important

La fréquence dépend des besoins. Une sauvegarde par jour est le plus commun, mais il est possible d'espacer encore plus la fréquence.

Cependant, il faut conserver à l'esprit que plus la sauvegarde est ancienne, plus la restauration sera longue car un plus grand nombre de journaux seront à rejouer.

2.5.6 Suivi de la sauvegarde de base



- Vue `pg_stat_progress_basebackup`
 - à partir de la v13

La version 13 permet de suivre la progression de la sauvegarde de base, quelque soit l'outil utilisé à condition qu'il passe par le protocole de réPLICATION.

Cela permet ainsi de savoir à quelle phase la sauvegarde se trouve, quelle volumétrie a été envoyée, celle à envoyer, etc.

2.6 RESTAURER UNE SAUVEGARDE PITR



- Une procédure relativement simple
- Mais qui doit être effectuée rigoureusement

La restauration se déroule en trois voire quatre étapes suivant qu'elle est effectuée sur le même serveur ou sur un autre serveur.

2.6.1 Restaurer une sauvegarde PITR (1/5)



- S'il s'agit du même serveur
 - arrêter PostgreSQL
 - supprimer le répertoire des données
 - supprimer les tablespaces

Dans le cas où la restauration a lieu sur le même serveur, quelques étapes préliminaires sont à effectuer.

Il faut arrêter PostgreSQL s'il n'est pas arrêté. Cela arrivera quand la restauration a pour but, par exemple, de récupérer des données qui ont été supprimées par erreur.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer l'ancienne sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant, il faut le supprimer, ainsi que les répertoires des tablespaces au cas où l'instance en possède.

2.6.2 Restaurer une sauvegarde PITR (2/5)



- Restaurer les fichiers de la sauvegarde
- Supprimer les fichiers compris dans le répertoire pg_wal restauré
 - ou mieux, ne pas les avoir inclus dans la sauvegarde initialement
- Restaurer le dernier journal de transactions connu (si disponible)

La sauvegarde des fichiers peut enfin être restaurée. Il faut bien porter attention à ce que les fichiers soient restaurés au même emplacement, tablespaces compris.

Une fois cette étape effectuée, il peut être intéressant de faire un peu de ménage. Par exemple, le fichier `postmaster.pid` peut poser un problème au démarrage. Conserver les journaux applicatifs n'est pas en soi un problème mais peut porter à confusion. Il est donc préférable de les supprimer. Quant aux journaux de transactions compris dans la sauvegarde, bien que ceux en provenance des archives seront utilisés même s'ils sont présents aux deux emplacements, il est préférable de les supprimer. La commande sera similaire à celle-ci :

```
$ rm postmaster.pid log/* pg_wal/[0-9A-F]*
```

Enfin, s'il est possible d'accéder au journal de transactions courant au moment de l'arrêt de l'ancienne instance, il est intéressant de le restaurer dans le répertoire `pg_wal` fraîchement nettoyé. Ce dernier sera pris en compte en toute fin de restauration des journaux depuis les archives et permettra donc de restaurer les toutes dernières transactions validées sur l'ancienne instance, mais pas encore archivées.

2.6.3 Restaurer une sauvegarde PITR (3/5)



- Commande de restauration dans `postgresql.[auto.]conf` (v12+)
 - `restore_command = '... une commande ...'`
 - fichier `recovery.signal`
- Jusque v11, fichier séparé:
 - `recovery.conf` (dans PGDATA)

Jusqu'en version 11 inclusive, la restauration se configure dans un fichier spécifique, appelé `recovery.conf`, impérativement dans le répertoire des données.

À partir de la version 12, on utilise directement `postgresql.conf`, ou un fichier inclus, ou `postgresql.auto.conf`. Par contre, il faut créer un fichier vide `recovery.signal`.

Ce sont ces fichiers `recovery.signal` ou `recovery.conf` qui permettent à PostgreSQL de savoir qu'il doit se lancer dans une restauration, et n'a pas simplement subi un arrêt brutal (auquel cas il ne restaurerait que les journaux en place).

Si vous êtes sur une version antérieure à la version 12, vous pouvez vous inspirer du fichier exemple fourni avec PostgreSQL. Pour une version 10 par exemple, sur Red Hat et dérivés, c'est `/usr/pgsql-10/share/recovery.conf.sample`. Sur Debian et dérivés, c'est `/usr/share/postgresql/10/recovery.conf.sample`. Il contient certains paramètres liés à la mise en place d'un serveur secondaire (*standby*) inutiles ici. Sinon, les paramètres sont dans les différentes parties du `postgresql.conf`.

Le paramètre essentiel est `restore_command`. Il est le pendant des paramètres `archive_command` et `archive_library` pour l'archivage. Cette commande est souvent fournie par l'outil de sauvegarde si vous en utilisez un. Si nous poursuivons notre exemple, ce paramètre pourrait être :

```
restore_command = 'cp /mnt/nfs1/archivage/%f %p'
```

Si le but est de restaurer tous les journaux archivés, il n'est pas nécessaire d'aller plus loin dans la configuration. La restauration se poursuivra jusqu'à l'épuisement de tous les journaux disponibles.

2.6.4 Restaurer une sauvegarde PITR (4/5)



- Jusqu'où restaurer :
 - `recovery_target_name`, `recovery_target_time`
 - `recovery_target_xid`, `recovery_target_lsn`
 - `recovery_target_inclusive`
- Le backup de base doit être antérieur !
- Suivi de timeline :
 - `recovery_target_timeline:latest` ?
- Et on fait quoi ?
 - `recovery_target_action:pause`
 - `pg_wal_replay_resume` pour ouvrir immédiatement
 - ou modifier & redémarrer

Si l'on ne veut pas simplement restaurer tous les journaux, par exemple pour s'arrêter avant une fausse manipulation désastreuse, plusieurs paramètres permettent de préciser le point d'arrêt :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` (le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`) ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;
- jusqu'à un certain identifiant de transaction, grâce au paramètre `recovery_target_xid`, numéro de transaction qu'il est possible de chercher dans les journaux eux-mêmes grâce à l'utilitaire `pg_waldump` ;
- jusqu'à un certain LSN (*Log Sequence Number*¹⁰), grâce au paramètre `recovery_target_lsn`, que là aussi on doit aller chercher dans les journaux eux-mêmes.

¹⁰<https://docs.postgresql.fr/current/datatype-pg-lsn.html>

Avec le paramètre `recovery_target_inclusive` (par défaut à `true`), il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transaction demandé, ou en les excluant.

Dans les cas complexes, nous verrons plus tard que choisir la *timeline* (avec `recovery_target_timeline`, en général à `latest`) peut être utile.



Ces restaurations ponctuelles ne sont possibles que si elles correspondent à un état cohérent d'**après** la fin du *base backup*, soit après le moment du `pg_stop_backup`. Si l'on a un historique de plusieurs sauvegardes, il faudra en choisir une antérieure au point de restauration voulu. Ce n'est pas forcément la dernière. Les outils ne sont pas forcément capables de deviner la bonne sauvegarde à restaurer.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec :

```
recovery_target_action = pause
```

C'est même l'action par défaut si une des options d'arrêt ci-dessus a été choisie : cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Les alternatives sont `promote` et `shutdown`.

Si la cible est atteinte mais que l'on décide de continuer la restauration jusqu'à un autre point (évidemment postérieur), il faut modifier la cible de restauration dans le fichier de configuration, et **redémarrer** PostgreSQL. C'est le seul moyen de rejouer d'autres journaux sans ouvrir l'instance en écriture.

Si l'on est arrivé au point de restauration voulu, un message de ce genre apparaît :

```
LOG: recovery stopping before commit of transaction 8693270, time 2021-09-02
        ↵ 11:46:35.394345+02
LOG: pausing at the end of recovery
HINT: Execute pg_wal_replay_resume() to promote.
```

(Le terme *promote* pour une restauration est un peu abusif.) `pg_wal_replay_resume()` — malgré ce que pourrait laisser croire son nom ! — provoque ici l'arrêt immédiat de la restauration, donc ignore les opérations contenues dans les WALs que l'on n'a pas souhaités restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.



Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` est spécifié mais n'est toujours pas atteint à la fin du rejet des archives, alors le mode `recovery` se termine et le serveur est promu sans erreur, et ce, même si `recovery_target_action` a la valeur `pause` ! (À condition, bien sûr, que le point de cohérence ait tout de même été dépassé.) Il faut donc être vigilant quant aux messages dans le fichier de trace de PostgreSQL !

À partir de PostgreSQL 13, l'instance détecte le problème et s'arrête avec un message FATAL : la restauration ne s'est pas déroulée comme attendu. S'il manque juste certains journaux de transactions, cela permet de relancer PostgreSQL après correction de l'oubli.

La documentation officielle complète sur le sujet est sur le site du projet¹¹.

2.6.5 Restaurer une sauvegarde PITR (5/5)



- Démarrer PostgreSQL
- Rejet des journaux
- Vérifier que le point de cohérence est atteint !

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL. PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions.

Les journaux doivent se dérouler au moins jusqu'à rencontrer le « point de cohérence », c'est-à-dire la mention insérée par `pg_backup_stop()`. Avant cela, il n'est pas possible de savoir si les fichiers issus du `base backup` sont à jour ou pas, et il est impossible de démarrer l'instance avant ce point. Le message apparaît dans les traces et, dans le doute, on doit vérifier sa présence :

```
2020-01-17 16:08:37.285 UTC [15221] LOG: restored log file
    ↵ "000000010000000100000031"...
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file
    ↵ "000000010000000100000032"...
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                read only connections
2020-01-17 16:08:38.009 UTC [15221] LOG: restored log file
    ↵ "000000010000000100000033"...
```

PostgreSQL continue ensuite jusqu'à arriver à la limite fixée, jusqu'à ce qu'il ne trouve plus de journal à rejouer, ou que le bloc de journal lu soit incohérent (ce qui indique qu'on est arrivé à la fin d'un

¹¹<https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET>

journal qui n'a pas été terminé, le journal courant au moment du crash par exemple). Par défaut en v12 il vérifiera qu'il n'existe pas une *timeline* supérieure sur laquelle basculer (par exemple s'il s'agit de la deuxième restauration depuis la sauvegarde du PGDATA). Puis il va s'ouvrir en écriture (sauf si vous avez demandé `recovery_target_action = pause`).

```
2020-01-17 16:08:45.938 UTC [15221] LOG: restored log file "00000001000000010000003C"
                                         from archive
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file
    ↵ "00000001000000010000003D"...
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file
    ↵ "00000001000000010000003E"...
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file
    ↵ "00000001000000010000003F"...
2020-01-17 16:08:47.842 UTC [15221] LOG: invalid record length at 1/3F0000A0:
                                         wanted 24, got 0
2020-01-17 16:08:47.842 UTC [15221] LOG: redo done at 1/3F000028
2020-01-17 16:08:47.842 UTC [15221] LOG: last completed transaction was
                                         at log time 2020-01-17 14:59:30.093491+00
2020-01-17 16:08:47.860 UTC [15221] LOG: restored log file
    ↵ "00000001000000010000003F"...
cp: cannot stat '/opt/pgsql/archives/00000002.history': No such file or directory
2020-01-17 16:08:47.966 UTC [15221] LOG: selected new timeline ID: 2
2020-01-17 16:08:48.179 UTC [15221] LOG: archive recovery complete
cp: cannot stat '/opt/pgsql/archives/00000001.history': No such file or directory
2020-01-17 16:08:51.613 UTC [15217] LOG: database system is ready
                                         to accept connections
```

À partir de la version 12, le fichier `recovery.signal` est effacé.

Avant la v12, si un fichier `recovery.conf` est présent, il sera renommé en `recovery.done`.



Ne jamais supprimer ou renommer manuellement un `recovery.conf` !

Le fichier `backup_label` d'une sauvegarde exclusive est renommé en `backup_label.old`.

2.6.6 Restauration PITR : durée



- Durée dépendante du nombre de journaux
 - rejet séquentiel des WAL
 - Accéléré en version 15 (*prefetch*)

La durée de la restauration est fortement dépendante du nombre de journaux. Ils sont rejoués séquentiellement. Mais avant cela, un fichier journal peut devoir être récupéré, décompressé, et restauré

dans pg_wal.

Il est donc préférable qu'il n'y ait pas trop de journaux à rejouer, et donc qu'il n'y ait pas trop d'espaces entre sauvegardes complètes successives.

La version 15 a optimisé le rejet en permettant l'activation du *prefetch* des blocs de données lors du rejet des journaux.

Un outil comme pgBackRest en mode asynchrone permet de paralléliser la récupération des journaux, ce qui permet de les récupérer via le réseau et de les décompresser par avance pendant que PostgreSQL traite les journaux précédents.

2.6.7 Restauration PITR : différentes timelines



- Fin de *recovery* => changement de *timeline* :
 - l'historique des données prend une autre voie
 - le nom des WAL change
 - fichier .history
- Permet plusieurs restaurations PITR à partir du même *basebackup*
- Choix : *recovery_target_timeline*
 - défaut : *latest* (v12) ou *current* (!) (<v12)

Lorsque le mode *recovery* s'arrête, au point dans le temps demandé ou faute d'archives disponibles, l'instance accepte les écritures. De nouvelles transactions se produisent alors sur les différentes bases de données de l'instance. Dans ce cas, l'historique des données prend un chemin différent par rapport aux archives de journaux de transactions produites avant la restauration. Par exemple, dans ce nouvel historique, il n'y a pas le *DROP TABLE* malencontreux qui a imposé de restaurer les données. Cependant, cette transaction existe bien dans les archives des journaux de transactions.

On a alors plusieurs historiques des transactions, avec des « bifurcations » aux moments où on a réalisé des restaurations. PostgreSQL permet de garder ces historiques grâce à la notion de *timeline*. Une *timeline* est donc l'un de ces historiques, elle se matérialise par un ensemble de journaux de transactions, identifiée par un numéro. Le numéro de la *timeline* est le premier nombre hexadécimal du nom des segments de journaux de transactions (le second est le numéro du journal et le troisième le numéro du segment). Lorsqu'une instance termine une restauration PITR, elle peut archiver immédiatement ces journaux de transactions au même endroit, les fichiers ne seront pas écrasés vu qu'ils seront nommés différemment. Par exemple, après une restauration PITR s'arrêtant à un point situé dans le segment 000000010000000000000000000009 :

```
$ ls -1 /backup/postgresql/archived_wal/
00000001000000010000003C
```

```
00000001000000010000003D
00000001000000010000003E
00000001000000010000003F
000000010000000100000040
00000002.history
00000002000000010000003F
000000020000000100000040
000000020000000100000041
```

À la sortie du mode *recovery*, l'instance doit choisir une nouvelle *timeline*. Les *timelines* connues avec leur point de départ sont suivies grâce aux fichiers *history*, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe *.history*, et archivés avec les fichiers WAL. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers *history* des *timelines* suivantes pour choisir la première disponible, et archive un nouveau fichier *.history* pour la nouvelle *timeline* sélectionnée, avec l'adresse du point de départ dans la *timeline* qu'elle quitte :

```
$ cat 00000002.history
1 0/9765A80 before 2015-10-20 16:59:30.103317+02
```

Après une seconde restauration, ciblant la *timeline* 2, l'instance choisit la *timeline* 3 :

```
$ cat 00000003.history
1 0/9765A80 before 2015-10-20 16:59:30.103317+02
2 0/105AF7D0 before 2015-10-22 10:25:56.614316+02
```

On peut choisir la *timeline* cible en configurant le paramètre *recovery_target_timeline*. À partir de la version 12, *recovery_target_timeline* est par défaut à *latest* et la restauration suit les changements de *timeline* depuis le moment de la sauvegarde.



Cependant, jusqu'en version 11 comprise, la valeur par défaut est *current* et la restauration se fait dans la même *timeline* que le *base backup*. Si entre-temps il y a eu une bascule ou une précédente restauration, la nouvelle *timeline* ne sera pas automatiquement suivie !

Pour choisir une autre *timeline*, il faut donner le numéro de la *timeline* cible comme valeur du paramètre *recovery_target_timeline*. Cela permet d'effectuer plusieurs restaurations successives à partir du même *base backup* et d'archiver au même endroit sans mélanger les journaux.

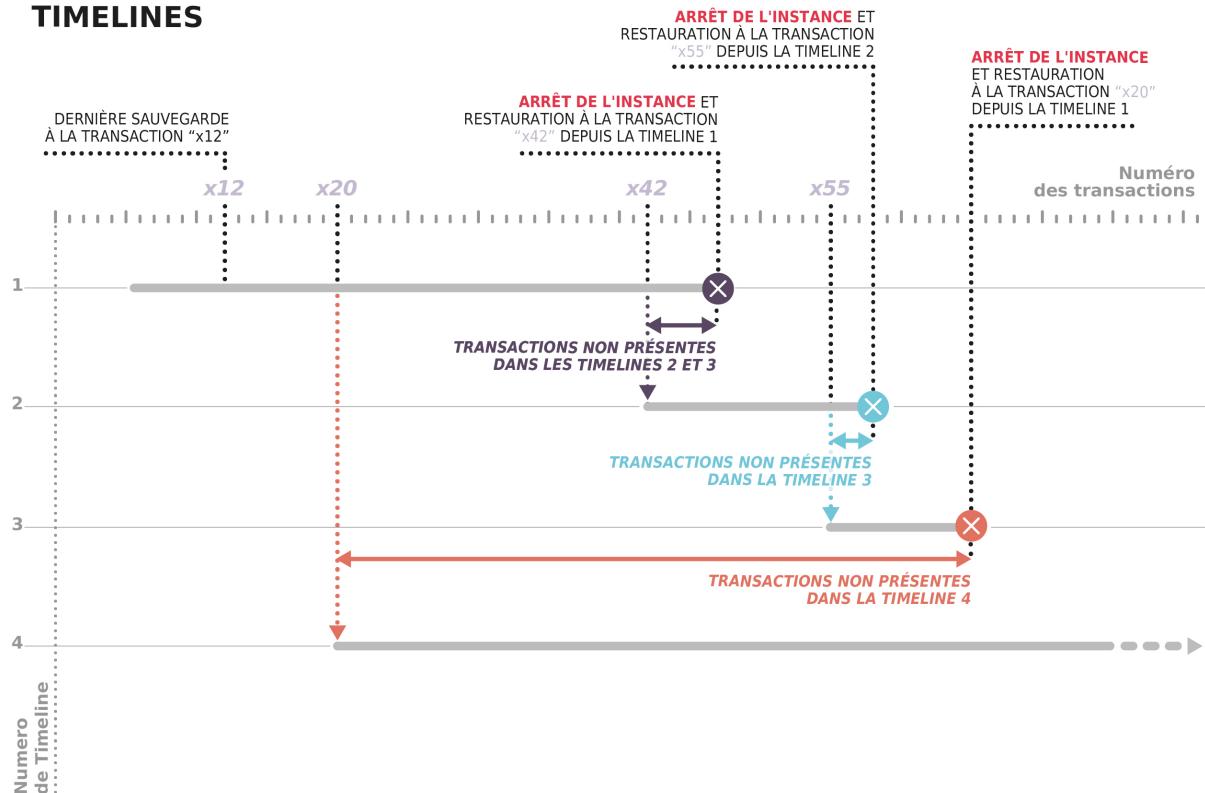


Le numéro de *timeline* dans les traces ou affiché par *pg_controldata* est en décimal. Mais les fichiers *.history* portent un numéro en hexadécimal (par exemple *00000014.history* pour la *timeline* 20). On peut fournir les deux à *recovery_target_timeline* (20 ou '*0x14*'). Attention, il n'y a pas de contrôle !

Attention : pour restaurer dans une *timeline* précise, il faut que le fichier *history* correspondant soit encore présent dans les archives, sous peine d'erreur.

2.6.8 Restauration PITR : illustration des timelines

TIMELINES



Ce schéma illustre ce processus de plusieurs restaurations successives, et la création de différentes *timelines* qui en résulte.

On observe ici les éléments suivants avant la première restauration :

- la fin de la dernière sauvegarde se situe en haut à gauche sur l'axe des transactions, à la transaction $x12$;
- cette sauvegarde a été effectuée alors que l'instance était en activité sur la *timeline* 1.

On décide d'arrêter l'instance alors qu'elle est arrivée à la transaction $x47$, par exemple parce qu'une nouvelle livraison de l'application a introduit un bug qui provoque des pertes de données. L'objectif est de restaurer l'instance avant l'apparition du problème afin de récupérer les données dans un état cohérent, et de relancer la production à partir de cet état. Pour cela, on restaure les fichiers de l'instance à partir de la dernière sauvegarde, puis on modifie le fichier de configuration pour que l'instance, lors de sa phase de *recovery* :

- restaure les WAL archivés jusqu'à l'état de cohérence (transaction $x12$) ;
- restaure les WAL archivés jusqu'au point précédent immédiatement l'apparition du bug appliquatif (transaction $x42$).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 2, la bifurcation s'effectuant à la transaction $x42$. L'instance étant de nouveau ouverte en

écriture, elle va générer de nouveaux WAL, qui seront associés à la nouvelle *timeline* : ils n'écrasent pas les fichiers WAL archivés de la *timeline* 1, ce qui permet de les réutiliser pour une autre restauration en cas de besoin (par exemple si la transaction x42 utilisée comme point d'arrêt était trop loin dans le passé, et que l'on désire restaurer de nouveau jusqu'à un point plus récent).

Un peu plus tard, on a de nouveau besoin d'effectuer une restauration dans le passé - par exemple, une nouvelle livraison applicative a été effectuée, mais le bug rencontré précédemment n'était toujours pas corrigé. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, puis on configure PostgreSQL pour suivre la *timeline* 2 (paramètre `recovery_target_timeline = 2`) jusqu'à la transaction x55. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction x12) ;
- restaurer les WAL archivés jusqu'au point de la bifurcation (transaction x42) ;
- suivre la *timeline* indiquée (2) et rejouer les WAL archivés jusqu'au point précédent immédiatement l'apparition du bug applicatif (transaction x55).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 3, la bifurcation s'effectuant cette fois à la transaction x55.

Enfin, on se rend compte qu'un problème bien plus ancien et subtil a été introduit précédemment aux deux restaurations effectuées. On décide alors de restaurer l'instance jusqu'à un point dans le temps situé bien avant, jusqu'à la transaction x20. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, et on configure le serveur pour restaurer jusqu'à la transaction x20. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction x12) ;
- restaurer les WAL archivés jusqu'au point précédent immédiatement l'apparition du bug applicatif (transaction x20).

Comme la création des deux *timelines* précédentes est archivée dans les fichiers *history*, l'ouverture de l'instance en écriture va basculer sur une nouvelle *timeline* (4). Suite à cette restauration, toutes les modifications de données provoquées par des transactions effectuées sur la *timeline* 1 après la transaction x20, ainsi que celles effectuées sur les *timelines* 2 et 3, ne sont donc pas présentes dans l'instance.

2.6.9 Après la restauration



- Bien vérifier que l'archivage a repris
 - et que les archives des journaux sont complètes
- Ne pas hésiter à reprendre une sauvegarde complète
- Bien vérifier que les secondaires ont suivi

Une fois le nouveau primaire en place, la production peut reprendre, mais il faut vérifier que la sauvegarde PITR est elle aussi fonctionnelle.

Ce nouveau primaire a généralement commencé à archiver ses journaux à partir du dernier journal récupéré de l'ancien primaire, renommé avec l'extension `.partial`, juste avant la bascule sur la nouvelle *timeline*. Il faut bien sûr vérifier que l'archivage des nouveaux journaux fonctionne.

Sur l'ancien primaire, les derniers journaux générés juste avant l'incident n'ont pas forcément été archivés. Ceux-ci possèdent un fichier témoin `.ready` dans `pg_wal/archive_status`. Même s'ils ont été copiés manuellement vers le nouveau primaire avant sa promotion, celui-ci ne les a pas archivés.

Rappelons qu'un « trou » dans le flux des journaux dans le dépôt des archives empêchera la restauration d'aller au-delà de ce point !

Il est possible de forcer l'archivage des fichiers `.ready` depuis l'ancien primaire, avant la bascule, en exécutant à la main les `restore_command` que PostgreSQL aurait générées, mais la facilité pour le faire dépend de l'outil.

La copie de journaux à la main est donc risquée. Il ne faut pas hésiter à reprendre une sauvegarde complète du nouveau primaire pour repartir d'une base sûre.

Quant aux éventuels autres secondaires, il faut bien vérifier que leur configuration a été modifiée et qu'ils suivent. S'ils sont en *log shipping*, la remarque sur l'archivage ci-dessus est encore plus importante.

2.7 POUR ALLER PLUS LOIN



- Gagner en place
 - ...en compressant les journaux de transactions
- Les outils dédiés à la sauvegarde

2.7.1 Compresser les journaux de transactions



- Objectif : éviter de consommer trop de place disque
- Outils de compression standards : gzip, bzip2, lzma
 - attention à ne pas ralentir l'archivage
- wal_compression

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Avec un journal généré toutes les 5 minutes, cela représente 16 Mo (par défaut) toutes les 5 minutes, soit 192 Mo par heure, ou 5 Go par jour. Il n'est pas forcément possible de conserver autant de journaux. Une solution est la compression à la volée et il existe deux types de compression.

La méthode la plus simple et la plus sûre pour les données est une compression non destructive, comme celle proposée par les outils gzip, bzip2, lzma, etc. L'algorithme peut être imposé ou inclus dans l'outil PITR choisi. La compression peut ne pas être très intéressante en terme d'espace disque gagné. Néanmoins, un fichier de 16 Mo aura généralement une taille compressée comprise entre 3 et 6 Mo. Attention par ailleurs au temps de compression des journaux, qui peut entraîner un retard conséquent de l'archivage par rapport à l'écriture des journaux en cas d'écritures lourdes : une compression élevée mais lente peut être dangereuse.

Noter que la compression des journaux à la source existe aussi (paramètre wal_compression, désactivé par défaut), qui s'effectue au niveau de la page, avec un coût en CPU à l'écriture des journaux.

2.7.2 Outils de sauvegarde PITR dédiés



- Se faciliter la vie avec différents outils
 - pgBackRest
 - barman
 - pitrery (< v15, déprécié)
- Fournissent :
 - un outil pour les backups, les purges...
 - une commande pour l'archivage

Il n'est pas conseillé de réinventer la roue et d'écrire soi-même des scripts de sauvegarde, qui doivent prévoir de nombreux cas et bien gérer les erreurs. La sauvegarde concurrente est également difficile à manier. Des outils reconnus existent, dont nous évoquerons brièvement les plus connus. Il en existe d'autres. Ils ne font pas partie du projet PostgreSQL à proprement parler et doivent être installés séparément.

Les outils décrits succinctement plus bas fournissent :

- un outil pour procéder aux sauvegardes, gérer la péremption des archives... ;
- un outil qui sera appelé par `archive_command`.

Leur philosophie peut différer, notamment en terme de centralisation ou de compromis entre simplicité et fonctionnalités. Ces dernières s'enrichissent d'ailleurs au fil du temps.

2.7.3 pgBackRest



- Gère la sauvegarde et la restauration
 - *pull* ou *push*, multidépôts
 - mono ou multi-serveurs
- Indépendant des commandes système
 - utilise un protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multi-thread, sauvegarde depuis un secondaire, archivage asynchrone...
- Projet mature

pgBackRest¹² est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de pg_wal ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser notamment),
- le support intégré de dépôts S3 ou Azure ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

Le projet est récent, très actif, considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest¹³.

2.7.4 barman



- Gère la sauvegarde et la restauration
 - mode *pull*
 - multi-serveurs
- Une seule commande (barman)
- Et de nombreuses actions
 - *list-server, backup, list-backup, recover...*
- Spécificité : gestion de pg_receivewal

barman est un outil créé par 2ndQuadrant (racheté depuis par EDB). Il a pour but de faciliter la mise en place de sauvegardes PITR. Il gère à la fois la sauvegarde et la restauration.

La commande barman dispose de plusieurs actions :

¹²<https://pgbackrest.org/>

¹³https://github.com/pgstef/check_pgbackrest/

- `list-server`, pour connaître la liste des serveurs configurés ;
- `backup`, pour lancer une sauvegarde de base ;
- `list-backup`, pour connaître la liste des sauvegardes de base ;
- `show-backup`, pour afficher des informations sur une sauvegarde ;
- `delete`, pour supprimer une sauvegarde ;
- `recover`, pour restaurer une sauvegarde (la restauration peut se faire à distance).

Contrairement aux autres outils présentés ici, barman permet d'utiliser `pg_receive_wal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman¹⁴

2.7.5 pitrery



- Projet en fin de vie, non compatible v15+
 - Gère la sauvegarde et la restauration
 - mode push
 - mono-serveur
 - Multi-commandes
 - `archive_wal`
 - `pitrery`
 - `restore_wal`

pitrery a été créé par la société Dalibo. Il mettait l'accent sur la simplicité de sauvegarde et la restauration de la base.



Après 10 ans de développement actif, le projet Pitrery est désormais placé en maintenance LTS (*Long Term Support*) jusqu'en novembre 2026. Plus aucune nouvelle fonctionnalité n'y sera ajoutée, les mises à jour concerneront les correctifs de sécurité uniquement. Il est désormais conseillé de lui préférer pgBackRest. Il n'est plus compatible avec PostgreSQL 15 et supérieur.

Site Web de pitrery¹⁵.

¹⁴<https://www.pgbarman.org/>

¹⁵<https://dalibo.github.io/pitrery/>

2.8 CONCLUSION



- Une sauvegarde
 - fiable
 - éprouvée
 - rapide
 - continue
- Mais
 - plus complexe à mettre en place que pg_dump
 - qui restaure toute l'instance

Cette méthode de sauvegarde est la seule utilisable dès que les besoins de performance de sauvegarde et de restauration augmentent (*Recovery Time Objective* ou RTO), ou que le volume de perte de données doit être drastiquement réduit (*Recovery Point Objective* ou RPO).

2.8.1 Questions



N'hésitez pas, c'est le moment !

2.9 QUIZ



https://dali.bo/i2_quiz

2.10 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 (similaire à Red Hat 8), Red Hat/CentOS 7, et Debian/Ubuntu.

Elle ne dure que quelques minutes.



ATTENTION : Red Hat et CentOS 6 et 7, comme Rocky 8, fournissent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est donc obligatoire.

2.10.1 Sur Rocky Linux 8

Installation du dépôt communautaire :

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Les dépôts de la communauté sont sur <https://yum.postgresql.org/>. Les commandes qui suivent peuvent être générées par l'assistant sur <https://www.postgresql.org/download/linux/redhat/>, en précisant :

- la version majeure de PostgreSQL (ici la 15) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Il faut installer le dépôt et désactiver le module PostgreSQL par défaut :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

```
# dnf -qy module disable postgresql
```

Installation de PostgreSQL 15 :

```
# dnf install -y postgresql15-server postgresql15-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Tout à fait optionnellement, une fonctionnalité avancée, le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# dnf install postgresql15-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer PG_SETUP_INITDB_OPTIONS, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-15/bin/postgresql-15-setup initdb
# cat /var/lib/pgsql/15/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé.

Chemins :

Objet	Chemin
Binaires	/usr/pgsql-15/bin
Répertoire de l'utilisateur postgres	/var/lib/pgsql
PGDATA par défaut	/var/lib/pgsql/15/data
Fichiers de configuration	dans PGDATA/
Traces	dans PGDATA/log

Configuration :

Modifier postgresql.conf est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

```
# systemctl start postgresql-15
# systemctl stop postgresql-15
# systemctl reload postgresql-15
```

Test rapide de bon fonctionnement et connexion à psql

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au démarrage du système d'exploitation :

```
# systemctl enable postgresql-15
```

Consultation de l'état de l'instance :

```
# systemctl status postgresql-15
```

Ouverture du *firewall* pour le port 5432 :

Si le *firewall* est actif (dans le doute, consulter `systemctl status firewalld`):

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faudra installer les binaires pour chacune, et l'instance par défaut de chaque version vivra dans un sous-répertoire différent de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf`.

Si plusieurs instances d'une même version majeure (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faudra les installer dans des PGDATA différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un sous-répertoire de `/var/lib/pgsql/15/` (ou l'équivalent pour d'autres versions majeures).
- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-15.service \
     /etc/systemd/system/postgresql-15-secondaire.service
```

- Modification du fichier avec le nouveau chemin :

`Environment=PGDATA=/var/lib/pgsql/15/secondaire`

- Option 1 : création d'une nouvelle instance vierge :

```
# /usr/pgsql-15/bin/postgresql-15-setup initdb postgresql-15-secondaire
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `postgresql.conf` (port !), `recovery.conf`...
- Commandes de maintenance :

```
# systemctl [start|stop|reload|status] postgresql-15-secondaire
# systemctl [enable|disable] postgresql-15-secondaire
```

- Ouvrir un port dans le firewall au besoin.

2.10.2 Sur Red Hat 7 / Cent OS 7

Fondamentalement, le principe reste le même qu'en version 8. Il faudra utiliser `yum` plutôt que `dnf`. Il n'y a pas besoin de désactiver de module AppStream. Le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql15-llvmjit
```

La création de l'instance et la suite sont identiques.

2.10.3 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Installation du dépôt communautaire :

Référence : <https://apt.postgresql.org/>

- Import des certificats et de la clé :

```
# apt install curl ca-certificates gnupg
# curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | gpg --dearmor | \
    sudo tee /etc/apt/trusted.gpg.d/apt.postgresql.org.gpg >/dev/null

- Création du fichier du dépôt /etc/apt/sources.list.d/pgdg.list (ici pour Debian
11 « bullseye » ; adapter au nom de code de la version de Debian ou Ubuntu correspondante :
stretch, bionic, focal...) :
```

```
deb http://apt.postgresql.org/pub/repos/apt bullseye-pgdg main
```

Installation de PostgreSQL 15 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

```
# apt update
# apt install postgresql-common
```

Dans /etc/postgresql-common/createcluster.conf, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-15 postgresql-client-15
```

(Pour les versions 9.x, installer aussi le paquet postgresql-contrib-9.x).

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	/usr/lib/postgresql/15/bin/
Répertoire de l'utilisateur postgres	/var/lib/postgresql
PGDATA de l'instance par défaut	/var/lib/postgresql/15/main
Fichiers de configuration	dans /etc/postgresql/15/main/
Traces	dans /var/log/postgresql/

Configuration

Modifier postgresql.conf est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils :

```
# pg_ctlcluster 15 main [start|stop|reload|status]
```

Démarrage de l'instance au lancement :

C'est en place par défaut, et modifiable dans /etc/postgresql/15/main/start.conf.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 15 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- les paramètres de création d'instance dans /etc/postgresql-common/createcluster.conf peuvent être modifiés, par exemple ici pour : les *checksums*, les messages en anglais, l'authentification sécurisée, le format des traces et un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C --auth-host=scram-sha-256
                 --auth-local=peer'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- création de l'instance, avec possibilité là aussi de préciser certains paramètres du postgresql.conf voire de modifier les chemins des fichiers (déconseillé si vous pouvez l'éviter) :

```
# pg_createcluster 15 secondaire \
--port=5433 \
--datadir=/PGDATA/11/basedecisionnelle \
--pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
-- --data-checksums --waldir=/ssd/postgresql/11/basedecisionnelle/journaux

- démarrage:
```

```
# pg_ctlcluster 15 secondaire start
```

2.10.4 Accès à l'instance sur le serveur même

Par défaut, l'instance n'est accessible que par l'utilisateur système postgres, qui n'a pas de mot de passe. Un détour par sudo est nécessaire :

```
$ sudo -iu postgres psql
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à trust le type de la connexion en local dans le pg_hba.conf :

```
local    all            postgres          trust
```

La connexion en tant qu'utilisateur postgres (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans pg_hba.conf, mise en place d'une authentification par mot de passe (md5 par défaut) pour les accès à localhost :

```
# IPv4 local connections:
host    all            all            127.0.0.1/32          md5
# IPv6 local connections:
host    all            all            ::1/128              md5
```

(une authentification scram-sha-256 est plus conseillée mais elle impose que password_encryption soit à cette valeur dans postgresql.conf avant de définir les mots de passe).

- ajout d'un mot de passe à l'utilisateur postgres de l'instance ;

```
dalibo:~$ sudo -iu postgres psql
psql (15.1)
Saisissez « help » pour l'aide.
postgres=# \password
Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
postgres=# \q

dalibo:~$ psql -h localhost -U postgres
Mot de passe pour l'utilisateur postgres :
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#

```

- pour se connecter sans taper le mot de passe, un fichier .pgpass dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

- ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- pour n'avoir à taper que psql, on peut définir ces variables d'environnement dans la session voire dans ~/.bashrc :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans /var/lib/pgsql/15/data/log ou /var/log/postgresql/);
- toute modification de pg_hba.conf implique de recharger la configuration par une de ces trois méthodes selon le système :

```
root:~# systemctl reload postgresql-15
root:~# pg_ctlcluster 15 main reload
postgres:~$ psql -c 'SELECT pg_reload_conf();'
```

2.11 INTRODUCTION À PG BENCH

pgbench est un outil de test livré avec PostgreSQL. Son but est de faciliter la mise en place de benchmarks simples et rapides. Par défaut, il installe une base assez simple, génère une activité plus ou moins intense et calcule le nombre de transactions par seconde et la latence. C'est ce qui sera fait ici dans cette introduction. On peut aussi lui fournir ses propres scripts.

La documentation complète est sur <https://docs.postgresql.fr/current/pgbench.html>. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017¹⁶.

2.11.1 Installation

L'outil est installé avec les paquets habituels de PostgreSQL, client ou serveur suivant la distribution.

Dans le cas des paquets RPM du PGDG, l'outil n'est pas dans le PATH par défaut ; il faudra donc fournir le chemin complet :

```
/usr/pgsql-15/bin/pgbench
```

Il est préférable de créer un rôle non privilégié dédié, qui possédera la base de données :

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebiencomplexé';
CREATE DATABASE pgbench OWNER pgbench ;
```

Le pg_hba.conf doit éventuellement être adapté.

La base par défaut s'installe ainsi (indiquer la base de données en dernier ; ajouter -p et -h au besoin) :

```
pgbench -U pgbench --initialize --scale=100 pgbench
```

--scale permet de faire varier proportionnellement la taille de la base. À 100, la base pèsera 1,5 Go, avec 10 millions de lignes dans la table principale pgbench_accounts :

```
pgbench@pgbench=# \d+
          Liste des relations
  Schéma |      Nom       | Type | Propriétaire | Taille | Description
-----+-----+-----+-----+-----+-----+
public | pg_buffercache | vue  | postgres    | 0 bytes |
public | pgbench_accounts | table | pgbench     | 1281 MB |
public | pgbench_branches | table | pgbench     | 40 kB   |
public | pgbench_history | table | pgbench     | 0 bytes |
public | pgbench_tellers | table | pgbench     | 80 kB   |
```

2.11.2 Générer de l'activité

Pour simuler une activité de 20 clients simultanés, répartis sur 4 processeurs, pendant 100 secondes :

¹⁶https://youtu.be/aTwh_CgRaE0

```
pgbench -U pgbench -c 20 -j 4 -T100 pgbench
```

NB : ne **pas** utiliser **-d** pour indiquer la base, qui signifie **--debug** pour pgbench, qui noiera alors l'affichage avec ses requêtes :

```
UPDATE pgbench_accounts SET abalance = abalance + -3455 WHERE aid = 3789437;
SELECT abalance FROM pgbench_accounts WHERE aid = 3789437;
UPDATE pgbench_tellers SET tbalance = tbalance + -3455 WHERE tid = 134;
UPDATE pgbench_branches SET bbalance = bbalance + -3455 WHERE bid = 78;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (134, 78, 3789437, -3455, CURRENT_TIMESTAMP);
```

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (*latency*) :

```
scaling factor: 100
query mode: simple
number of clients: 20
number of threads: 4
duration: 10 s
number of transactions actually processed: 20433
latency average = 9.826 ms
tps = 2035.338395 (including connections establishing)
tps = 2037.198912 (excluding connections establishing)
```

Modifier le paramétrage est facile grâce à la variable d'environnement PGOPTIONS :

```
PGOPTIONS=' -c synchronous_commit=off -c commit_siblings=20' \
pgbench -d pgbench -U pgbench -c 20 -j 4 -T100 2>/dev/null

latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```



Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

2.12 TRAVAUX PRATIQUES

2.12.1 pg_basebackup : sauvegarde ponctuelle & restauration



But : Créer une sauvegarde physique à chaud à un moment précis de la base avec pg_basebackup, et la restaurer.

Configurer la réPLICATION dans postgresql.conf :

- désactiver l'archivage s'il est actif
- autoriser des connexions de réPLICATION en streaming en local.

Pour insérer des données :

- générer de l'activité avec pgbench en tant qu'utilisateur **postgres** :

```
$ createdb bench  
$ /usr/pgsql-15/bin/pgbench -i -s 100 bench  
$ /usr/pgsql-15/bin/pgbench bench -n -P 5 -T 720
```

- laisser tourner en arrière-plan
- surveiller l'évolution de l'activité sur la table pgbench_history, par exemple ainsi :

```
$ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
```

En parallèle, sauvegarder l'instance avec :

- pg_basebackup au format tar, compressé avec gzip ;
- sans oublier les journaux ;
- avec l'option --max-rate=16M pour ralentir la sauvegarde ;
- le répertoire de sauvegarde sera /var/lib/pgsql/15/backups/basebackup ;
- surveillez la progression dans une autre session avec la vue système adéquate.

Une fois la sauvegarde terminée :

- regarder les fichiers générés ;
- arrêter la session pgbench ; Afficher la date de dernière modification dans pgbench_history.

- Arrêter l'instance.
- Faire une copie à froid des données (par exemple avec cp -rfp) vers /var/lib/pgsql/15/data.old

| (cette copie resservira plus tard).

- | – Vider le répertoire des données.
- | – Restaurer la sauvegarde pg_basebackup en décompressant ses deux archives.
- | – Redémarrer l'instance.

| Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

| Quelle est la dernière donnée restaurée ?

| Tenter une nouvelle restauration depuis l'archive pg_basebackup *sans* restaurer les journaux de transaction. Que se passe-t-il ?

2.12.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants



| But : Coupler une sauvegarde à chaud avec pg_basebackup et l'archivage

| Remettre en place la copie de l'instance prise précédemment. Configurer l'archivage vers un répertoire /var/lib/pgsql/15/archives, par exemple avec rsync. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

| Générer à nouveau de l'activité avec pgbench. Vérifier que l'archivage fonctionne.

| En parallèle, lancer une nouvelle sauvegarde avec pg_basebackup au format plain.

| Utiliser pg_verify_backup pour contrôler l'intégrité de la sauvegarde.

| À quoi correspond le fichier .backup dans les archives ?

| Noter la valeur des dernières données insérées.

| Effacer le PGDATA. Restaurer la sauvegarde *sans* les journaux. Configurer la restore_command. Créer le fichier recovery.signal. Démarrer PostgreSQL.

| Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

| Vérifier quelles données ont été restaurées.

2.13 TRAVAUX PRATIQUES (SOLUTIONS)

2.13.1 pg_basebackup : sauvegarde ponctuelle & restauration

Configurer la réPLICATION dans `postgresql.conf`:

- désactiver l'archivage s'il est actif
- autoriser des connexions de réPLICATION en streaming en local.

On n'aura ici pas besoin de l'archivage. S'il est déjà actif, on peut se contenter d'inhiber ainsi la commande d'archivage :

```
archive_mode = on
archive_command = '/bin/true'
```

(Cela permet d'épargner le redémarrage à chaque modification de `archive_mode`.)

Vérifier la configuration de l'autorisation de connexion en réPLICATION dans `pg_hba.conf`. Si besoin, mettre à jour la ligne en fin de fichier :

```
local    replication    all                                peer
```

Cela va ouvrir l'accès sans mot de passe depuis l'utilisateur système **postgres**.

Redémarrer PostgreSQL :

```
# systemctl restart postgresql-15
```

Pour insérer des données :

- générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :

```
$ createdb bench
$ /usr/pgsql-15/bin/pgbench -i -s 100 bench
$ /usr/pgsql-15/bin/pgbench bench -n -P 5 -T 720
```

- laisser tourner en arrière-plan
- surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :

```
$ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
```

En parallèle, sauvegarder l'instance avec :

- `pg_basebackup` au format tar, compressé avec gzip ;
- sans oublier les journaux ;
- avec l'option `--max-rate=16M` pour ralentir la sauvegarde ;
- le répertoire de sauvegarde sera `/var/lib/pgsql/15/backups/basebackup` ;
- surveillez la progression dans une autre session avec la vue système adéquate.

En tant que **postgres** :

```
$ pg_basebackup -D /var/lib/pgsql/15/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M

1583675/1583675 kB (100%), 1/1 tablespace
```

La progression peut se suivre depuis psql avec :

```
$ \x on
$ SELECT * FROM pg_stat_progress_basebackup ;
$ \watch

Thu Jan  5 16:58:05 2023 (every 2s)

-[ RECORD 1 ]-----+-----
pid                | 19763
phase              | waiting for checkpoint to finish
backup_total        |
backup_streamed    | 0
tablespaces_total   | 0
                           ↵
tablespaces_streamed | 0
                           ↵
```

Thu Jan 5 16:58:07 2023 (every 2s)

```
-[ RECORD 1 ]-----+-----
pid                | 19763
phase              | streaming database files
backup_total        | 1611215360
backup_streamed    | 29354496
tablespaces_total   | 1
tablespaces_streamed | 0
...
```

Évidemment, en production, il ne faut pas sauvegarder en local.

Une fois la sauvegarde terminée :

- regarder les fichiers générés ;
- arrêter la session pgbench ; Afficher la date de dernière modification dans pg-bench_history.

```
$ ls -lha /var/lib/pgsql/15/backups/basebackup
...
-rw-----. 1 postgres postgres 180K Jan  5 17:00 backup_manifest
-rw-----. 1 postgres postgres  91M Jan  5 17:00 base.tar.gz
-rw-----. 1 postgres postgres  23M Jan  5 17:00 pg_wal.tar.gz
```

On obtient donc :

- une archive de la sauvegarde « de base » ;
- une archive des journaux nécessaires ;
- un fichier manifeste au format texte contenant les sommes de contrôles des fichiers archivés.

pgbench s'arrête avec un simple **Ctrl-C**. L'heure de dernière modification est :

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
max
-----
2023-01-05 17:01:51.595414
```

- Arrêter l'instance.
- Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/15/data.old` (cette copie resservira plus tard).

```
# systemctl stop postgresql-15
$ cp -rfp /var/lib/pgsql/15/data /var/lib/pgsql/15/data.old
```

- Vider le répertoire des données.
- Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
- Redémarrer l'instance.

On restaure dans le répertoire de données l'archive de base, puis les journaux dans leur sous-répertoire. La suppression des journaux est optionnelle, mais elle nous permettra de ne pas mélanger les traces d'avant et d'après la restauration.

```
$ rm -rf /var/lib/pgsql/15/data/*
$ tar -C /var/lib/pgsql/15/data \
      -xzf /var/lib/pgsql/15/backups/basebackup/base.tar.gz
$ tar -C /var/lib/pgsql/15/data/pg_wal \
      -xzf /var/lib/pgsql/15/backups/basebackup/pg_wal.tar.gz
$ rm -rf /var/lib/pgsql/15/data/log/*
# systemctl start postgresql-15
```

Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

```
$ tail -F /var/lib/pgsql/15/data/log/postgresql-*.log
...
2023-01-05 17:10:51.412 UTC [20057] LOG:  database system was interrupted; last
  ↵ known up at 2023-01-05 16:59:03 UTC
2023-01-05 17:10:51.510 UTC [20057] LOG:  redo starts at 0/830000B0
2023-01-05 17:10:52.105 UTC [20057] LOG:  consistent recovery state reached at
  ↵ 0/8E8450F0
2023-01-05 17:10:52.105 UTC [20057] LOG:  redo done at 0/8E8450F0 system usage: CPU:
  ↵  user: 0.28 s, system: 0.24 s, elapsed: 0.59 s
2023-01-05 17:10:52.181 UTC [20055] LOG:  checkpoint starting: end-of-recovery
  ↵ immediate wait
2023-01-05 17:10:53.446 UTC [20055] LOG:  checkpoint complete: wrote 16008 buffers
  ↵ (97.7%); ...
2023-01-05 17:10:53.466 UTC [20051] LOG:  database system is ready to accept
  ↵ connections
```

PostgreSQL considère qu'il a été interrompu brutalement et part en *recovery*. Noter en particulier la mention *consistent recovery state reached* : la sauvegarde est bien cohérente.

Quelle est la dernière donnée restaurée ?

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'

max
-----
2023-01-05 17:00:40.936925
```

Grâce aux journaux (pg_wal) restaurés, l'ensemble des modifications survenues **pendant** la sauvegarde ont bien été récupérées. Par contre, les données générées après la sauvegarde n'ont, elles, pas été récupérées.

Tenter une nouvelle restauration depuis l'archive pg_basebackup *sans* restaurer les journaux de transaction. Que se passe-t-il ?

```
# systemctl stop postgresql-15
# rm -rf /var/lib/pgsql/15/data/*
# tar -C /var/lib/pgsql/15/data \
      -xzf /var/lib/pgsql/15/backups/basebackup/base.tar.gz
# rm -rf /var/lib/pgsql/15/data/log/*
# systemctl start postgresql-15
```

Résultat :

```
Job for postgresql-15.service failed because the control process exited with error
  code.
See "systemctl status postgresql-15.service" and "journalctl -xe" for details.
```

Pour trouver la cause :

```
# tail -F /var/lib/pgsql/15/data/log/postgresql-*.log
...
2023-01-05 17:16:52.048 UTC [20177] LOG: database system was interrupted; last
  known up at 2023-01-05 16:59:03 UTC
2023-01-05 17:16:52.134 UTC [20177] LOG: invalid checkpoint record
2023-01-05 17:16:52.134 UTC [20177] FATAL: could not locate required checkpoint
  record
2023-01-05 17:16:52.134 UTC [20177] HINT: If you are restoring from a backup, touch
  "/var/lib/pgsql/15/data/recovery.signal" and add required recovery options.
    If you are not restoring from a backup, try removing the file
      "/var/lib/pgsql/15/data/backup_label".
    Be careful: removing "/var/lib/pgsql/15/data/backup_label" will result in a
      corrupt cluster if restoring from a backup.
```

PostgreSQL ne trouve pas les journaux nécessaires à sa restauration à un état cohérent, le service refuse de démarrer. Il a trouvé un checkpoint dans le fichier backup_label créé au début de la sauvegarde, mais aucun checkpoint postérieur dans les journaux (et pour cause).

Les traces contiennent ensuite des suggestions qui peuvent être utiles.

Cependant, un fichier recovery.signal ne sert à rien sans recovery_command, et nous n'en avons pas encore paramétré ici.



Quant au fichier `backup_label`, le supprimer permettrait peut-être de démarrer l'instance mais celle-ci serait alors dans un état incohérent !

Il y a plus de chance que cela finisse ainsi :

```
2023-01-05 17:33:53.046 UTC [20512] PANIC: could not locate a valid checkpoint
    ↵ record
```

En résumé : la restauration des journaux n'est pas optionnelle !

2.13.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

Remettre en place la copie de l'instance prise précédemment. Configurer l'archivage vers un répertoire `/var/lib/pgsql/15/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

```
# systemctl stop postgresql-15
# rm -rf /var/lib/pgsql/15/data
# cp -rfp /var/lib/pgsql/15/data.old /var/lib/pgsql/15/data
```

Créer le répertoire d'archivage s'il n'existe pas déjà :

```
$ mkdir /var/lib/pgsql/15/archives
```

Là encore, en production, ce sera un partage distant. L'utilisateur système **postgres** doit avoir le droit d'y écrire.

L'archivage se définit dans `postgresql.conf` :

```
archive_mode = on
archive_command = 'rsync %p /var/lib/pgsql/15/archives/%f'
```

et on peut y définir aussi tout de suite la commande de restauration :

```
restore_command = 'rsync /var/lib/pgsql/15/archives/%f %p'
# systemctl start postgresql-15
```

Générer à nouveau de l'activité avec `pgbench`. Vérifier que l'archivage fonctionne.

```
$ /usr/pgsql-15/bin/pgbench bench -n -P 5 -T 720
$ ls -lha /var/lib/pgsql/15/archives
...
-rw-----. 1 postgres postgres 16M Jan  5 18:32 000000010000000000000000BB
-rw-----. 1 postgres postgres 16M Jan  5 18:32 000000010000000000000000BC
-rw-----. 1 postgres postgres 16M Jan  5 18:32 000000010000000000000000BD
...
```

En parallèle, lancer une nouvelle sauvegarde avec pg_basebackup au format plain.

```
$ rm -rf /var/lib/pgsql/15/backups/basebackup
$ pg_basebackup -D /var/lib/pgsql/15/backups/basebackup -Fp \
--checkpoint=fast --progress --max-rate=16M

1586078/1586078 kB (100%), 1/1 tablespace
```

Le répertoire cible devra avoir été vidé.

La taille de la sauvegarde sera bien sûr nettement plus grosse qu'en tar compressé.

Utiliser pg_verifybackup pour contrôler l'intégrité de la sauvegarde.

Si tout va bien, le message sera lapidaire :

```
$ /usr/pgsql-15/bin/pg_verifybackup /var/lib/pgsql/15/backups/basebackup
backup successfully verified
```

S'il y a un problème, des messages de ce genre apparaîtront :

```
pg_verifybackup: error: "global/TEST" is present on disk but not in the manifest
pg_verifybackup: error: "global/2671" is present in the manifest but not on disk
pg_verifybackup: error: "postgresql.conf" has size 29507 on disk but size 29506 in
  ↵ the manifest
```

À quoi correspond le fichier .backup dans les archives ?

En effet, ce fichier apparaît parmi les journaux archivés :

```
$ ls -lha /var/lib/pgsql/15/archives
...
-rw----- 1 postgres postgres 16M Jan  5 18:33 000000010000000000000000BE
-rw----- 1 postgres postgres 341 Jan  5 18:34
  ↵ 000000010000000000000000BE.00003E00.backup
-rw----- 1 postgres postgres 16M Jan  5 18:33 000000010000000000000000BF
...
```

Son contenu correspond au futur backup_label :

```
START WAL LOCATION: 0/BE003E00 (file 000000010000000000000000BE)
STOP WAL LOCATION: 0/C864D0F8 (file 000000010000000000000000C8)
CHECKPOINT LOCATION: 0/BE0AB340
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-01-05 18:32:52 UTC
LABEL: pg_basebackup base backup
START TIMELINE: 1
STOP TIME: 2023-01-05 18:34:29 UTC
STOP TIMELINE: 1
```

Noter la valeur des dernières données insérées.

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'  
max  
-----  
2023-01-05 18:41:23.068948
```

Effacer le PGDATA. Restaurer la sauvegarde *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

```
# systemctl stop postgresql-15  
# rm -rf /var/lib/pgsql/15/data/*  
# tar -C /var/lib/pgsql/15/data \  
-xzf /var/lib/pgsql/15/backups/basebackup/base.tar.gz  
# rm -rf /var/lib/pgsql/15/data/log/*
```

Créer le fichier `recovery.signal`:

```
$ touch /var/lib/pgsql/15/data/recovery.signal
```

Démarrer le service :

```
# systemctl start postgresql-15
```

Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

Les traces sont plus complexes à cause de la restauration depuis les archives :

```
# tail -F /var/lib/pgsql/15/data/log/postgresql-*.log  
...  
2023-01-05 18:43:24.572 UTC [22535] LOG: database system was interrupted; last  
↳ known up at 2023-01-05 18:32:52 UTC  
rsync: link_stat "/var/lib/pgsql/15/archives/00000002.history" failed: No such file  
↳ or directory (2)  
rsync error: some files/attrs were not transferred (see previous errors) (code 23)  
↳ at main.c(1189) [sender=3.1.3]  
2023-01-05 18:43:24.657 UTC [22535] LOG: starting archive recovery  
2023-01-05 18:43:24.728 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000BE" from archive  
2023-01-05 18:43:24.797 UTC [22535] LOG: redo starts at 0/BE003E00  
2023-01-05 18:43:24.960 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000BF" from archive  
2023-01-05 18:43:25.167 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000C0" from archive  
2023-01-05 18:43:25.408 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000C1" from archive  
...  
2023-01-05 18:43:27.036 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000C8" from archive  
2023-01-05 18:43:27.240 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000C9" from archive  
2023-01-05 18:43:27.331 UTC [22535] LOG: consistent recovery state reached at  
↳ 0/C864D0F8  
2023-01-05 18:43:27.331 UTC [22530] LOG: database system is ready to accept  
↳ read-only connections  
2023-01-05 18:43:27.551 UTC [22535] LOG: restored log file  
↳ "00000001000000000000000CA" from archive
```

```
2023-01-05 18:43:27.799 UTC [22535] LOG: restored log file
    ↵ "0000000100000000000000CB" from archive
...
2023-01-05 18:43:34.510 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000E0" from archive
2023-01-05 18:43:34.792 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000E1" from archive
2023-01-05 18:43:35.047 UTC [22535] LOG: redo in progress, elapsed time: 10.25 s,
    ↵ current LSN: 0/E0FF3438
2023-01-05 18:43:35.254 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000E2" from archive
2023-01-05 18:43:36.016 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000E3" from archive
...
2023-01-05 18:43:40.052 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000EF" from archive
2023-01-05 18:43:40.376 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000F0" from archive
rsync: link_stat "/var/lib/pgsql/15/archives/00000001000000000000F1" failed: No
    ↵ such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
    ↵ at main.c(1189) ...
rsync: link_stat "/var/lib/pgsql/15/archives/00000001000000000000F1" failed: No
    ↵ such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
    ↵ at main.c(1189) ...
2023-01-05 18:43:40.576 UTC [22535] LOG: redo done at 0/F0A6C9E0 system usage:
    CPU: user: 2.51 s, system: 2.28 s, elapsed:
        ↵ 15.77 s
2023-01-05 18:43:40.577 UTC [22535] LOG: last completed transaction
    was at log time 2023-01-05 18:41:23.077219+00
2023-01-05 18:43:40.638 UTC [22535] LOG: restored log file
    ↵ "00000001000000000000F0" from archive
rsync: link_stat "/var/lib/pgsql/15/archives/00000002.history" failed: No such file
    ↵ or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
    ↵ at main.c(1189) ...
2023-01-05 18:43:40.743 UTC [22535] LOG: selected new timeline ID: 2
rsync: link_stat "/var/lib/pgsql/15/archives/00000001.history" failed: No such file
    ↵ or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
    ↵ at main.c(1189) ...
2023-01-05 18:43:40.875 UTC [22535] LOG: archive recovery complete
2023-01-05 18:43:40.883 UTC [22533] LOG: checkpoint starting: end-of-recovery
    ↵ immediate wait
2023-01-05 18:43:43.155 UTC [22533] LOG: checkpoint complete: wrote 16012 buffers
    ↵ (97.7%); ...
2023-01-05 18:43:43.195 UTC [22530] LOG: database system is ready to accept
    ↵ connections
```

Les messages d'erreur de rsync ne sont pas inquiétants : celui-ci ne trouve simplement pas les fichiers demandés à la restore_command. PostgreSQL sait ainsi qu'il n'y a pas de fichier 00000002.history et donc pas de timeline de ce numéro. Il devine aussi qu'il a restauré tous les journaux quand la récupération de l'un d'entre eux échoue.

La progression de la restauration peut être suivie grâce aux différents messages ci-dessous, de démarrage, d'atteinte du point de cohérence, de statut... jusqu'à l'heure exacte de restauration. Enfin, il y a bascule sur une nouvelle *timeline*, et un checkpoint.

```
LOG: starting archive recovery
LOG: redo starts at 0/BE003E00
LOG: consistent recovery state reached at 0/C864D0F8
LOG: redo in progress, elapsed time: 10.25 s, current LSN: 0/E0FF3438
LOG: redo done at 0/F0A6C9E0 ...
LOG: last completed transaction was at log time 2023-01-05 18:41:23.077219+00
LOG: selected new timeline ID: 2
LOG: archive recovery complete
LOG: checkpoint complete:
```

Noter que les journaux portent la nouvelle *timeline* :

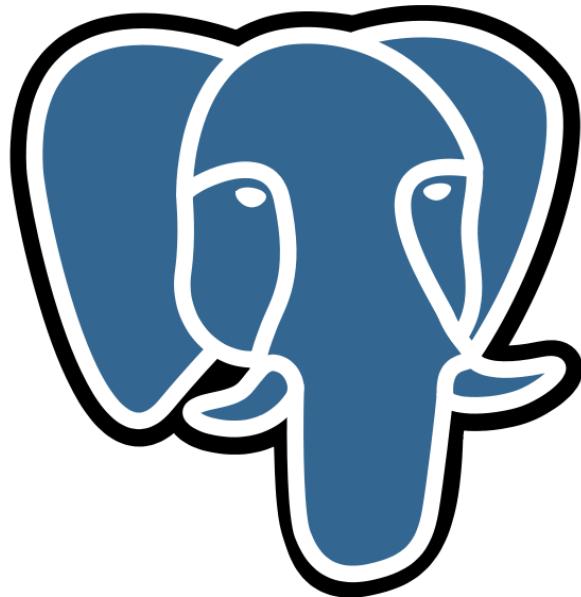
```
$ ls -l /var/lib/pgsql/15/data/pg_wal/
...
-rw-----. 1 postgres postgres 16777216 Jan  5 18:43 000000020000000100000023
-rw-----. 1 postgres postgres 16777216 Jan  5 18:43 000000020000000100000024
-rw-----. 1 postgres postgres      42 Jan  5 18:43 00000002.history
drwx-----. 2 postgres postgres     35 Jan  5 18:43 archive_status
```

Vérifier quelles données ont été restaurées.

Cette fois, toutes les données générées après la sauvegarde ont bien été récupérées :

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history; '
max
-----
2023-01-05 18:41:23.068948
```

3/ PostgreSQL : Outils de sauvegarde physique



3.1 INTRODUCTION



- 2 mécanismes de sauvegarde natifs et robustes
- Industrialisation fastidieuse
- Des outils existent !!

Nous avons vu le fonctionnement interne du mécanisme de sauvegarde physique. Celui-ci étant en place nativement dans le moteur PostgreSQL depuis de nombreuses versions, sa robustesse n'est plus à prouver. Cependant, son industrialisation reste fastidieuse.

Des outils tiers existent et vont permettre de faciliter la gestion des sauvegardes, de leur mise en place jusqu'à la restauration. Dans ce module nous allons voir en détail certains de ces outils et étudier les critères qui vont nous permettre de choisir la meilleure solution selon notre contexte.

3.1.1 Au menu



- Présentation:
 - pg_basebackup
 - pgBackRest
 - Barman
- Comment choisir ?

Lors de cette présentation, nous allons passer en revue les différents outils principaux de gestion de sauvegardes, leurs forces, le paramétrage, l'installation et l'exploitation.

3.1.2 Définition du besoin - Contexte



- Sauvegarde locale (ex. NFS) ?
- Copie vers un serveur tiers (push) ?
- Sauvegarde distante initiée depuis un serveur tiers (pull) ?
- Ressources à disposition ?
- Accès SSH ?
- OS ?
- Sauvegardes physiques ? Logiques ?
- Version de PostgreSQL ?
- Politique de rétention ?

Où les sauvegardes doivent-elles être stockées ?

Quelles ressources sont à disposition : serveur de sauvegarde dédié ? quelle puissance pour la compression ?

De quel type d'accès aux serveurs de base de données dispose-t-on ? Quelle est la version du système d'exploitation ?

Il est très important de se poser toutes ces questions, les réponses vont servir à établir le contexte et permettre de choisir l'outil et la méthode la plus appropriée.



Attention, pour des raisons de sécurité et de fiabilité, les répertoires choisis pour la restauration des données de votre instance **ne doivent pas** être à la racine d'un point de montage.

Si un ou plusieurs points de montage sont dédiés à l'utilisation de PostgreSQL, positionnez toujours les données dans un sous-répertoire, voire deux niveaux en dessous du point de montage (eg. <point de montage>/<version majeure>/<nom instance>).

3.2 PG_BASEBACKUP - PRÉSENTATION



- Outil intégré à PostgreSQL
- Prévu pour créer une instance secondaire
- Pour sauvegarde ponctuelle
 - PITR avec outils complémentaires

`pg_basebackup`¹ est une application cliente intégrée à PostgreSQL, au même titre que `pg_dump` ou `pg_dumpall`.

`pg_basebackup` a été conçu pour permettre l'initialisation d'une instance secondaire, et il peut donc être utilisé pour effectuer facilement une sauvegarde physique ponctuelle. Celle-ci inclut les fichiers et journaux nécessaires pour une restauration telle que l'instance était à la fin de la sauvegarde.

`pg_basebackup` peut aussi être à la base d'outils permettant le PITR (par exemple barman). Ces outils s'occupent en plus de l'archivage des journaux générés pendant et après la sauvegarde initiale, pour une restauration dans un état postérieur à la fin de cette sauvegarde.

3.2.1 pg_basebackup - Formats de sauvegarde



- `--format plain`
 - arborescence identique à l'instance sauvegardée
- `--format tar`
 - archive
 - compression : `-z, -Z (0..9)`

Le format par défaut de la sauvegarde est `plain`, ce qui signifie que les fichiers seront créés tels quels dans le répertoire de destination (ou les répertoires en cas de tablespaces). C'est idéal pour obtenir une copie immédiatement utilisable.

Pour une archive à proprement parler, préférer l'option `--format tar`. `pg_basebackup` génère alors une archive `base.tar` pour le PGDATA de l'instance, puis une archive `<oid>.tar` par tablespace. Les journaux récupérés seront également dans un fichier `.tar`.

¹<https://www.postgresql.org/docs/current/static/app-pgbasebackup.html>

L'option `--gzip` (`-z`) ajoute la compression gzip. Le niveau de compression peut également être spécifié avec `--compress=1 à 9` (`-Z`). Cela permet d'arbitrer entre la durée de la sauvegarde et sa taille.

3.2.2 pg_basebackup - Avantages



- Transfert des WAL pendant la sauvegarde
- Slot de réPLICATION automatique (temporaire voire permanent)
- Limitation du débit
- Relocalisation des tablespaces
- Fichier manifeste (v13+)
- Vérification des checksums (v11+)
- Sauvegarde possible à partir d'un secondaire
- Compression côté serveur ou client (v15+)
- Emplacement de la sauvegarde (client/server/blackhole) (v15+)
- Suivi : `pg_stat_progress_basebackup` (v13+)

`pg_basebackup` s'est beaucoup amélioré au fil des versions et son comportement a parfois changé. Regardez bien la documentation² de votre version.



Même avec un serveur un peu ancien, il est possible d'installer un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

Récupération des journaux :

`pg_basebackup` sait récupérer les fichiers WAL nécessaires à la restauration de la sauvegarde sans passer par la commande d'archivage. Il connaît deux méthodes :

Avec l'option `--wal-method fetch` (ou `-X`), les WAL générés pendant la sauvegarde seront demandés une fois celle-ci terminée, à condition qu'ils n'aient pas été recyclés entre-temps (ce qui peut nécessiter un slot de réPLICATION, ou éventuellement une configuration élevée du paramètre `wal_keep_size/wal_keep_segments`).

L'option par défaut depuis PostgreSQL 10 est cependant `-X stream` : les WAL sont récupérés non pas en fin de sauvegarde, mais en *streaming* pendant celle-ci. Cela nécessite néanmoins l'utilisation d'un *wal sender* supplémentaire, le paramètre `max_wal_senders` doit parfois être augmenté en conséquence.

²<https://docs.postgresql.fr/current/app-pgbasebackup.html>

Rappelons que si l'archivage des WAL n'est pas actif, la sauvegarde effectuée ne sera utilisée que pour restaurer l'instance telle qu'elle était au moment de la fin de la sauvegarde : il ne sera pas possible de réaliser une restauration PITR.

À l'inverse, `-X none` peut être utile si la récupération des journaux est réalisée par ailleurs (généralement par `archive_command` ou `archive_library`). Attention, l'archive réalisée avec `pg_basebackup` ne sera alors pas « complète », et ne pourra pas être restaurée sans ces archives des journaux (il faudra indiquer où aller les chercher avec `restore_command`.)

SLOTS de réPLICATION :

Par défaut, `pg_basebackup` va créer un slot de réPLICATION temporaire sur le serveur pour sécuriser la sauvegarde. Il disparaîtra une fois celle-ci terminée.

Pour faciliter la mise en place d'une instance secondaire, et garantir que tous les journaux nécessaires seront encore sur le primaire à son démarrage, il est possible de créer un slot de réPLICATION permanent, et de le fournir à `pg_basebackup` avec `--slot nom_du_slot`. `pg_basebackup` peut le créer lui-même avec `--create`. Si l'on préfère le créer préalablement, il suffit d'exécuter la requête suivante :

```
SELECT pg_create_physical_replication_slot ('nom_du_slot');
```

Rappelons qu'un slot initialisé mais inutilisé doit être rapidement supprimé pour ne pas mener à une dangereuse accumulation des journaux.

SÉCURISATION DE LA SAUVEGARDE :

Par défaut, `pg_basebackup` crée un fichier manifeste (à partir de PostgreSQL 13). Ce fichier contient la liste des fichiers sauvegardés, leur taille et leur somme de contrôle. Cela permet après coup de vérifier l'intégrité de la sauvegarde à l'aide de l'outil `pg_verifybackup`.

L'algorithme par défaut de la somme de contrôle, CRC32, suffit pour détecter une erreur technique accidentelle ; d'autres algorithmes disponibles permettent de détecter une manipulation volontaire de la sauvegarde.

VÉRIFICATION DES SOMMES DE CONTRÔLE :

À partir de la version 11, une sauvegarde avec `pg_basebackup` entraîne la vérification des sommes de contrôle de l'instance. Cela garantit que la sauvegarde n'héritera pas d'une corruption existante, sinon l'outil tombe en erreur.

L'option `--no-verify-checksums` autorise la sauvegarde d'une instance où une corruption est détectée (sauvegarde aussi problématique, certes, mais qui peut permettre de travailler sur la récupération, ou de sauver l'essentiel).

EMPLACEMENT DE LA SAUVEGARDE

À partir de la version 15, l'option `--target` permet de spécifier où la sauvegarde doit être réalisée :

- sur le serveur où la commande est lancée (`client`) ;
- sur le serveur de base de données (`server`) ;
- dans le vide (`blackhole`).

Des destinations peuvent être ajoutées par des extensions, `basebackup_to_shell`³ est fournie à titre d'exemple et permet de réaliser une

Lorsque la destination `server` est choisie, plusieurs restrictions s'appliquent à la sauvegarde :

- le format doit être `tar` ;
- l'utilisateur employé pour la réaliser doit être membre du rôle `pg_write_server_files` ;
- la méthode de récupération des WAL doit être `fetch` ou `none`.

Compression de la sauvegarde :

À partir de la version 15, il est possible de demander la compression de la sauvegarde avec un grand niveau de personnalisation :

- algorithme de compression parmi `gzip`, `lz4` et `zstd` ;
- rapidité de la compression hors parallélisme (`lz4`) ;
- niveau de compression (`zstd`) ;
- parallélisation de la compression (`zstd`) ;
- localisation de la compression (serveur ou client).

Cela permet de gérer différents scénarios et d'éviter certains goulets d'étranglement lors d'une sauvegarde.

Autres options :

Le débit de la sauvegarde est configurable avec l'option `--max-rate=` (`-r`) pour limiter l'impact sur l'instance ou le réseau. Cette restriction de débit ne concerne pas les journaux transférés en parallèle (`-X stream`).

Pour gagner un peu de temps, si l'instance n'est pas trop chargée, `--checkpoint=fast` accélère le checkpoint préalable à la sauvegarde.

Avec une sauvegarde `plain`, il est possible de modifier sur la cible les chemins des éventuels tablespaces avec l'option `--tablespace-mapping=<vieuxrep>=<nouveaurep>` (ou `-T`), et de re-localiser le répertoire des fichiers WAL avec l'option `--waldir=<nouveau chemin>`.

Depuis un secondaire :

`pg_basebackup` permet nativement de réaliser une sauvegarde à partir d'une instance secondaire. Le paramétrage nécessaire figure plus bas.

Suivi :

Pour suivre le déroulement de la sauvegarde depuis un terminal, il existe l'option `--progress` (`-P`).

À partir de PostgreSQL 13, il existe aussi une vue pour ce suivi : `pg_stat_progress_basebackup`⁴.

Options complètes :

Pour mémoire, toutes les options disponibles sont celles-ci (en version 15) :

³<https://docs.postgresql.fr/current/basebackup-to-shell.html>

⁴<https://docs.postgresql.fr/current/progress-reporting.html#BASEBACKUP-PROGRESS-REPORTING>

```
$ pg_basebackup --help
```

pg_basebackup prend une sauvegarde binaire d'un serveur PostgreSQL en cours d'exécution.

Usage :

```
pg_basebackup [OPTION]...
```

Options contrôlant la sortie :

-D, --pgdata=RÉPERTOIRE	reçoit la sauvegarde de base dans ce répertoire
-F, --format=p t	format en sortie (plain (par défaut), tar)
-r, --max-rate=TAUX	taux maximum de transfert du répertoire de données (en Ko/s, ou utiliser le suffixe « k » ou « M »)
-R, --write-recovery-conf	écrit la configuration pour la réPLICATION
-t, --target=CIBLE[:DETAIL]	cible de sauvegarde (si autre que client)
-T, --tablespace-mapping=ANCIENREP=NOUVEAUREP	déplace le répertoire ANCIENREP en NOUVEAUREP
--waldir=RÉP_WAL	emplacement du répertoire des journaux de transactions
-X, --wal-method=none fetch stream	inclus les journaux de transactions requis avec la méthode spécifiée
-z, --gzip	comprime la sortie tar
-Z, --compress=[{client server}-]METHODE[:DETAIL]	comprime sur le client ou le serveur comme indiqué
-Z, --compress=none	ne compress pas la sortie tar

Options générales :

-c, --checkpoint=fast spread	exécute un CHECKPOINT rapide ou réparti
--create-slot	crée un slot de réPLICATION
-l, --label=LABEL	configure le label de sauvegarde
-n, --no-clean	ne nettoie pas en cas d'erreur
-N, --no-sync	n'attend pas que les modifications soient proprement écrites sur disque
-P, --progress	affiche la progression de la sauvegarde
-S, --slot=NOMREP	slot de réPLICATION à utiliser
-v, --verbose	affiche des messages verbeux
-V, --version	affiche la version puis quitte
--manifest-checksums=SHA{224,256,384,512} CRC32C NONE	utilise cet algorithme pour les sommes de contrôle du manifeste
--manifest-force-encode	encode tous les noms de fichier dans le manifeste en hexadécimal
--no-estimate-size	ne réalise pas d'estimation sur la taille de la sauvegarde côté serveur
--no-manifest	supprime la génération de manifeste de sauvegarde
--no-slot	empêche la création de slots de réPLICATION temporaires
--no-verify-checksums	ne vérifie pas les sommes de contrôle
-?, --help	affiche cette aide puis quitte

Options de connexion :

-d, --dbname=CHAÎNECONNEX	chaîne de connexion
-h, --host=HÔTE	hôte du serveur de bases de données ou

<code>-p, --port=PORT</code>	répertoire des sockets numéro de port du serveur de bases de données
<code>-s, --status-interval=INTERVAL</code>	durée entre l'envoi de paquets de statut au serveur (en secondes)
<code>-U, --username=UTILISATEUR</code>	se connecte avec cet utilisateur
<code>-w, --no-password</code>	ne demande jamais le mot de passe
<code>-W, --password</code>	force la demande du mot de passe (devrait survenir automatiquement)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>. Page d'accueil de PostgreSQL : <<https://www.postgresql.org/>>

3.2.3 pg_basebackup - Limitations



- Configuration *streaming* nécessaire
- Pas de configuration de l'archivage
- Pas d'association WAL archivés / sauvegarde
- Pas de politique de rétention
 - sauvegarde ponctuelle
- Pas de gestion de la restauration !
 - manuel : `recovery.signal`, `restore_command...`
 - pour un secondaire : `--write-recovery-conf`

Configuration :

pg_basebackup étant conçu pour la mise en place d'une instance en réPLICATION, l'instance principale nécessite d'être configurée en conséquence :

- `max_wal_senders` doit avoir une valeur supérieure à 0 pour permettre à pg_basebackup de se connecter (au moins 2 si on utilise le transfert des WAL par streaming) — c'est le cas par défaut ;
- le fichier `pg_hba.conf` de l'instance principale doit être configuré pour autoriser les connexions de type `replication` depuis la machine où la sauvegarde est déclenchée, par exemple ainsi :

```
host replication repli_user 192.168.0.100/32 scram-sha-256
```

Dans l'idéal, l'utilisateur employé est dédié à la réPLICATION. Pour automatiser, stocker le mot de passe nécessaire dans un fichier `.pgpass`.

L'archivage n'est pas géré par pg_basebackup. Il ne récupère par *streaming* que les journaux nécessaires à la cohérence de sa sauvegarde. Il faudra paramétrier `archive_command` ou `archive_library` à la main pour une sauvegarde PITR.

Si la sauvegarde est effectuée à partir d'une instance secondaire :

- ces paramétrages sont nécessaires (et en place par défaut) :
 - instance secondaire ouverte en lecture (hot_standby à on) ;
 - max_wal_senders supérieur 0 et droits en place pour permettre à pg_basebackup de se connecter ;
 - écriture complète des pages dans les WAL activée (full_page_writes à on) ;
- pour du PITR :
 - l'archivage des fichiers WAL doit être configuré indépendamment.
 - une attention particulière doit être apportée au fait que tous les fichiers WAL nécessaires à la restauration ont bien été archivés.

Gestion des sauvegardes :

La gestion des sauvegardes (rétention, purge...) n'est pas prévue dans l'outil.

pg_basebackup n'effectue pas non plus de lien entre les WAL archivés et les sauvegardes effectuées (si pg_basebackup ne les sauvegarde pas lui-même avec l'option -X).

Restauration :

pg_basebackup n'offre pas d'outil ni d'option pour la restauration.

La copie est directement utilisable, éventuellement après déplacement, décompression des .tar.gz. Mais généralement on ajoutera un recovery.signal (recovery.conf jusqu'en v11), et on définira la restore_command pour récupérer les archives. Dans l'idéal, restore_command sera déjà prête dans postgresql.conf (ou un modèle pour recovery.conf).

Si le but est de monter un serveur secondaire de l'instance copiée, il existe une option utile :--write-recovery-conf (ou -R), qui génère la configuration nécessaire dans le répertoire de la sauvegarde (recovery.conf, ou postgresql.auto.conf et fichier vide standby.signal), avec les paramètres pour une réPLICATION en *streaming*.

3.3 PGBACKREST - PRÉSENTATION GÉNÉRALE



- David Steele (Crunchy Data)
- Langage : **C**
- License : **MIT** (libre)
- Type d'interface : **CLI** (ligne de commande)

3.3.1 pgBackRest - Fonctionnalités



- Gère la sauvegarde et la restauration
 - *pull* ou *push*, multidépôts
 - mono ou multi-serveurs
- Indépendant des commandes système
 - utilise un protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multi-thread, sauvegarde depuis un secondaire, archivage asynchrone...
- Projet mature

pgBackRest⁵ est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser notamment),
- le support intégré de dépôts S3 ou Azure ;
- la sauvegarde depuis un serveur secondaire ;

⁵<https://pgbackrest.org/>

- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

Le projet est récent, très actif, considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest⁶.

3.3.2 pgBackRest - Sauvegardes



- Type de sauvegarde : **physique/PITR** (à chaud)
- Type de stockage : **local, push ou pull**
- Planification : **crontab**
- Complètes, différentielles et incrémentales
- Compression des WAL

pgBackRest gère uniquement des sauvegardes physiques.

Il peut fonctionner soit en local (directement sur le serveur hébergeant l'instance à sauvegarder) pour un stockage local des sauvegardes, soit être exécuté depuis un serveur distant, déléguant ainsi l'ordonnancement, la compression et le stockage des données à celui-ci.

La technique utilisée pour la prise de sauvegarde repose sur le mécanisme interne standard et historique : pg_backup_start(), copie des fichiers, pg_backup_stop().

3.3.3 pgBackRest - Restauration



- Depuis le serveur de BDD avec un dépôt local ou à distance
- Point dans le temps : date, identifiant de transaction, timeline ou point de restauration

La restauration d'une sauvegarde peut se faire soit localement, si les sauvegardes sont stockées en local, soit à distance. Dans ce dernier cas, les données à restaurer seront transférées via SSH.

Plusieurs types de point dans le temps peuvent être utilisés comme cible :

⁶https://github.com/pgstef/check_pgbackrest/

- la date ;
- un identifiant de transaction ;
- une timeline (en cas de divergence de timeline, pgBackRest peut restaurer les transactions issues d'une timeline précise) ;
- un point de restauration créé par un appel préalable à la fonction :
 - `pg_create_restore_point()`.

3.3.4 pgBackRest - Installation



- Accéder au dépôt communautaire PGDG
- Installer le paquet pgbackrest

pgBackRest est disponible sur le dépôt communautaire maintenu par la communauté PostgreSQL pour les systèmes d'exploitation disposant des gestionnaires de paquet au format deb (Debian, Ubuntu...)⁷ ou rpm (Red Hat, Rocky Linux, CentOS, Fedora...)⁸.

Il est recommandé de manière générale de privilégier une installation à partir de ces paquets plutôt que par les sources, essentiellement pour des raisons de maintenance.

⁷<https://apt.postgresql.org/pub/repos/apt/>

⁸<https://yum.postgresql.org/>

3.3.5 pgBackRest - Utilisation



Usage:
 `pgbackrest [options] [command]`

Commands:

<code>annotate</code>	Add or modify backup annotation.
<code>archive-get</code>	Get a WAL segment from the archive.
<code>archive-push</code>	Push a WAL segment to the archive.
<code>backup</code>	Backup a database cluster.
<code>check</code>	Check the configuration.
<code>expire</code>	Expire backups that exceed retention.
<code>help</code>	Get help.
<code>info</code>	Retrieve information about backups.
<code>repo-get</code>	Get a file from a repository.
<code>repo-ls</code>	List files in a repository.
<code>restore</code>	Restore a database cluster.
<code>server</code>	pgBackRest server.
<code>server-ping</code>	Ping pgBackRest server.
<code>stanza-create</code>	Create the required stanza data.
<code>stanza-delete</code>	Delete a stanza.
<code>stanza-upgrade</code>	Upgrade a stanza.
<code>start</code>	Allow pgBackRest processes to run.
<code>stop</code>	Stop pgBackRest processes from running.
<code>verify</code>	Verify contents of the repository.
<code>version</code>	Get version.

pgBackRest propose différentes commandes pouvant être passées en argument afin de contrôler les actions.

L'usage de ces différentes commandes sera détaillé ultérieurement.

3.3.6 pgBackRest - Configuration



- /etc/pgbackrest.conf
- Configuration générale dans la section [global]
- Chaque instance à sauvegarder doit avoir sa propre section, appelée stanza
- possibilité d'éclater la configuration dans plusieurs fichiers : config-include-path

Le format de configuration INI permet de définir des sections, qui sont matérialisées sous la forme d'une ligne : [nomdesection].

pgBackRest s'attend à lire un fichier de configuration contenant la section [global], contenant les paramètres de configuration globaux, et une section par instance à sauvegarder.

pgBackRest utilise le terme stanza pour regrouper l'ensemble des configurations à appliquer pour une **instance** à sauvegarder.

Exemple de configuration :

```
[global]
repo1-path=/var/lib/pgsql/10/backups

[erp_prod]
pg1-path=/var/lib/pgsql/10/data
```

Il peut y avoir plusieurs stanzas déclarées dans le fichier, notamment s'il est situé sur le serveur où sont stockées les sauvegardes de plusieurs instances.

Pour des questions de lisibilité, il est possible de créer un fichier de configuration par instance à sauvegarder. Le nom du fichier doit se terminer par .conf pour être pris en compte. Les fichiers doivent être regroupés dans un répertoire référencé par le paramètre config-include-path.

3.3.7 pgBackRest - Configuration PostgreSQL



- Adapter l'archivage dans le fichier postgresql.conf

```
archive_mode = on
wal_level = replica
archive_command = 'pgbackrest --stanza=erp_prod archive-push %p'
```

Il est nécessaire d'activer l'archivage des journaux de transactions en positionnant le paramètre `archive_mode` à `on` et en définissant un niveau d'enregistrement d'informations dans les journaux de transactions (`wal_level`) supérieur ou égal à `replica` (ou `archive` avant la version 9.6).

pgBackRest fournit une commande permettant de simplifier la configuration de l'archivage. Pour l'utiliser, il faut configurer le paramètre `archive_command` pour qu'il utilise l'option `archive-push` de la commande `pgbackrest`. Il faut également fournir à cette commande le nom de la stanza à utiliser.

3.3.8 pgBackRest - Configuration globale



- Fichier `pgbackrest.conf`
- Section **[global]** pour la configuration globale

```
[global]
process-max=1
repo1-path=/var/lib/pgbackrest
```

- **process-max** : nombre de processus maximum à utiliser pour la compression et le transfert des sauvegardes ;
- **repo1-path** : chemin où seront stockées les sauvegardes et les archives ;
- **repo-cipher-pass** : passphrase à utiliser pour chiffrer/déchiffrer le répertoire des sauvegardes ;
- **log-level-console** : par défaut à `warn`, définit le niveau de traces des commandes exécutées en console.

3.3.9 pgBackRest - Configuration de la rétention



- Type de rétention des sauvegardes complètes

```
repo1-retention-full-type=count|time
```

- **Nombre** de sauvegardes complètes

```
repo1-retention-full=2
```

- **Nombre** de sauvegardes différentielles

```
repo1-retention-diff=3
```

La politique de rétention des sauvegardes complètes peut être configurée avec l'option `repo1-retention-full-type`. Elle peut prendre deux valeurs :

- `count` : le nombre de sauvegardes à conserver, c'est la valeur par défaut ;
- `time` : un nombre de jours pendant lequel on doit pouvoir restaurer, c'est-à-dire que l'on doit avoir au moins une sauvegarde plus vieille que ce nombre de jours.

Voici un exemple pour illustrer la mode de rétention `time`, dont le fonctionnement n'est pas très intuitif. Si l'on dispose des trois sauvegardes complètes suivantes :

- F1 : 25 jours ;
- F2 : 20 jours ;
- F3 : 10 jours.

Avec une rétention de 15 jours, seule la sauvegarde F1 sera supprimée. F2 sera conservée, car il doit exister au moins une sauvegarde de plus de 15 jours pour garantir de pouvoir restaurer pendant cette période.

Il est possible de différencier le nombre de sauvegardes complètes et différentielles. La rétention pour les sauvegardes différentielles ne peut être définie qu'en nombre.

Lorsqu'une sauvegarde complète expire, toutes les sauvegardes différentielles et incrémentales qui lui sont associées expirent également.

3.3.10 pgBackRest - Configuration SSH



- Utilisateur `postgres` pour les serveurs PostgreSQL
- Échanger les clés SSH publiques entre les serveurs PostgreSQL et le serveur de sauvegarde
- Configurer `repo1-host*` dans la `pgbackrest.conf`

Dans le cadre de la mise en place de sauvegardes avec un stockage des données sur un serveur tiers, pgBackRest fonctionnera par SSH.

Il est donc impératif d'autoriser l'authentification SSH par clé, et d'échanger les clés publiques entre les différents serveurs hébergeant les instances PostgreSQL et le serveur de sauvegarde.

Il faudra ensuite adapter les paramètres `repo1-host*` dans la configuration de pgBackRest.

- **repo1-host** : hôte à joindre par SSH ;
- **repo1-host-user** : utilisateur pour la connexion SSH ;
- ...

3.3.11 pgBackRest - Configuration TLS



- Alternative au SSH
- {repo1|pg1}-host-type = tls
- paramètres tls-server-{address|auth|cert|key|ca}
- paramètres repo1-host-{cert|key|ca}
- paramètres pg1-host-{cert|key|ca}
- pgbackrest server

Il existe une alternative à l'utilisation de SSH qui consiste à configurer un serveur TLS en valorisant le paramètre `repo1-host-type` et `pg1-host-type` à `tls` (défaut : `ssh`). La configuration du serveur se fait ensuite avec les paramètres :

- `tls-server-address` : adresse IP sur laquelle le serveur écoute pour servir des requêtes clients ;
- `tls-server-auth` : la liste des clients autorisés à se connecter sous la forme `<client-cn>=<stanza>` ;
- `tls-server-ca-file` : certificat de l'autorité ;
- `tls-server-cert-file` : certificat du serveur ;
- `tls-server-key-file` : clé du serveur.

Il faut ensuite configurer l'accès au dépôt de sauvegarde :

- `repo1-host-type=tls` : la connexion au dépôt utilise TLS ;
- `repo1-host-cert-file` : certificat pour se connecter au dépôt ;
- `repo1-host-key-file` : clé pour se connecter au dépôt ;
- `repo1-host-ca-file` : certificat de l'autorité.

Exemple de configuration :

```
[global]
repo1-host=backrest-srv
repo1-host-user=backrest
repo1-host-type=tls
repo1-host-cert-file=/etc/certs/srv1-cert.pem
repo1-host-key-file=/etc/certs/srv1-key.pem
repo1-host-ca-file=/etc/certs/CA-cert.pem

tls-server-address=*
tls-server-cert-file=/etc/certs/srv1-cert.pem
tls-server-key-file=/etc/certs/srv1-key.pem
tls-server-ca-file=/etc/certs/CA-cert.pem
tls-server-auth=backrest-srv=main

[main]
pg1-path=/var/lib/pgsql/14/data
```

Sur le serveur de sauvegarde, la configuration est similaire :

- pg1-host-type=tls : la connexion au serveur PostgreSQL utilise TLS ;
- pg1-host-cert-file : certificat pour se connecter au serveur de bases de données ;
- pg1-host-key-file : certificat pour se connecter au serveur de bases de données ;
- pg1-host-ca-file : certificat de l'autorité.

Exemple de configuration du serveur de sauvegarde :

```
[global]
repo1-path=/var/lib/pgbackrest
repo1-retention-full=2

tls-server-address=*
tls-server-cert-file=/etc/certs/backrest-srv-cert.pem
tls-server-key-file=/etc/certs/backrest-srv-key.pem
tls-server-ca-file=/etc/certs/CA-cert.pem
tls-server-auth=srv1=main

[main]
pg1-host=srv1
pg1-port=5432
pg1-path=/var/lib/pgsql/14/data

pg1-host-type=tls
pg1-host-cert-file=/etc/certs/backrest-srv-cert.pem
pg1-host-key-file=/etc/certs/backrest-srv-key.pem
pg1-host-ca-file=/etc/certs/CA-cert.pem
```

Le serveur TLS doit ensuite être démarré avec la commande `pgbackrest server`. Un service est prévu à cet effet et installé automatiquement sur les distributions de type RedHat et Debian.

Un ping vers le serveur TLS peut être testé avec la commande `pgbackrest server-ping <hote>`. Suivant les distributions, il peut être nécessaire d'ouvrir le port 8432 (valeur par défaut de `tls-server-port`).

```
[postgres@backrest log]$ pgbackrest server-ping srv1
INFO: server-ping command begin 2.41: [srv1] --exec-id=7467-76e4b8cf
--log-level-console=info --tls-server-address=*
INFO: server-ping command end: completed successfully (47ms)
```

Génération des clés et certificats auto-signés :

```
# Générer une clé privée et un certificat pour l'autorité de certification
openssl req -new -x509 \
             -days 365 \
             -nodes \
             -out CA-cert.pem \
             -keyout CA-key.pem \
             -subj "/CN=root-ca"

# Générer une clé privée et demande de certificat (CSR)
openssl req -new -nodes \
             -out backrest-srv-csr.pem \
             -keyout backrest-srv-key.pem \
```

```
-subj "/CN=backrest-srv"
openssl req -new -nodes \
             -out srv1-csr.pem \
             -keyout srv1-key.pem \
             -subj "/CN=srv1"

# Générer le certificat signé
openssl x509 -req -in backrest-srv-csr.pem \
               -days 365 \
               -CA CA-cert.pem \
               -CAkey CA-key.pem \
               -CAcreateserial \
               -out backrest-srv-crt.pem
openssl x509 -req -in srv1.csr \
               -days 365 \
               -CA CA-cert.pem \
               -CAkey CA-key.pem \
               -CAcreateserial \
               -out srv1-crt.pem
```

3.3.12 pgBackRest - Configuration par instance



- Une section par instance
 - appelée stanza

Après avoir vu les options globales, nous allons voir à présent les options spécifiques à chaque instance à sauvegarder.

3.3.13 pgBackRest - Exemple configuration par instance



- Section spécifique par instance
- Permet d'adapter la configuration aux différentes instances
- Exemple

```
[erp_prod]
pg1-path=/var/lib/pgsql/10/data
```

Une stanza définit l'ensemble des configurations de sauvegardes pour un cluster PostgreSQL

spécifique. Chaque section stanza définit l'emplacement du répertoire de données ainsi que l'hôte/utilisateur si le cluster est distant. Chaque configuration de la partie globale peut être surchargée par stanza.

Le nom de la stanza est important et doit être significatif car il sera utilisé lors des tâches d'exploitation pour identifier l'instance cible.

Il est également possible d'ajouter ici des recovery-option afin de personnaliser les options du fichier recovery.conf (ou postgresql.auto.conf en version 12 et supérieures) qui sera généré automatiquement à la restauration d'une sauvegarde.

3.3.14 pgBackRest - Initialiser le répertoire de stockage des sauvegardes



- Pour initialiser le répertoire de stockage des sauvegardes

```
$ sudo -u postgres pgbackrest --stanza=erp_prod stanza-create
```

- Vérifier la configuration de l'archivage

```
$ sudo -u postgres pgbackrest --stanza=erp_prod check
```

La commande d'initialisation doit être lancée sur le serveur où se situe le répertoire de stockage après que la stanza ait été configurée dans pgbackrest.conf.

La commande check valide que pgBackRest et le paramètre archive_command soient correctement configurés. Les commandes pg_create_restore_point('pgBackRest Archive Check') et pg_switch_wal() sont appelées à cet effet pour forcer PostgreSQL à archiver un segment WAL.

3.3.15 pgBackRest - Effectuer une sauvegarde



- Pour déclencher une nouvelle sauvegarde complète

```
$ sudo -u postgres pgbackrest --stanza=erp_prod --type=full backup
```

- Types supportés : incr, diff, full

- La plupart des paramètres peuvent être surchargés

La sauvegarde accepte de nombreux paramètres dont :

- **--archive-copy** : archive les WAL dans la sauvegarde en plus de les mettre dans le dépôt de WAL ;
- **--backup-standby** : déclenche la sauvegarde sur un serveur secondaire ;
- **--no-online** : fait une sauvegarde à froid ;
- **--resume** : reprend une sauvegarde précédemment échouée en conservant les fichiers qui n'ont pas changés ;
- **--start-fast** : exécuter le checkpoint immédiatement.

Exemple de sortie d'une sauvegarde complète :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod --type=full backup |grep P00
P00    INFO: backup command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data --process-max=1
--repo1-path=/var/lib/pgsql/12/backups --repo1-retention-full=1
--stanza=erp_prod --type=full
P00    INFO: execute non-exclusive pg_start_backup() with label
"pgBackRest backup started at 2019-11-26 12:39:26":
backup begins after the next regular checkpoint completes
P00    INFO: backup start archive = 00000001000000000000000005, lsn = 0/5000028
P00    INFO: full backup size = 24.2MB
P00    INFO: execute non-exclusive pg_stop_backup() and wait for all WAL
segments to archive
P00    INFO: backup stop archive = 00000001000000000000000005, lsn = 0/5000100
P00    INFO: new backup label = 20191126-123926F
P00    INFO: backup command end: completed successfully
P00    INFO: expire command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data --process-max=1
--repo1-path=/var/lib/pgsql/12/backups --repo1-retention-full=1
--stanza=erp_prod --type=full
P00    INFO: expire full backup 20191126-123848F
P00    INFO: remove expired backup 20191126-123848F
P00    INFO: expire command end: completed successfully
```

La commande se charge automatiquement de supprimer les sauvegardes devenues obsolètes.

Il est possible d'ajouter des annotations aux sauvegardes comme ceci :

```
$ sudo -u postgres pgbackrest
--stanza=erp_prod
--type=full
--annotation=desc="Premier backup"
backup
```

L'annotation peut être observé en affichant les informations du *backup set*.

3.3.16 pgBackRest - Lister les sauvegardes



- Lister les sauvegardes présentes et leur taille

```
$ sudo -u postgres pgbackrest --stanza=erp_prod info  
- ou une sauvegarde spécifique (backup set)  
$ sudo -u postgres pgbackrest --stanza=erp_prod --set 20221026-071751F  
↪ info
```

Exemple de sortie des commandes :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod info  
stanza: erp_prod  
  status: ok  
  cipher: none  
  
db (current)  
  wal archive min/max (14): 000000030000000000000019/000000030000000000000001B  
  
  full backup: 20221026-071751F  
    timestamp start/stop: 2022-10-26 07:17:51 / 2022-10-26 07:17:57  
    wal start/stop: 000000030000000000000001B / 0000000300000000000000001B  
    database size: 25.2MB, database backup size: 25.2MB  
    repo1: backup set size: 3.2MB, backup size: 3.2MB  
  
$ sudo -u postgres pgbackrest --stanza=erp_prod --set 20221026-071751F info  
stanza: erp_prod  
  status: ok  
  cipher: none  
  
db (current)  
  wal archive min/max (14): 000000030000000000000019/000000030000000000000001B  
  
  full backup: 20221026-071751F  
    timestamp start/stop: 2022-10-26 07:17:51 / 2022-10-26 07:17:57  
    wal start/stop: 000000030000000000000001B / 0000000300000000000000001B  
    lsn start/stop: 0/1B000028 / 0/1B000100  
    database size: 25.2MB, database backup size: 25.2MB  
    repo1: backup set size: 3.2MB, backup size: 3.2MB  
    database list: postgres (13748)  
    annotation(s)  
      desc: Premier backup
```

3.3.17 pgBackRest - Planification



- Pas de planificateur intégré
 - le plus simple est d'utiliser cron

La planification des sauvegardes peut être faite par n'importe quel outil de planification de tâches, le plus connu étant cron.

pgBackRest maintient les traces de ses activités par défaut dans /var/log/pgbackrest avec un niveau de traces plus élevé qu'en console. Il n'est donc généralement pas nécessaire de gérer cela au niveau de la planification.

3.3.18 pgBackRest - Dépôts



- Plusieurs dépôts simultanés possibles
 - Sauvegarde des journaux en parallèle
 - --repo1-option=..., appel avec --repo=1
- Types : POSIX (NFS, ssh), CIFS, SFTP
- Cloud : S3, Azure, GFS

pgBackRest permet de maintenir plusieurs dépôts de sauvegarde simultanément⁹.

Un intérêt est de gérer des rétentions différentes. Par exemple un dépôt local contiendra juste les dernières sauvegardes et journaux, alors qu'un deuxième dépôt sera sur un autre site plus lointain, éventuellement moins cher, et/ou une rétention supérieure.

Les propriétés des différents dépôts (type, chemin, rétention...) se définissent avec les options repo1-path, repo2-path, etc. Désigner un dépôt particulier se fait avec --repo=1 par exemple.

Une sauvegarde se fait en désignant le dépôt, mais l'archivage est simultané sur tous les dépôts. L'archivage asynchrone est conseillé dans ce cas.

Les types de dépôts supportés sont ceux montés sur le serveur ou accessibles par ssh, NFS (avec la même attention aux options de montage que pour PostgreSQL¹⁰), CIFS (avec des restrictions sur les

⁹<https://pgbackrest.org/configuration.html#section-repository>

¹⁰<https://docs.postgresql.fr/current/creating-cluster.html#CREATING-CLUSTER-FILESYSTEM>

liens symboliques ou le fsync), mais aussi ceux à base de *buckets* : S3 ou compatible, Google Cloud, et Azure Blob.

3.3.19 pgBackRest - bundling et sauvegarde incrémentale en mode block



- Regrouper les petits fichiers dans des bundles

```
repo1-bundle=y
```

- Sauvegarde incrémentale en mode block (requiert le bundling)

```
repo1-bundle=y
```

```
repo1-block=y
```

« Bundling » des petits fichiers

Si une instances contient de nombreux petits fichiers (base aux nombreuses toutes petites tables, pg_commit_ts rempli à cause de track_commit_timestamp à on, très nombreuses petites partitions, chacune avec des fichiers annexes...), il est possible de les regrouper par paquets.

```
repo1-bundle=y
# défauts
repo1-bundle-limit=2MiB
repo-bundle-size=20MiB
```

Les bundles ne sont pas conservés en cas de backup interrompu puis redémarré. Les fichiers doivent être re-sauvegardés lors de la relance. Bundles et hard-links ne peuvent pas être utilisés ensemble.

Cette fonctionnalité est particulièrement utile avec un stockage comme S3 où le coût de création de fichier est prohibitif.

Sauvegarde incrémentale en mode bloc (2.46)

La sauvegarde incrémentale par bloc permet plus de granularité en divisant les fichiers en blocs qui peuvent être sauvegardés indépendamment. C'est particulièrement intéressant pour des fichiers avec peu de modifications, car pgBackRest ne sauvegardera que quelques blocs au lieu du fichier complet (les tables et index sont segmentés en fichiers de 1 Go). Cela permet donc d'économiser de l'espace dans le dépôt de sauvegarde et accélère les restaurations par delta.

La sauvegarde incrémentale par bloc doit être activée sur tous les types de sauvegardes : full, incrémentielle ou différentielle. Cela aura pour impact de rendre la sauvegarde full un peu plus grosse du fait de la création de fichier de cartographie des blocs. En revanche, les sauvegardes différentielles et incrémentielles suivantes pourront utiliser cette cartographie pour économiser de l'espace.

La taille du bloc pour un fichier donné est définie en fonction de l'âge et de la taille du fichier. Généralement, les fichiers les plus gros et/ou les plus anciens auront des tailles de bloc supérieures. Si un fichier est assez vieux, aucune cartographie ne sera créée.

Cette fonctionnalité nécessite le *bundling* et s'active ainsi :

```
repo1-block=y  
repo1-bundle=y
```

3.3.20 pgBackRest - Restauration



- Effectuer une restauration

```
$ sudo -u postgres pgbackrest --stanza=erp_prod restore
```

- Nombreuses options à la restauration, notamment :

- --delta
- --target / --type

Exemple de sortie de la commande :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod restore | grep P00
P00    INFO: restore command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data
--process-max=1 --repo1-path=/var/lib/pgsql/12/backups --stanza=erp_prod
P00    INFO: restore backup set 20191126-123926F
P00    INFO: write updated /var/lib/pgsql/12/data/postgresql.auto.conf
P00    INFO: restore global/pg_control (performed last to ensure aborted
restores cannot be started)
P00    INFO: restore command end: completed successfully
```

L'option --delta permet de ne restaurer que les fichiers qui seraient différents entre la sauvegarde et le répertoire de données déjà présent sur le serveur. Elle permet de gagner beaucoup de temps pour reprendre une restauration qui a été interrompue pour une raison ou une autre, pour resynchroniser une instance qui a « décroché », pour restaurer une version légèrement antérieure ou postérieure dans du PITR.

La cible à restaurer peut être spécifiée avec --target, associé à --type. Par exemple, pour restaurer à une date précise sur une timeline précise :

```
pgbackrest --stanza=instance --delta \
--type=time --target='2020-07-16 11:07:00' \
--target-timeline=4 \
--target-action=resume \
--set=20200716-102845F \
restore
```

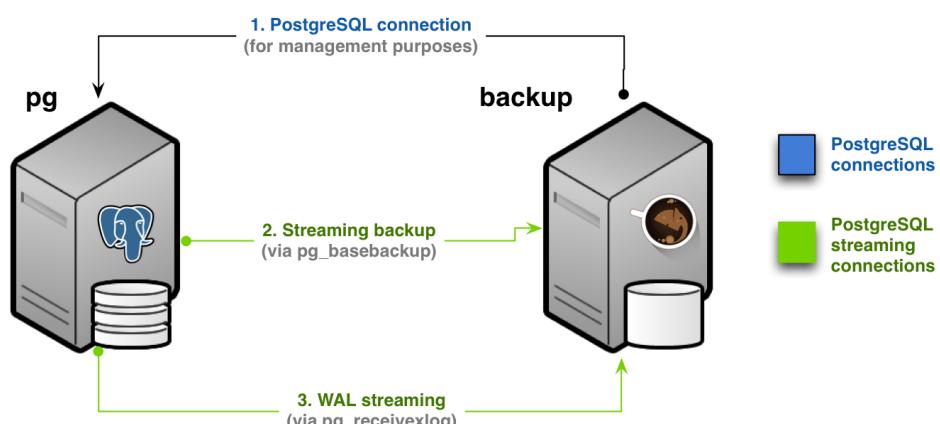
3.4 BARMAN - PRÉSENTATION GÉNÉRALE



- 2ndQuadrant Italia
- Langage: **python** >= 3.4
- OS: **Unix/Linux**
- Versions compatibles: **>= 8.3**
- License: **GPL3** (libre)
- Type d'interface: **CLI** (ligne de commande)

barman est un outil développé avec le langage python, compatible uniquement avec les environnements Linux/Unix. Il a été développé par la société 2ndQuadrant Italia (à présent partie de EDB) et distribué sous license GPL3.

3.4.1 Barman - Scénario « streaming-only »



Le scénario évoqué ci-dessus est communément appelé **streaming-only** puisqu'il ne requiert pas de connexion SSH pour les opérations de sauvegardes et d'archivage. Il faudra quand même configurer le SSH pour rendre possible la restauration depuis un serveur dédié ou faciliter la restauration en local.

En effet, les outils `pg_basebackup` et `pg_recvwal` sont utilisés pour ces opérations et se basent donc uniquement sur le protocole de réPLICATION. Cela a pour avantage que les améliorations faites aux outils dans le cadre des mises à jour majeures de PostgreSQL sont disponible directement dans Barman.

Par exemple :

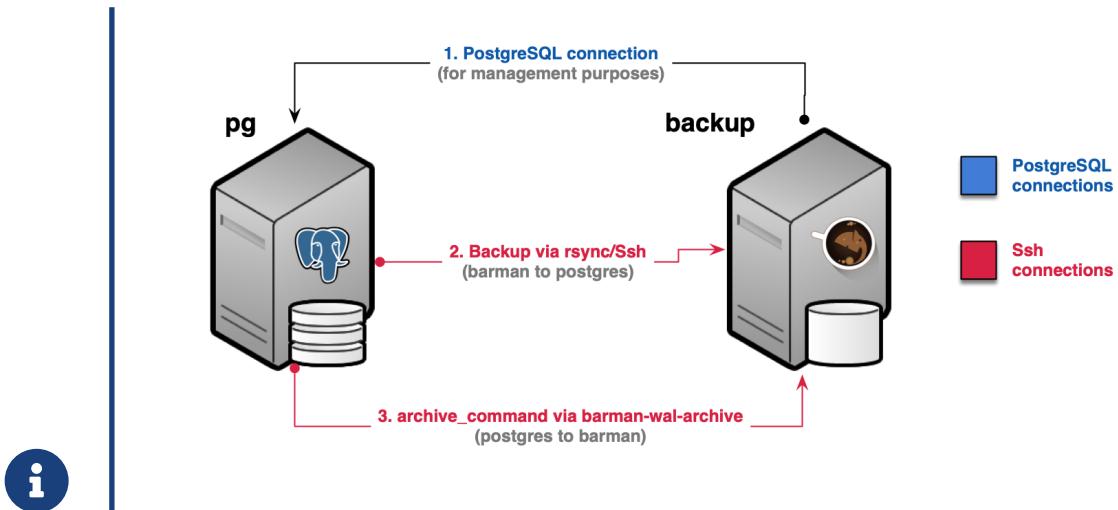
- la possibilité d'utiliser pg_stat_progress_basebackup pour la supervision ;
- les fichiers manifestes de sauvegarde et la vérification des sauvegardes ;
- la compression des sauvegardes.

Afin de garantir que l'instance sauvegardée conserve bien les WAL nécessaires, Barman permet de créer automatiquement un slot de réPLICATION. Il se chargera également de démarrer pg_receivewal grâce à sa tâche de maintenance programmée en crontab.

L'archivage peut être configuré à la place ou en plus du streaming des WAL.

Ce mode de sauvegarde permet de sauvegarder un serveur PostgreSQL installé sous Windows.

3.4.2 Barman - Scénario « rsync-over-ssh »



Ce deuxième scénario se base donc sur une connexion SSH afin de réaliser les sauvegardes et récupérer les archives des journaux WAL.

Cette méthode ne permet pas de compresser les sauvegardes mais permet de faire de la déduplication avec des *hard links* et de bénéficier de la parallélisation.

3.4.3 Barman - Sauvegardes



- Type de sauvegarde : **physique/PITR** (à chaud)
- Type de stockage : **local** ou **pull**
- Planification : **crontab**
- Méthodes :
 - **pg_backup_start()** / **rsync** / **pg_backup_stop()**
 - **pg_basebackup** / **pg_receivewal**
- Incrémentales : si **rsync + hardlink**
- Compression des WAL

Barman gère uniquement des sauvegardes physiques.

Il peut fonctionner soit en local (directement sur le serveur hébergeant l'instance à sauvegarder) pour un stockage local des sauvegardes, et peut aussi être exécuté depuis un serveur distant, déléguant ainsi l'ordonnancement, la compression et le stockage des données.

La technique utilisée pour la prise de sauvegarde repose sur le mécanisme interne standard et historique : `pg_backup_start()`, copie des fichiers, `pg_backup_stop()`.

Contrairement aux autres outils présentés, Barman peut également se servir de `pg_basebackup` et `pg_receivewal` pour récupérer les sauvegardes et les archives des journaux WAL.

Il est possible d'activer la dé-duplication de fichiers entre deux sauvegardes lorsque la méthode via `rsync` est employée.

3.4.4 Barman - Sauvegardes (suite)



- Limitation du débit réseau lors des transferts
- Compression des données lors des transferts via le réseau
- Sauvegardes concurrentes
- Hook pre/post sauvegarde
- Hook pre/post archivage WAL
- Compression WAL : `gzip`, `bzip2`, `pigz`, `pbzip2`, etc.
- Compression des données via `pg_basebackup`

Barman supporte la limitation du débit réseau lors du transfert des données sur un serveur tiers, ainsi que la compression des données à la volée le temps du transfert.

Quatre niveaux de scripts ancrés (*hooks*) sont possibles :

- **avant la sauvegarde** ;
- **après la sauvegarde** ;
- **avant l'archivage d'un WAL** ;
- **après l'archivage d'un WAL**.

Attention, l'opération d'archivage citée ici est celle effectuée par Barman lorsqu'il déplace et compresses un WAL à partir du répertoire `incoming_wals/` vers le répertoire `wals/`, il ne s'agit pas de l'archivage au sens PostgreSQL.

3.4.5 Barman - Politique de rétention



- **Durée** (jour/semaine)
- **Nombre** de sauvegardes

La politique de rétention peut être exprimée soit en nombre de sauvegardes à conserver, soit en fenêtre de restauration : une semaine, deux mois, etc.

3.4.6 Barman - Restauration



- Locale ou à distance
- Point dans le temps : date, identifiant de transaction, timeline ou point de restauration

La restauration d'une sauvegarde peut se faire soit localement, si les sauvegardes sont stockées en local, soit à distance. Dans ce dernier cas, les données à restaurer seront transférées via SSH.

Plusieurs types de point dans le temps peuvent être utilisés comme cible :

- la date ;
- un identifiant de transaction ;
- une timeline (en cas de divergence de timeline, barman peut restaurer les transactions issues d'une timeline précise) ;
- un point de restauration créé par un appel préalable à la fonction :
 - `pg_create_restore_point()`.

3.4.7 Barman - Installation



- Accéder au dépôt communautaire PGDG
- Installer les paquets barman et barman-cli

Barman est disponible sur le dépôt communautaire maintenu par la communauté PostgreSQL pour les systèmes d'exploitation disposant des gestionnaires de paquet au format deb (Debian, Ubuntu...)¹¹ ou rpm (Red Hat, Rocky Linux, CentOS, Fedora...)¹².

Il est recommandé de manière générale de privilégier une installation à partir des paquets issus du PGDG plutôt que par les sources, essentiellement pour des raisons de maintenance.

¹¹<https://apt.postgresql.org/pub/repos/apt/>

¹²<https://yum.postgresql.org/>

3.4.8 Barman - Utilisation



```
usage: barman [-h] [-v] [-c CONFIG] [--color {never,always,auto}] [-q]
   [-d]
   [-f {json,console}]

   {archive-wal,backup,check,check-backup,check-wal-archive,cron,
    delete,diagnose,generate-manifest,get-wal,help,keep,list-backup,
    list-backups,list-files,list-server,list-servers,put-wal,
    rebuild-xlogdb,receive-wal,recover,replication-status,show-backup,
    show-backups,show-server,show-servers,status,switch-wal,switch-xlog,
    sync-backup,sync-info,sync-wals,verify,verify-backup}

[...]
optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -c CONFIG, --config CONFIG
                        uses a configuration file (defaults: ~/.barman.conf,
                        /etc/barman.conf, /etc/barman/barman.conf)
  --color {never,always,auto}, --colour {never,always,auto}
                        Whether to use colors in the output (default:
   ↵  'auto')
  -q, --quiet           be quiet (default: False)
  -d, --debug           debug output (default: False)
  -f {json,console}, --format {json,console}
                        output format (default: 'console')
```

Barman propose différentes commandes pouvant être passées en argument afin de contrôler les actions.

L'usage de ces différentes commandes sera détaillé ultérieurement.

L'option `-c` (ou `--config`) permet d'indiquer l'emplacement du fichier de configuration. L'option `-q` (ou `--quiet`) désactive l'envoi de messages sur la sortie standard.

3.4.9 Barman - Configuration



- /etc/barman.conf
- Format INI
- Configuration générale dans la section [barman]
- Chaque instance à sauvegarder doit avoir sa propre section
- Un fichier de configuration par instance via la directive :

```
configuration_files_directory = /etc/barman.d
```

Le format de configuration INI permet de définir des sections, qui sont matérialisées sous la forme d'une ligne : [nomdesection].

Barman s'attend à lire un fichier de configuration contenant la section [barman], contenant les paramètres de configuration globaux, et une section par instance à sauvegarder, le nom de la section définissant ainsi le nom de l'instance.

Pour des questions de lisibilité, il est possible de créer un fichier de configuration par instance à sauvegarder. Ce fichier doit alors se trouver (par défaut) dans le dossier /etc/barman.d. Le nom du fichier doit se terminer par .conf pour être pris en compte.

3.4.10 Barman - Configuration utilisateur



- Utilisateur système barman

L'utilisateur système barman est utilisé pour les connexions SSH. Il faut donc penser à générer ses clés RSA, les échanger et établir une première connexion avec les serveurs hébergeant les instances PostgreSQL à sauvegarder.

3.4.11 Barman - Configuration SSH



- Utilisateur `postgres` pour les serveurs PostgreSQL
- Utilisateur `barman` pour le serveur de sauvegardes
- Générer les clés SSH (RSA) des utilisateurs système `postgres` (serveurs PG) et `barman` (serveur barman)
- Échanger les clés SSH publiques entre les serveurs PostgreSQL et le serveur de sauvegarde
- Établir manuellement une première connexion SSH entre chaque machine
- Inutile si utilisation de `pg_basebackup / pg_recvwal`

Dans le cadre de la mise en place de sauvegardes avec un stockage des données sur un serveur tiers, la plupart des outils et méthodes historiques de sauvegardes reposent sur le protocole SSH et des outils tels que `rsync` pour assurer les transferts au travers du réseau.

Afin d'automatiser ces transferts via le protocole SSH, il est impératif d'autoriser l'authentification SSH par clé, et d'échanger les clés publiques entre les différents serveurs hébergeant les instances PostgreSQL et le serveur de sauvegarde.

3.4.12 Barman - Configuration PostgreSQL



- Adapter la configuration de l'archivage dans le fichier `postgresql.conf`:

```
wal_level = 'replica'  
archive_mode = on  
archive_command = 'barman-wal-archive backup-srv pgsql %p'
```

- ... ou paramétrier la réplication si utilisation de `pg_basebackup / pg_recvwal`

Le paramétrage de l'archivage des journaux de transactions reste classique. La directive `archive_command` doit faire appel directement à l'outil système en charge du transfert du fichier.

Le paramètre `archive_mode` peut prendre la valeur `always` pour permettre un archivage à partir des serveurs secondaires.

Depuis la version 2.6 de Barman, il est recommandé d'utiliser la commande `barman-wal-archive` intégrée (fournie par le paquet `barman-cli`) pour gérer l'archivage. Cette commande interagit directement avec Barman pour recevoir le fichier, écrire son contenu via `fsync` et l'envoyer dans le réper-

toire *incomming* adapté. Cela réduit donc le risque de corruption, perte de données ou simplement d'erreur de répertoire.

3.4.13 Barman - Configuration globale



- barman.conf

```
[barman]
barman_home = /var/lib/barman
barman_user = barman
log_file = /var/log/barman/barman.log
log_level = INFO
configuration_files_directory = /etc/barman.d
```

- **barman_home** : répertoire racine de travail de Barman, contenant les sauvegardes et les journaux de transactions archivés ;
- **barman_user** : utilisateur système ;
- **log_file** : fichier contenant les traces Barman ;
- **configuration_files_directory**: chemin vers le dossier d'inclusion des fichiers de configuration supplémentaires (défaut : /etc/barman.d) ;
- **log_level** : niveau de verbosité des traces, par défaut INFO.

3.4.14 Barman - Configuration sauvegardes



- Configuration globale des options de sauvegarde

```
compression = gzip
backup_compression = gzip
immediate_checkpoint = false
basebackup_retry_times = 0
basebackup_retry_sleep = 30
```

- **compression** : méthode de compression des journaux de transaction - sont disponibles : gzip, bzip2, custom, laissant la possibilité d'utiliser l'utilitaire de compression de son choix (défaut : gzip) ;
- **backup_compression** : méthode utilisée par Barman pour compresser la sauvegarde, les options disponibles dépendent de la version de PostgreSQL utilisée (la version 15 apporte beaucoup de nouveautés à ce niveau) ;

- **immediate_checkpoint**: force la création immédiate d'un checkpoint impliquant une augmentation des écritures, le but étant de débuter la sauvegarde le plus rapidement possible (défaut : off) ;
- **basebackup_retry_times** : nombre de tentative d'écriture d'un fichier - utile pour relancer la copie d'un fichier en cas d'échec sans compromettre le déroulement global de la sauvegarde ;
- **basebackup_retry_sleep** : spécifié en secondes, il s'agit ici de l'intervalle de temps entre deux tentatives de copie d'un fichier en cas d'échec.

3.4.15 Barman - Configuration réseau



- Possibilité de réduire la bande passante
- Et de compresser le trafic réseau
- Exemple

```
bandwidth_limit = 4000
network_compression = false
```

- **bandwidth_limit** : limitation de l'utilisation de la bande passante réseau lors du transfert de la sauvegarde, s'exprime en kbps (par défaut à 0, autrement dit pas de limitation) ;
- **network_compression** : activation de la compression à la volée des données lors du transfert réseau de la sauvegarde - utilisé à la sauvegarde ou lors d'une restauration (défaut : false).

3.4.16 Barman - Configuration rétention



- Configuration de la rétention en nombre de sauvegardes
- Ou en « fenêtre de restauration », en jours, semaines ou mois
- Déclenchement d'une erreur en cas de sauvegarde trop ancienne
- Exemple

```
minimum_redundancy = 5
retention_policy = RECOVERY WINDOW OF 7 DAYS
last_backup_maximum_age = 2 DAYS
```

- **minimum_redundancy** : nombre minimum de sauvegardes à conserver - si ce n'est pas respecté, Barman empêchera la suppression (défaut : 0) ;
- **retention_policy** : définit la politique de rétention en s'exprimant soit en nombre de sauvegarde via la syntaxe REDUNDANCY <valeur>, soit en fenêtre de restauration via la syntaxe

RECOVERY OF <valeur> {DAYS | WEEKS | MONTHS} (défaut : aucune rétention appliquée) ;

- **last_backup_maximum_age** : expression sous la forme <value> {DAYS | WEEKS | MONTHS}, définit l'âge maximal de la dernière sauvegarde - si celui-ci n'est pas respecté, lors de l'utilisation de la commande barman check, une erreur sera levée.

3.4.17 Barman - Configuration des hooks



- Lancer des scripts avant ou après les sauvegardes
- Et avant ou après le traitement du WAL archivé par Barman
- Exemple :

```
pre_backup_script = ...
post_backup_script = ...
pre_archive_script = ...
post_archive_script = ...
```

Barman offre la possibilité d'exécuter des commandes externes (scripts) avant et/ou après les opérations de sauvegarde et les opérations d'archivage des journaux de transaction.

Attention, la notion d'archivage de journal de transactions dans ce contexte ne concerne pas l'archivage réalisé depuis l'instance PostgreSQL, qui copie les WAL dans un répertoire <incoming> sur le serveur Barman, mais bien l'opération de récupération du WAL depuis ce répertoire <incoming>.

3.4.18 Barman - Configuration d'un dépôt synchronisé



- Copie à l'identique du dépôt d'origine
- Sur le dépôt à synchroniser :
 - primary_ssh_command
- Commandes :
 - barman sync-info --primary <instance> <ID-sauvegarde>
 - barman sync-backup <instance> <ID-sauvegarde>
 - barman sync-wal <instance>

Barman permet de créer une copie d'un dépôt barman pour répondre à des besoins de redondance géographique. Il suffit pour cela de valoriser le paramètre `primary_ssh_command` pour que le serveur barman client se connecte au serveur principal et duplique les sauvegardes et WAL.

La commande `barman sync-info --primary <instance> <ID-sauvegarde>` permet d'afficher les informations de synchronisation. Le processus de copie est lancé automatiquement par la tâche de maintenance automatisée. Il est aussi possible de lancer la synchronisation manuellement pour une sauvegarde en particulier avec `barman sync-backup <instance> <ID-sauvegarde>` ou pour les WAL avec `barman sync-wal <instance>`.

3.4.19 Barman - Configuration par instance



- `configuration_files_directory`
 - un fichier de configuration par instance
 - Ou une section par instance

Après avoir vu les options globales, nous allons voir à présent les options spécifiques à chaque instance à sauvegarder.

Afin de conserver une certaine souplesse dans la gestion de la configuration Barman, il est recommandé de paramétrier la directive `configuration_files_directory` de la section `[barman]` afin de pouvoir charger d'autres fichiers de configuration, permettant ainsi d'isoler la section spécifique à chaque instance à sauvegarder dans son propre fichier de configuration.

3.4.20 Barman - Exemple configuration par instance



- Section spécifique par instance
- Permet d'adapter la configuration aux différentes instances
- Exemple

```
[pgsrv]
description = "PostgreSQL Instance pgsrv"
ssh_command = ssh postgres@pgsrv
conninfo = host=pgsrv user=postgres dbname=postgres
backup_method = rsync
reuse_backup = link
backup_options = exclusive_backup
archiver = on
```

La première ligne définit le nom de la section. Ce nom est important et doit être significatif car il sera utilisé lors des tâches d'exploitation pour identifier l'instance cible.

L'idéal est d'utiliser le nom d'hôte ou l'adresse IP du serveur si celui-ci n'héberge qu'une seule instance.

- **description** : chaîne de caractère servant de descriptif de l'instance ;
- **ssh_command** : commande shell utilisée pour établir la connexion ssh vers le serveur hébergeant l'instance à sauvegarder ;
- **conninfo** : chaîne de connexion PostgreSQL.

Tous les autres paramètres, à l'exception de `log_file` et `log_level`, peuvent être redéfinis pour chaque instance.

3.4.21 Barman - Exemple configuration Streaming Only



```
[pgsrv]
description = "Sauvegarde de pgsrv via Streaming Replication"
conninfo = host=pgsrv user=barman dbname=postgres
streaming_conninfo = host=pgsrv user=streaming_barman
backup_method = postgres
streaming_archiver = on
create_slot = auto
slot_name = barman

- barman replication-status pgsrv
```

La commande `barman replication-status` permet d'afficher l'état de la réplication :

```
$ barman replication-status pgsrv
Status of streaming clients for server 'pgsrv':
  Current LSN on master: 0/140001B0
  Number of streaming clients: 1

1. Async WAL streamer
  Application name: barman_receive_wal
  Sync stage      : 3/3 Remote write
  Communication   : Unix domain socket
  User name       : barman
  Current state   : streaming (async)
  Replication slot: barman
  WAL sender PID  : 29439
  Started at      : 2022-10-17 14:54:02.122742+00:00
  Sent LSN        : 0/140001B0 (diff: 0 B)
  Write LSN       : 0/140001B0 (diff: 0 B)
  Flush LSN       : 0/14000000 (diff: -432 B)
```

3.4.22 Barman - Vérification de la configuration



- La commande `show-server` montre la configuration

```
$ sudo -u barman barman show-server {<instance> | all}
```

- La commande `check` effectue des tests pour la valider

```
$ sudo -u barman barman check {<instance> | all}  
$ sudo -u barman barman check {<instance> | all} --nagios
```

La commande `show-server` permet de visualiser la configuration de Barman pour l'instance spécifiée, ou pour toutes les instances si le mot-clé `all` est utilisé.

La commande `check` vérifie le bon paramétrage de Barman pour l'instance spécifiée, ou pour toutes les instances si le mot-clé `all` est utilisé.

Elle permet de s'assurer que les points clés sont fonctionnels, tels que l'accès SSH, l'archivage des journaux de transaction (`archive_command`, `archive_mode`...), la politique de rétention, la compression, etc.

Il est possible d'utiliser l'option `--nagios` qui permet de formater la sortie de la commande `check` et de l'utiliser en tant que sonde Nagios.

Exemple de sortie de la commande `show-server` :

```
$ barman show-server pgsrv  
Server pgsrv:  
  active: True  
  archive_command: None  
  archive_mode: None  
  archiver: True  
  archiver_batch_size: 0  
  backup_directory: /var/lib/barman/pgsrv  
  backup_method: rsync  
  backup_options: BackupOptions(['exclusive_backup'])  
  bandwidth_limit: None  
  barman_home: /var/lib/barman  
  barman_lock_directory: /var/lib/barman  
  basebackup_retry_sleep: 30  
  basebackup_retry_times: 0  
  basebackups_directory: /var/lib/barman/pgsrv/base  
  check_timeout: 30  
  compression: None  
  conninfo: host=pgsrv user=postgres dbname=postgres  
  create_slot: manual  
  current_xlog: None  
  custom_compression_filter: None  
  custom_decompression_filter: None
```

```
data_directory: None
description: PostgreSQL Instance pgsrv
disabled: False
errors_directory: /var/lib/barman/pgsrv/errors
immediate_checkpoint: False
incoming_wals_directory: /var/lib/barman/pgsrv/incoming
is_in_recovery: None
is_superuser: None
last_backup_maximum_age: None
max_incoming_wals_queue: None
minimum_redundancy: 0
msg_list: []
name: pgsrv
network_compression: False
parallel_jobs: 1
passive_node: False
path_prefix: None
pgespresso_installed: None
post_archive_retry_script: None
post_archive_script: None
post_backup_retry_script: None
post_backup_script: None
post_delete_retry_script: None
post_delete_script: None
post_recovery_retry_script: None
post_recovery_script: None
post_wal_delete_retry_script: None
post_wal_delete_script: None
postgres_systemid: None
pre_archive_retry_script: None
pre_archive_script: None
pre_backup_retry_script: None
pre_backup_script: None
pre_delete_retry_script: None
pre_delete_script: None
pre_recovery_retry_script: None
pre_recovery_script: None
pre_wal_delete_retry_script: None
pre_wal_delete_script: None
primary_ssh_command: None
recovery_options: RecoveryOptions([])
replication_slot: None
replication_slot_support: None
retention_policy: None
retention_policy_mode: auto
reuse_backup: link
server_txt_version: None
slot_name: None
ssh_command: ssh postgres@pgsrv
streaming_archiver: False
streaming_archiver_batch_size: 0
streaming_archiver_name: barman_receive_wal
streaming_backup_name: barman_streaming_backup
streaming_conninfo: host=pgsrv user=postgres dbname=postgres
streaming_wals_directory: /var/lib/barman/pgsrv/streaming
synchronous_standby_names: None
```

```
tablespace_bandwidth_limit: None
wal_retention_policy: main
wals_directory: /var/lib/barman/pgsrv/wals
```

Exemple de sortie de la commande check :

```
$ barman check pgsql
Server pgsql:
  PostgreSQL: OK
  superuser or standard user with backup privileges: OK
  PostgreSQL streaming: OK
  wal_level: OK
  replication slot: OK
  directories: OK
  retention policy settings: OK
  backup maximum age: OK (no last_backup_maximum_age provided)
  backup minimum size: OK (33.6 MiB)
  wal maximum age: OK (no last_wal_maximum_age provided)
  wal size: OK (0 B)
  compression settings: OK
  failed backups: OK (there are 0 failed backups)
  minimum redundancy requirements: OK (have 2 backups, expected at least 0)
  pg_basebackup: OK
  pg_basebackup compatible: OK
  pg_basebackup supports tablespaces mapping: OK
  systemid coherence: OK
  pg_receivexlog: OK
  pg_receivexlog compatible: OK
  receive-wal running: OK
  archiver errors: OK
```

3.4.23 Barman - Statut



- La commande status affiche des informations détaillées
 - sur la configuration Barman
 - sur l'instance spécifiée
- Exemple

```
$ sudo -u barman barman status {<instance> | all}
```

La commande status retourne de manière détaillée le statut de l'instance spécifiée, ou de toutes si le mot-clé all est utilisé.

Les informations renvoyées sont, entre autres :

- la description extraite du fichier de configuration de Barman ;

- la version de PostgreSQL ;
- si l'extension pgespresso est utilisée ;
- l'emplacement des données sur l'instance (PGDATA) ;
- la valeur de l'archive_command ;
- des informations sur les journaux de transactions :

- position courante
 - dernier segment archivé
-
- des informations sur les sauvegardes :

- nombre de sauvegarde
- ID de la première sauvegarde
- ID de la dernière sauvegarde
- politique de rétention

Exemple de sortie de la commande :

```
$ barman status pgsql
Server pgsql:
  Description: PostgreSQL Instance pgsql
  Active: True
  Disabled: False
  PostgreSQL version: 12.1
  Cluster state: in production
  pgespresso extension: Not available
  Current data size: 24.4 MiB
  PostgreSQL Data directory: /var/lib/pgsql/12/data
  Current WAL segment: 000000010000000000000004
  PostgreSQL 'archive_command' setting: barman-wal-archive localhost pgsql %p
  Last archived WAL: 000000010000000000000003, at Wed Dec 11 11:44:12 2019
  Failures of WAL archiver: 52 (000000010000000000000001 at Wed Dec 11 11:44:04 2019)
  Server WAL archiving rate: 1.41/hour
  Passive node: False
  Retention policies: not enforced
  No. of available backups: 0
  First available backup: None
  Last available backup: None
  Minimum redundancy requirements: satisfied (0/0)
```

3.4.24 Barman - Diagnostiquer



- La commande `diagnose` renvoie
 - les informations renvoyées par la commande `status`
 - des informations supplémentaires (sur le système par exemple)
 - au format `json`
- Exemple

```
$ sudo -u barman barman diagnose
```

La commande `diagnose` retourne les informations importantes concernant toutes les instances à sauvegarder, en donnant par exemple les versions de chacun des composants utilisés.

Elle reprend également les informations rentrées par la commande `status`, le tout au format JSON.

3.4.25 Barman - Nouvelle sauvegarde



- Pour déclencher une nouvelle sauvegarde
- ```
$ sudo -u barman barman backup {<instance> | all} [--wait]
```
- Le détail de sauvegarde effectuée est affiché en sortie

La commande `backup` lance immédiatement une nouvelle sauvegarde, pour une seule instance si un identifiant est passé en argument, ou pour toutes les instances configurées si le mot-clé `all` est utilisé.

L'option `--wait` permet d'attendre que les WAL soient archivés avant de rendre la main.

Exemple de sortie de la commande :

```
$ barman backup pgsql
Starting backup using rsync-exclusive method for server pgsql in
/var/lib/barman/pgsql/base/20191211T121244
Backup start at LSN: 0/5000028 (000000010000000000000005, 00000028)
This is the first backup for server pgsql
WAL segments preceding the current backup have been found:
000000010000000000000001 from server pgsql has been removed
```

```
000000010000000000000002 from server pgsql has been removed
000000010000000000000003 from server pgsql has been removed
Starting backup copy via rsync/SSH for 20191211T121244
Copy done (time: 1 second)
This is the first backup for server pgsql
Asking PostgreSQL server to finalize the backup.
Backup size: 24.3 MiB. Actual size on disk: 24.3 MiB (-0.00% deduplication ratio).
Backup end at LSN: 0/5000138 (000000010000000000000005, 00000138)
Backup completed (start time: 2019-12-11 12:12:44.788598, elapsed time: 5 seconds)
Processing xlog segments from file archival for pgsql
000000010000000000000004
000000010000000000000005
000000010000000000000005.00000028.backup
```

### 3.4.26 Barman - Lister les sauvegardes



- Pour lister les sauvegardes existantes

```
$ sudo -u barman barman list-backup {<instance> | all}
```

- Affiche notamment la taille de la sauvegarde et des WAL associés

Liste les sauvegardes du catalogue, soit par instance, soit toutes si le mot-clé all est passé en argument.

Exemple de sortie de la commande :

```
$ barman list-backup pgsql
pgsql 20191211T121244 - Wed Dec 11 12:12:47 2019 - Size: 40.3 MiB -
 WAL Size: 0 B
```

### 3.4.27 Barman - Détail d'une sauvegarde



- show-backup affiche le détail d'une sauvegarde (taille...)

```
$ sudo -u barman barman show-backup <instance> <ID-sauvegarde>
```

- list-files affiche le détail des fichiers d'une sauvegarde

```
$ sudo -u barman barman list-files <instance> <ID-sauvegarde>
```

La commande `show-backup` affiche toutes les informations relatives à une sauvegarde en particulier, comme l'espace disque occupé, le nombre de journaux de transactions associés, etc.

La commande `list-files` permet quant à elle d'afficher la liste complète des fichiers contenus dans la sauvegarde.

Exemple de sortie de la commande `show-backup`:

```
$ barman show-backup pgsql 20191211T121244
Backup 20191211T121244:
 Server Name : pgsql
 System Id : 6769104211696624889
 Status : DONE
 PostgreSQL Version : 120001
 PGDATA directory : /var/lib/pgsql/12/data

 Base backup information:
 Disk usage : 24.3 MiB (40.3 MiB with WALs)
 Incremental size : 24.3 MiB (-0.00%)
 Timeline : 1
 Begin WAL : 0000000100000000000000000005
 End WAL : 0000000100000000000000000005
 WAL number : 1
 Begin time : 2019-12-11 12:12:44.526305+01:00
 End time : 2019-12-11 12:12:47.794687+01:00
 Copy time : 1 second + 1 second startup
 Estimated throughput : 14.3 MiB/s
 Begin Offset : 40
 End Offset : 312
 Begin LSN : 0/5000028
 End LSN : 0/5000138

 WAL information:
 No of files : 0
 Disk usage : 0 B
 Last available : 0000000100000000000000000005

 Catalog information:
 Retention Policy : not enforced
 Previous Backup : - (this is the oldest base backup)
 Next Backup : - (this is the latest base backup)
```

### 3.4.28 Barman - Suppression d'une sauvegarde



- Pour supprimer manuellement une sauvegarde

```
$ sudo -u barman barman delete <instance> <ID-sauvegarde>
```

- Renvoie une erreur si la redondance minimale ne le permet pas

La suppression d'une sauvegarde nécessite de spécifier l'instance ciblée et l'identifiant de la sauvegarde à supprimer.

Cet identifiant peut être trouvé en utilisant la commande Barman `list-backup`.

Si le nombre de sauvegardes (après suppression) ne devait pas respecter le seuil défini par la directive `minimum_redundancy`, la suppression ne sera alors pas possible.

### 3.4.29 Barman - Conserver une sauvegarde



- Pour conserver une sauvegarde

```
$ sudo -u barman barman keep <instance> <ID-sauvegarde>
```

- Pour relâcher une sauvegarde

```
$ sudo -u barman barman keep --release <instance> <ID-sauvegarde>
```

Il est possible de marquer une sauvegarde pour qu'elle soit conservée par barman quelle que soit la rétention configurée avec la commande `barman keep <instance> <ID-sauvegarde>`.

La sauvegarde peut être relâchée en ajoutant le paramètre `--release`.

### 3.4.30 Barman - Tâches de maintenance



- La commande Barman `cron` déclenche la maintenance

- récupération des WAL archivés
- compression
- politique de rétention
- démarrage de `pg_receivewal`

- Exemple

```
$ sudo -u barman barman cron
```

- À planifier ! (vérifier `/etc/cron.d/barman`)

La commande `cron` permet d'exécuter les tâches de maintenance qui doivent être exécutées pér-

riodiquement, telles que l'archivage des journaux de transactions (déplacement du dossier `incoming_wals/` vers `wals/`), ou la compression.

L'application de la politique de rétention est également faite dans ce cadre.

Le démarrage de la commande `pg_recvwal` est aussi gérée par ce biais.

L'exécution de cette commande doit donc être planifiée via votre ordonnanceur préféré (`cron` d'Unix par exemple), par exemple toutes les minutes.

Si vous avez installé Barman via les paquets (rpm ou debian), une tâche `cron` exécutée toutes les minutes a été créée automatiquement.

### 3.4.31 Barman - Restauration



- Copie/transfert de la sauvegarde
- Copie/transfert des journaux de transactions
- Génération du fichier `recovery.conf`
- Copie/transfert des fichiers de configuration

Le processus de restauration géré par Barman reste classique, mais nécessite tout de même quelques points d'attention.

En particulier, les fichiers de configuration sauvegardés sont restaurés dans le dossier `$PGDATA`, ce n'est potentiellement pas le bon emplacement selon le type d'installation / configuration de l'instance. Dans une installation basée sur les paquets Debian/Ubuntu par exemple, les fichiers de configuration se trouvent dans `/etc/postgresql/<version>/<instance>` et non dans le répertoire `PGDATA`. Il convient donc de penser à les supprimer du `PGDATA` s'ils n'ont rien à y faire avant de démarrer l'instance.

De même, la directive de configuration `archive_command` est passée à `false` par Barman. Une fois l'instance démarrée et fonctionnelle, il convient de modifier la valeur de ce paramètre pour réactiver l'archivage des journaux de transactions.

### 3.4.32 Barman - Options de restauration



- Locale ou à distance
- Cibles : timeline, date, ID de transaction ou point de restauration
- Déplacement des tablespaces

Au niveau de la restauration, Barman offre la possibilité de restaurer soit en local (sur le serveur où se trouvent les sauvegardes), soit à distance.

Le cas le plus commun est une restauration à distance, car les sauvegardes sont généralement centralisées sur le serveur de sauvegarde d'où Barman est exécuté.

Pour la restauration à distance, Barman s'appuie sur la couche SSH pour le transfert des données.

Barman supporte différents types de cibles dans le temps pour la restauration :

- **timeline** : via l'option `--target-tli`, lorsqu'une divergence de timeline a eu lieu, il est possible de restaurer et rejouer toutes les transactions d'une timeline particulière ;
- **date** : via l'option `--target-time` au format **YYYY-MM-DD HH:MM:SS.mmm**, spécifie une date limite précise dans le temps au delà de laquelle la procédure de restauration arrête de rejouer les transactions ;
- **identifiant de transaction** : via l'option `--target-xid`, restauration jusqu'à une transaction précise ;
- **point de restauration** : via l'option `--target-name`, restauration jusqu'à un point de restauration créé préalablement sur l'instance via l'appel à la fonction `pg_create_restore_point(nom)`.

Barman permet également de relocaliser un tablespace lors de la restauration.

Ceci est utile lorsque l'on souhaite restaurer une sauvegarde sur un serveur différent, ne disposant pas des mêmes points de montage des volumes que l'instance originelle.

### 3.4.33 Barman - Exemple de restauration à distance



- Exemple d'une restauration
  - déclenchée depuis le serveur Barman
  - avec un point dans le temps spécifié

```
$ sudo -u barman barman recover
 --remote-ssh-command "ssh postgres@pgsrv"
 --target-time "2019-12-11 14:00:00"
 pgsql 20191211T121244 /var/lib/pgsql/12/data/
```

Dans cet exemple, nous souhaitons effectuer une restauration à distance via l'option `--remote-ssh-command`, prenant en argument "`ssh postgres@pgsrv`" correspondant à la commande SSH pour se connecter au serveur à restaurer.

L'option `--target-time` définit ici le point de restauration dans le temps comme étant la date « `2019-12-11 14:00:00` ».

Les trois derniers arguments sont :

- l'identifiant de l'instance dans le fichier de configuration de Barman : pgsql ;
- l'identifiant de la sauvegarde cible : 20191211T121244 ;
- et enfin le dossier PGDATA de l'instance à restaurer.

### 3.5 PITRERY - PRÉSENTATION GÉNÉRALE



- R&D Dalibo
- Langage : **bash**
- OS : **Unix/Linux**
- Versions compatibles : **8.2 à 14** (pas 15+)
- Développement arrêté, ne plus utiliser

pitrery est un outil de gestion de sauvegarde physique et restauration PITR, écrit en bash, issu du labo R&D de Dalibo.

Il est compatible avec tous les environnements Unix/Linux disposant de l'interpréteur de shell bash, et supporte toutes les versions de PostgreSQL depuis la 8.2 jusqu'à la version 14, qui est la dernière version supportée.



Après 10 ans de développement actif, le projet Pitrery est désormais placé en maintenance LTS (*Long Term Support*) jusqu'en novembre 2026. Plus aucune nouvelle fonctionnalité n'y sera ajoutée, les mises à jour concerneront les correctifs de sécurité uniquement. Il est désormais conseillé de lui préférer pgBackRest. Il n'est plus compatible avec PostgreSQL 15 et supérieur.

Site Web de pitrery<sup>13</sup>.

---

<sup>13</sup><https://dalibo.github.io/pitrery/>

## 3.6 AUTRES OUTILS DE L'ÉCOSYSTÈME



- De nombreux autres outils existent
  - ...ou ont existé
- WAL-E, OmniPITR, pg\_rman, walmgr, pg\_rman...
- WAL-G

Du fait du dynamisme du projet, l'écosystème des outils autour de PostgreSQL est très changeant.

À côté des outils évoqués ci-dessus, que nous recommandons, on trouve de nombreux projets autour du thème de la gestion des sauvegardes.

Certains de ces projets répondent à des problématiques spécifiques, d'autres sont assez anciens et plus guère maintenus (comme WAL-E<sup>14</sup>), rendus inutiles par l'évolution de PostgreSQL ces dernières années (comme walmgr, de la suite SkyTools<sup>15</sup>, ou OmniPITR<sup>16</sup>) ou simplement peu actifs et peu rencontrés en production (par exemple pg\_rman<sup>17</sup>, développé par NTT).

Le plus intéressant et actif est sans doute WAL-G.

### 3.6.1 WAL-G - présentation



- Successeur de WAL-E, par Citus Data & Yandex
- Orientation cloud
- Aussi pour MySQL et SQL Server

WAL-G<sup>18</sup> est une réécriture d'un ancien outil assez populaire, WAL-E, par Citus<sup>19</sup> et Yandex, et actif.

De par sa conception, il est optimisé pour l'archivage des journaux de transactions vers des stockages *cloud* (Amazon S3, Google, Yandex), la compression multi-processeurs par différents algorithmes<sup>20</sup> et l'optimisation du temps de restauration. Il supporte aussi MySQL et SQL Server (et d'autres dans le futur).

<sup>14</sup><https://github.com/wal-e/wal-e>

<sup>15</sup><https://wiki.postgresql.org/wiki/SkyTools>

<sup>16</sup><https://github.com/omniti-labs/omnipitr>

<sup>17</sup>[https://github.com/ossc-db/pg\\_rman](https://github.com/ossc-db/pg_rman)

<sup>18</sup><https://github.com/wal-g/wal-g>

<sup>19</sup><https://www.citusdata.com/blog/2017/08/18/introducing-wal-g-faster-restores-for-postgres/>

<sup>20</sup><https://wal-g.readthedocs.io/>

### 3.7 CONCLUSION



- Des outils pour vous aider !
- Pratiquer, pratiquer et pratiquer
- Superviser les sauvegardes !

Nous venons de vous présenter des outils qui vont vous permettre de vous simplifier la tâche dans la mise en place d'une solution de sauvegarde fiable et robuste de vos instances PostgreSQL.

Cependant, leur maîtrise passera par de la pratique, et en particulier, la pratique de la restauration.

Le jour où la restauration d'une instance de production se présente, ce n'est généralement pas une situation confortable à cause du stress lié à une perte/corruption de données, interruption du service, etc. Autant maîtriser les outils qui vous permettront de sortir de ce mauvais pas.

N'oubliez pas également l'importance de la supervision des sauvegardes !

### 3.8 QUIZ



[https://dali.bo/i4\\_quiz](https://dali.bo/i4_quiz)

## 3.9 TRAVAUX PRATIQUES

### 3.9.1 Utilisation de pgBackRest (Optionnel)



**But :** Sauvegarder et restaurer avec pgBackRest

Installer pgBackRest à partir des paquets du PGDG.

En vous aidant de <https://pgbackrest.org/user-guide.html#quickstart> : - configurer pgBackRest pour sauvegarder le serveur PostgreSQL en local dans /var/lib/pgsql/1/backups ; - le nom de la stanza sera **instance\_dev** ; - prévoir de ne conserver qu'une seule sauvegarde complète.

Configurer l'archivage des journaux de transactions de PostgreSQL avec pgBackRest.

Initialiser le répertoire de stockage des sauvegardes et vérifier la configuration de l'archivage.

Lancer une sauvegarde complète. Afficher les détails de cette sauvegarde.

Ajouter des données :

- ajouter une table avec 1 million de lignes ;
- forcer la rotation du journal de transaction courant (pg\_switch\_wal) pour s'assurer que les dernières modifications sont archivées ;
- vérifier que le journal concerné est bien dans les archives.

Simulation d'un incident : supprimer tout le contenu de la table.

Restaurer les données avant l'incident à l'aide de pgBackRest.

### 3.9.2 Utilisation de barman (Optionnel)



**But :** Sauvegarder et restaurer avec barman

### 3.9.3 Utilisation de barman (Optionnel)

Installer barman depuis les dépôts communautaires (la documentation est sur <https://www.pgbarman.org/documentation/>).

Configurer barman pour la sauvegarde du serveur via Streaming Replication (pg\_basebackup et pg\_receivewal).

Vérifier que l'archivage fonctionne et que la configuration de barman est correcte.

Faire une sauvegarde.

Ajouter des données :

- ajouter une table avec 1 million de lignes ;
- forcer la rotation du journal de transaction courant pour garantir que les dernières modifications sont archivées.

Vérifier que le journal concerné est bien dans les archives.

Lister les sauvegardes.

Afficher les informations sur la sauvegarde.

Simulation d'un incident : supprimer tout le contenu de la table.

Restaurer les données avant l'incident à l'aide de barman.

## 3.10 TRAVAUX PRATIQUES (SOLUTIONS)

### 3.10.1 Utilisation de pgBackRest (Optionnel)

Installer pgBackRest à partir des paquets du PGDG.

L'installation du paquet est triviale :

```
yum install pgbackrest # CentOS 7
dnf install pgbackrest # Rocky Linux
```

En vous aidant de <https://pgbackrest.org/user-guide.html#quickstart>, configurer pgBackRest pour sauvegarder le serveur PostgreSQL en local dans /var/lib/pgsql/14/backups. Le nom de la stanza sera **instance\_dev**. Ne conserver qu'une seule sauvegarde complète.

Le fichier de configuration est /etc/pgbackrest.conf :

```
[global]
repo1-path=/var/lib/pgsql/14/backups
repo1-retention-full=1

[instance_dev]
pg1-path=/var/lib/pgsql/14/data
```

Configurer l'archivage des journaux de transactions de PostgreSQL avec pgBackRest.

```
wal_level = replica
archive_mode = on
archive_command = 'pgbackrest --stanza=instance_dev archive-push %p'
```

Redémarrer PostgreSQL.

Initialiser le répertoire de stockage des sauvegardes et vérifier la configuration de l'archivage.

Sous l'utilisateur **postgres** :

```
$ pgbackrest --stanza=instance_dev --log-level-console=info stanza-create
```

Vérifier la configuration de pgBackRest et de l'archivage :

```
$ pgbackrest --stanza=instance_dev --log-level-console=info check
```

Vérifier que l'archivage fonctionne :

```
$ ls /var/lib/pgsql/14/backups/archive/instance_dev/14-1/0000000100000000/
```

```
SELECT * FROM pg_stat_archiver;
```

Lancer une sauvegarde complète. Afficher les détails de cette sauvegarde.

```
$ pgbackrest --stanza=instance_dev --type=full \
--log-level-console=info backup | grep P00
```

```
P00 INFO: backup command begin 2.19: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/data --repo1-path=/var/lib/pgsql/14/backups
--repo1-retention-full=1 --stanza=instance_dev --type=full
P00 INFO: execute non-exclusive pg_start_backup() with label
"pgBackRest backup started at 2021-11-26 12:25:32":
backup begins after the next regular checkpoint completes
P00 INFO: backup start archive = 0000000100000000000000000003, lsn = 0/3000060
2P00 INFO: full backup size = 24.2MB
P00 INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments
 to archive
P00 INFO: backup stop archive = 0000000100000000000000000003, lsn = 0/3000138
P00 INFO: new backup label = 20211126-122532F
P00 INFO: backup command end: completed successfully (8694ms)
P00 INFO: expire command begin 2.19: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/data --repo1-path=/var/lib/pgsql/14/backups
--repo1-retention-full=1 --stanza=instance_dev --type=full
P00 INFO: expire command end: completed successfully (8ms)
```

Lister les sauvegardes :

```
$ pgbackrest --stanza=instance_dev info
stanza: instance_dev
 status: ok
 cipher: none

 db (current)
 wal archive min/max (14-1): 000000010000000000000003/000000010000000000000003

 full backup: 20211126-122532F
 timestamp start/stop: 2021-11-26 12:25:32 / 2021-11-26 12:25:41
 wal start/stop: 000000010000000000000003 / 000000010000000000000003
 database size: 24.2MB, backup size: 24.2MB
 repository size: 2.9MB, repository backup size: 2.9MB
```

**Ajouter des données :** Ajouter une table avec 1 million de lignes. Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées. Vérifier que le journal concerné est bien dans les archives.

```
CREATE TABLE matable AS SELECT i FROM generate_series(1,1000000) i ;
SELECT 1000000
```

Forcer la rotation du journal :

```
SELECT pg_switch_wal();
```

Vérifier que le journal concerné est bien dans les archives.

**Simulation d'un incident :** supprimer tout le contenu de la table.

```
TRUNCATE TABLE matable;
```

**Restaurer les données avant l'incident à l'aide de pgBackRest.**

D'abord, stopper PostgreSQL.

Lancer la commande de restauration :

```
$ pgbackrest --stanza=instance_dev --log-level-console=info \
--delta \
--target="2021-11-26 12:30:15" \
--target-action=promote \
--type=time \
--target-exclusive \
restore |grep P00

P00 INFO: restore command begin 2.19: --delta --log-level-console=info
--pg1-path=/var/lib/pgsql/14/data --repo1-path=/var/lib/pgsql/14/backups
--stanza=instance_dev --target="2021-11-26 12:30:15" --target-action=promote
--target-exclusive --type=time
P00 INFO: restore backup set 20211126-122532F
P00 INFO: remove invalid files/links/paths from '/var/lib/pgsql/14/data'
P00 INFO: write updated /var/lib/pgsql/14/data/postgresql.auto.conf
P00 INFO: restore global/pg_control
(performed last to ensure aborted restores cannot be started)
P00 INFO: restore command end: completed successfully (501ms)
```

Démarrer PostgreSQL.

Vérifier les logs et la présence de la table disparue.

```
SELECT count(*) FROM matable ;

count

1000000
```

#### Remarque :

Sans spécifier de --target-action=promote, on obtiendrait dans les traces de PostgreSQL, après restore :

```
LOG: recovery has paused
HINT: Execute pg_wal_replay_resume() to continue.
```

### 3.10.2 Utilisation de barman (Optionnel)

Installer barman depuis les dépôts communautaires (la documentation est sur <https://www.pgbarman.org/documentation/>).

**Pré-requis :** sous CentOS 7, le dépôt EPEL est nécessaire à cause des dépendances python, s'il n'est pas déjà installé :

```
yum install epel-release
```

La commande suivante suffit pour installer l'outil et ses dépendances.

```
yum install barman # CentOS 7
dnf install barman # Rocky Linux 8
```

Le paquet crée un utilisateur barman qui exécutera la sauvegarde et sera leur propriétaire. L'outil barman sera à exécuter uniquement avec cet utilisateur.

Configurer barman pour la sauvegarde du serveur via Streaming Replication (pg\_basebackup et pg\_receivewal).

/etc/barman.conf doit contenir :

```
[barman]
barman_user = barman
configuration_files_directory = /etc/barman.d
barman_home = /var/lib/barman
log_file = /var/log/barman/barman.log
log_level = INFO
compression = gzip
immediate_checkpoint = true
path_prefix = "/usr/pgsql-14/bin"
```

Ce fichier indique que l'utilisateur système est l'utilisateur barman. Les sauvegardes et journaux de transactions archivés seront placés dans /var/lib/barman.

Puis, il faut créer un fichier par hôte (uniquement localhost ici) et le placer dans le répertoire pointé par la variable configuration\_files\_directory (/etc/barman.d ici). On y indiquera les chaînes de connexion PostgreSQL pour la maintenance ainsi que pour la réPLICATION.

Dans /etc/barman.d/, créez un fichier nommé localhost.conf contenant ceci (vous pouvez repartir d'un modèle existant dans ce répertoire) :

```
[localhost]
description = "Sauvegarde de localhost via Streaming Replication"
conninfo = host=localhost user=barman dbname=postgres
streaming_conninfo = host=localhost user=streaming_barman
backup_method = postgres
streaming_archiver = on
slot_name = barman
```

Il faut donc d'abord créer les utilisateurs qui serviront aux connections :

```
postgres$ createuser --superuser --pwprompt barman
postgres$ createuser --replication --pwprompt streaming_barman
```

Ensuite, il faut s'assurer que ces utilisateurs puissent se connecter sur l'instance PostgreSQL, en modifiant pg\_hba.conf et peut-être postgresql.conf.

```
local all barman md5
host all barman 127.0.0.1/32 md5
host all barman ::1/128 md5
local replication streaming_barman md5
host replication streaming_barman 127.0.0.1/32 md5
host replication streaming_barman ::1/128 md5
```

Recharger la configuration (voire redémarrer PostgreSQL si nécessaire).

Configurer les droits du fichier ~/.pgpass de l'utilisateur système **barman** et ses droits d'accès comme suit :

```
barman$ chmod 600 ~/.pgpass
barman$ cat ~/.pgpass
```

```
:::barman:barmanpwd
:::streaming_barman:barmanpwd
```

Vérifier maintenant que les utilisateurs peuvent bien se connecter :

```
barman$ psql -c 'SELECT version()' -U barman -h localhost postgres
version

PostgreSQL 14.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 ...
barman$ psql -U streaming_barman -h localhost -c "IDENTIFY_SYSTEM" replication=1
systemid | timeline | xlogpos | dbname
-----+-----+-----+-----+
6769169214324921667 | 1 | 0/169E438 |
```

Afin d'éviter que le serveur principal ne recycle les journaux que nous souhaitons archiver via le protocole de réPLICATION (et pg\_receivewal), créer le slot de réPLICATION mentionné dans le fichier de configuration localhost.conf :

```
barman$ barman receive-wal --create-slot localhost
Creating physical replication slot 'barman' on server 'localhost'
Replication slot 'barman' created
```

**Vérifier que l'archivage fonctionne et que la configuration de barman est correcte.**

Après 1 minute (laissant à la tâche cron le soin de démarrer les processus adéquats), vérifier que l'archivage fonctionne :

```
$ ps -ef |grep streaming_barman
barman 10248 10244 0 14:55 ? 00:00:00 /usr/pgsql-14/bin/pg_receivewal
--dbname=dbname=replication host=localhost
options=cdatestyle=iso replication=true user=streaming_barman
application_name=barman_receive_wal
--verbose --no-loop --no-password
--directory=/var/lib/barman/localhost/streaming --slot=barman

postgres 10249 9575 0 14:55 ? 00:00:00 postgres: walsender
streaming_barman ::1(49182) streaming 0/169E438
```

On constate bien ici les 2 processus pg\_receivewal ainsi que walsender.

On peut également forcer la génération d'une nouvelle archive :

```
barman$ barman switch-wal localhost --force --archive
The WAL file 0000000100000000000000000001 has been closed on server 'localhost'
Waiting for the WAL file 0000000100000000000000000001 from server 'localhost'
Processing xlog segments from streaming for localhost
0000000100000000000000000001
```

Vérifier que la configuration de barman est correcte avec la commande suivante :

```
barman$ barman check localhost
```

```
Server localhost:
PostgreSQL: OK
```

```
is_superuser: OK
PostgreSQL streaming: OK
wal_level: OK
replication slot: OK
directories: OK
retention policy settings: OK
backup maximum age: OK (no last_backup_maximum_age provided)
compression settings: OK
failed backups: OK (there are 0 failed backups)
minimum redundancy requirements: OK (have 0 backups, expected at least 0)
pg_basebackup: OK
pg_basebackup compatible: OK
pg_basebackup supports tablespaces mapping: OK
systemid coherence: OK (no system Id stored on disk)
pg_receivexlog: OK
pg_receivexlog compatible: OK
receive-wal running: OK
archiver errors: OK
```

| Faire une sauvegarde.

```
barman$ barman backup localhost --wait

Starting backup using postgres method for server localhost in
/var/lib/barman/localhost/base/20211111T153507
Backup start at LSN: 0/40000C8 (000000100000000000000004, 000000C8)
Starting backup copy via pg_basebackup for 20211111T153507
Copy done (time: 1 second)
Finalising the backup.
This is the first backup for server localhost
WAL segments preceding the current backup have been found:
 000000010000000000000003 from server localhost has been removed
Backup size: 24.2 MiB
Backup end at LSN: 0/6000000 (000000100000000000000005, 00000000)
Backup completed (start time: 2021-11-11 15:35:07.610047, elapsed time: 2 seconds)
Waiting for the WAL file 000000010000000000000005 from server 'localhost'
Processing xlog segments from streaming for localhost
 000000010000000000000004
Processing xlog segments from streaming for localhost
 000000010000000000000005
```

| Ajouter des données : Ajouter une table avec 1 million de lignes. Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées.

```
CREATE TABLE matable AS SELECT i FROM generate_series(1,1000000) i;
```

Forcer la rotation du journal :

```
SELECT pg_switch_wal();
```

| Vérifier que le journal concerné est bien dans les archives.

Le processus pg\_receivewal récupère en flux continu les journaux de transactions de l'instance principale dans un fichier .partial, présent dans le répertoire <barman\_home>/<instance>/streaming.

Lors d'une rotation de journal, le fichier est déplacé de façon asynchrone dans le répertoire correspondant au segment auquel il appartient.

```
barman$ find /var/lib/barman/localhost/{streaming,wals} -type f

/var/lib/barman/localhost/streaming/00000001000000000000000A.partial
/var/lib/barman/localhost/wals/xlog.db
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000003
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000004
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000005
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000006
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000007
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000008
/var/lib/barman/localhost/wals/0000000100000000/000000010000000000000009
```

#### Lister les sauvegardes.

```
barman$ barman list-backup localhost
```

```
localhost 20211111T153507 - Wed Nov 11 15:35:09 2021 - Size: 24.2 MiB -
WAL Size: 12.5 MiB
```

#### Afficher les informations sur la sauvegarde.

```
barman$ barman show-backup localhost 20211111T153507
```

```
Backup 20211111T153507:
 Server Name : localhost
 System Id : 6769169214324921667
 Status : DONE
 PostgreSQL Version : 140001
 PGDATA directory : /var/lib/pgsql/14/data

 Base backup information:
 Disk usage : 24.2 MiB (24.2 MiB with WALs)
 Incremental size : 24.2 MiB (-0.00%)
 Timeline : 1
 Begin WAL : 000000010000000000000005
 End WAL : 000000010000000000000005
 WAL number : 1
 WAL compression ratio: 99.90%
 Begin time : 2021-11-11 15:35:08+01:00
 End time : 2021-11-11 15:35:09.509201+01:00
 Copy time : 1 second
 Estimated throughput : 12.8 MiB/s
 Begin Offset : 40
 End Offset : 0
 Begin LSN : 0/5000028
 End LSN : 0/6000000

 WAL information:
 No of files : 4
 Disk usage : 12.5 MiB
 WAL rate : 266.82/hour
 Compression ratio : 80.42%
```

```
Last available : 000000010000000000000009

Catalog information:
 Retention Policy : not enforced
 Previous Backup : - (this is the oldest base backup)
 Next Backup : - (this is the latest base backup)
```

| Simulation d'un incident : supprimer tout le contenu de la table.

```
TRUNCATE TABLE matable;
```

| Restaurer les données avant l'incident à l'aide de barman.

Arrêter l'instance PostgreSQL. Pour le TP, on peut renommer le PGDATA mais il n'est pas nécessaire de le supprimer vous-même.

Il faut savoir que --remote-ssh-command est nécessaire, sinon barman tentera de restaurer un PGDATA sur son serveur et avec ses droits.

Pour éviter de devoir configurer la connexion SSH, nous pouvons autoriser l'utilisateur système barman à faire des modifications dans le répertoire /var/lib/pgsql/14. Par exemple :

```
chmod 777 /var/lib/pgsql/
chmod 777 /var/lib/pgsql/14
```

Lancer la commande de restauration en tant que **barman** :

```
barman$ barman recover \
--target-time "20211111 15:40:00" \
--target-action "promote" \
localhost 20211111T153507 /var/lib/pgsql/14/data

Starting local restore for server localhost using backup 20211111T153507
Destination directory: /var/lib/pgsql/14/data
Doing PITR. Recovery target time: '2021-11-11 15:40:00+01:00'
Copying the base backup.
Copying required WAL segments.
Generating recovery configuration
Identify dangerous settings in destination directory.
Recovery completed (start time: 2021-11-11 15:59:13.697531, elapsed time: 1 second)
Your PostgreSQL server has been successfully prepared for recovery!
```

Rétablissement les droits sur le répertoire nouvellement créé par barman :

```
chown -R postgres: /var/lib/pgsql/14/data
```

Démarrer PostgreSQL.

Vérifier les logs et la présence de la table disparue.

```
$ cat /var/lib/pgsql/14/data/log/postgresql-Wed.log

[...]
2021-11-11 16:01:21.699 CET [28525] LOG: redo done at 0/9D49D68
2021-11-11 16:01:21.699 CET [28525] LOG: last completed transaction was
```

```
at log time 2021-11-11 15:36:08.184735+01
2021-11-11 16:01:21.711 CET [28525] LOG: restored log file
 "000000010000000000000009" from archive
2021-11-11 16:01:21.777 CET [28525] LOG: selected new timeline ID: 2
2021-11-11 16:01:21.855 CET [28525] LOG: archive recovery complete
2021-11-11 16:01:22.043 CET [28522] LOG: database system is ready to
 accept connections
```

```
SELECT count(*) FROM matable ;

count

1000000
```

Avant de passer à la suite de la formation, pour stopper les commandes démarrées par barman cron :

```
barman$ barman receive-wal --stop localhost
```

Il est possible de vérifier la liste des serveurs sur lesquels appliquer cette modification à l'aide de la commande barman list-server.

Pour désactiver totalement barman :

```
$ mv /etc/barman.d/localhost.conf /etc/barman.d/localhost.conf.old
$ sudo -iu barman barman cron
```



## 4/ Solutions de réPLICATION



Source de la photo : epSos.de<sup>1</sup> via Wikimedia<sup>2</sup>, licence CC-BY-2.0.

<sup>1</sup><https://www.flickr.com/photos/36495803@N05/3574411866/>

<sup>2</sup>[https://commons.wikimedia.org/wiki/File:Green\\_Elephants\\_Garden\\_Sculptures.jpg](https://commons.wikimedia.org/wiki/File:Green_Elephants_Garden_Sculptures.jpg)

## 4.1 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.

### 4.1.1 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 rejet 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.

#### 4.1.2 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.

## 4.2 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.

### 4.2.1 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* ».

#### 4.2.2 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).

### 4.2.3 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 ÉCRIPTIONS 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<sup>3</sup>, 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 entre 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

<sup>3</sup><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<sup>4</sup> 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.

#### 4.2.4 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 noeuds 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

<sup>4</sup><https://www.bucardo.org/Bucardo/>

celui du primaire 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.

#### 4.2.5 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 inconvenients, 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 asymétrique 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.

#### 4.2.6 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 réappliquer 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.

## 4.3 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.

### 4.3.1 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\_recvreplay

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_recvreplay` (nommé `pg_receivexlog` jusqu'en 9.6) qui crée à distance les journaux de transactions à partir d'un flux de réPLICATION.

#### 4.3.2 Streaming replication



- But
  - avoir un retard moins important sur le serveur primaire
- 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 répliquée sur un secondaire.

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

#### 4.3.3 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*.

#### 4.3.4 *Hot Standby*

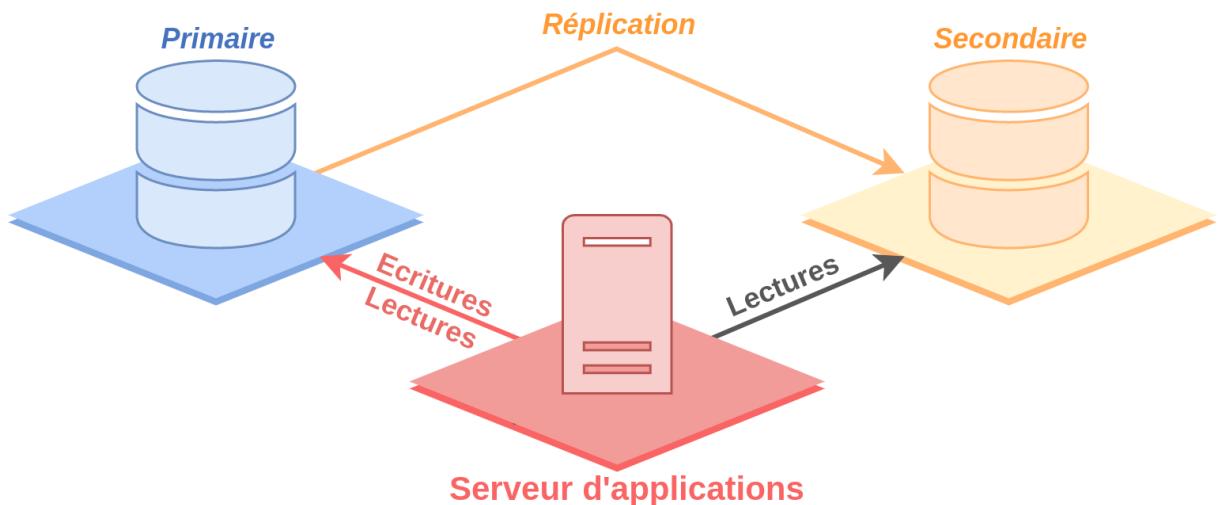


- Serveur secondaire
  - accepte les connexions entrantes
  - requêtes en lecture seule et sauvegardes
  - prêt à prendre le relai du primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée

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...

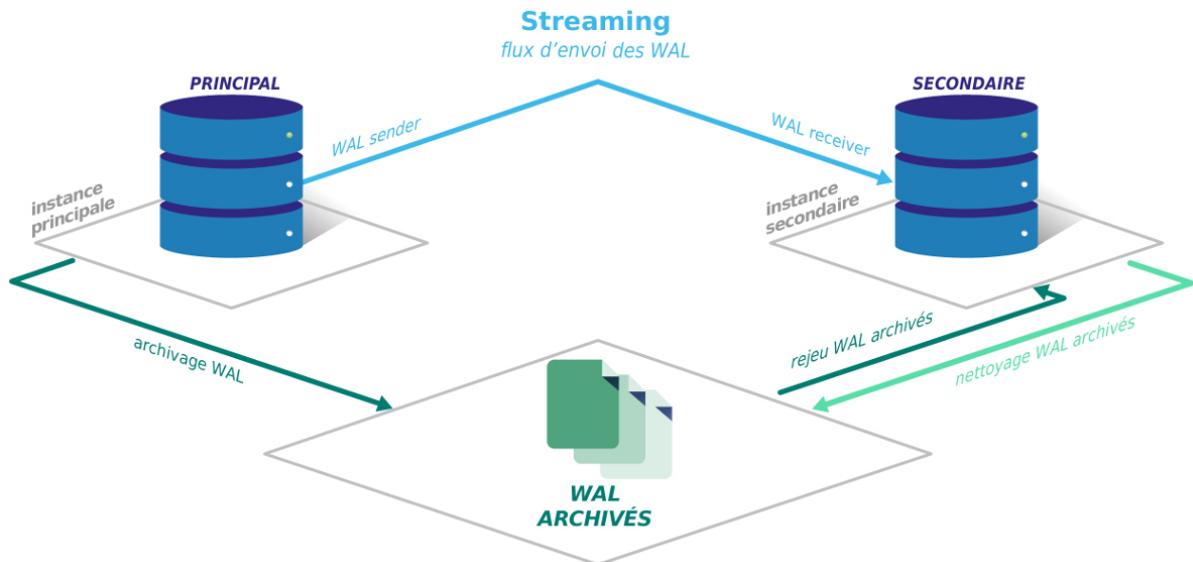
#### 4.3.5 Exemple



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.

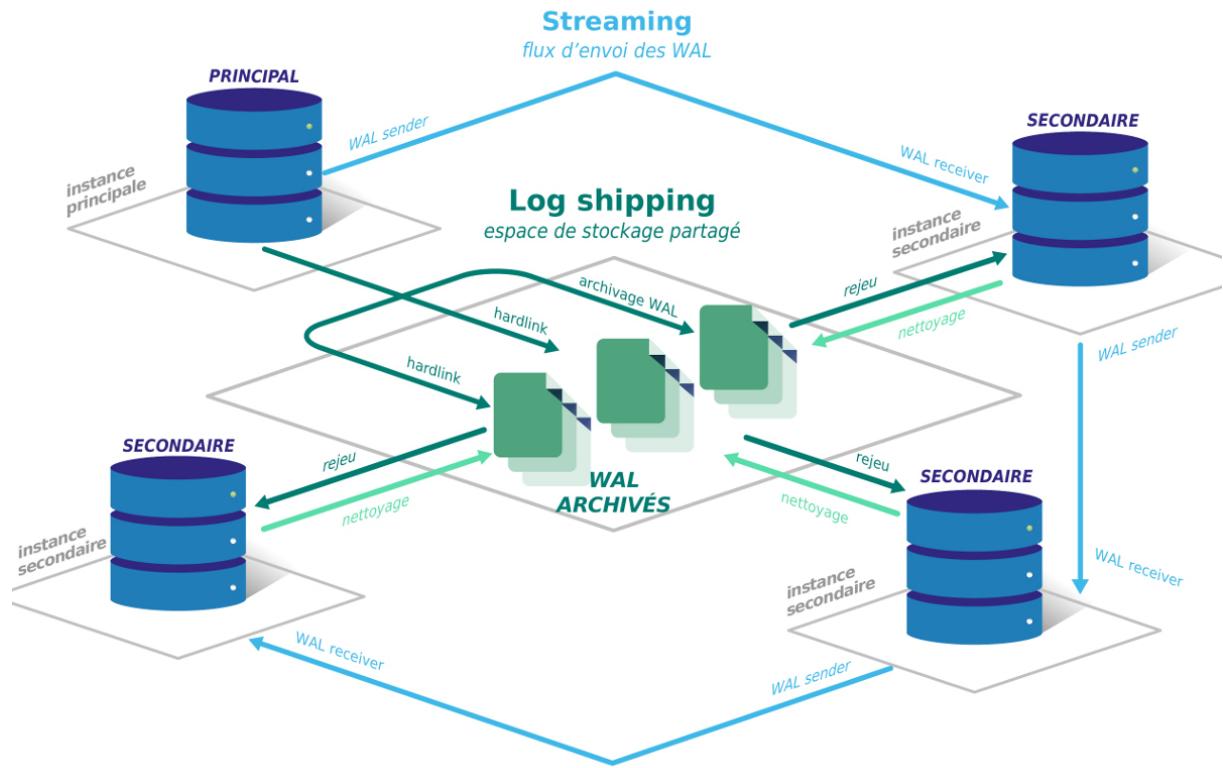
#### 4.3.6 RéPLICATION interne

### RÉPLICATION INTERNE POSTGRESQL



#### 4.3.7 RéPLICATION EN CASCADE

### RÉPLICATION POSTGRESQL EN CASCADE



## 4.4 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.

### 4.4.1 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.

## 4.5 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 combinent 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<sup>5</sup> ou cet article : Haute disponibilité avec PostgreSQL<sup>6</sup>, Guillaume Lelarge, 2009.

---

<sup>5</sup>[https://wiki.postgresql.org/wiki/Replication,\\_Clustering,\\_and\\_Connection\\_Pooling](https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling)

<sup>6</sup>[https://public.dalibo.com/exports/conferences/\\_archives/\\_2011/201102\\_haute\\_disponibilite\\_avec\\_postgresql/solutionsha.pdf](https://public.dalibo.com/exports/conferences/_archives/_2011/201102_haute_disponibilite_avec_postgresql/solutionsha.pdf)

## 4.6 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 réPARTITION 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) vous 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. À partir de la version 11, les insertions dans une table partitionnée peuvent même être redirigées vers une partition distante de manière transparente. Vous trouverez un exemple dans cet article<sup>7</sup>. La réPLICATION logique peut servir à synchroniser des tables non distribuées sur les instances. Le partitionnement des tables au sein d'une instance est une option. Il reste

<sup>7</sup><https://pgdash.io/blog/postgres-11-sharding.html>

cependant des limites : le parcours simultané des partitions distantes n'est possible qu'à partir de PostgreSQL 14<sup>8</sup>) ; un index sur une table partitionnée ne peut être unique s'il y existe des partitions distantes, et il ne sera créé que sur les partitions locales ; enfin le DDL reste manuel.

- 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<sup>9</sup> 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 du 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<sup>10</sup> ou Greenplum<sup>11</sup>. 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<sup>12</sup>.

---

<sup>8</sup>[https://dali.bo/workshop14\\_html#lecture-asyncrone-des-tables-distantes](https://dali.bo/workshop14_html#lecture-asyncrone-des-tables-distantes)

<sup>9</sup><https://www.citusdata.com/product>

<sup>10</sup><https://www.postgres-xl.org/>

<sup>11</sup><https://greenplum.org/>

<sup>12</sup><https://momjian.us/main/writings/pgsql/sharding.pdf>

## 4.7 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 redonner l'alimentation, la mémoire, les processeurs, etc.

### 4.7.1 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.

#### 4.7.2 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<sup>13</sup> 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...).

#### 4.7.3 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/fallback* parfois sophistiqués. Ces solutions présentent peu ou prou les mêmes caractéristiques, avantages et inconvénients que DRBD. Ces technologies ont en revanche le défaut d'être opaques et de nécessiter une main d'œuvre hautement qualifiée.

## 4.8 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<sup>14</sup> » (documentation officielle)

#### 4.8.1 Questions



N'hésitez pas, c'est le moment !

---

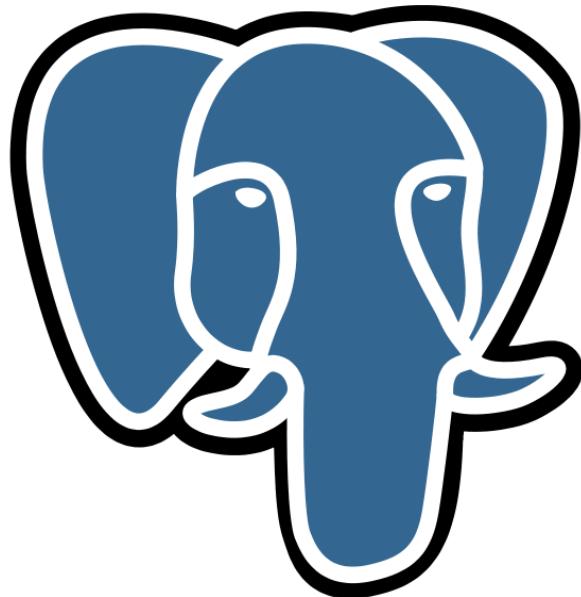
<sup>14</sup><https://docs.postgresql.fr/current/high-availability.html>

## 4.9 QUIZ



[https://dali.bo/w1\\_quiz](https://dali.bo/w1_quiz)

## 5/ RéPLICATION PHYSIQUE : FONDAMENTAUX



## 5.1 INTRODUCTION



- Principes
- Mise en place

La réPLICATION physique de PostgreSQL existe depuis la version 8.2 avec la réPLICATION par fichier. Au fil des versions, de nouvelles fonctionnalités sont apparues.

Avec la version 10, PostgreSQL dispose d'une mécanique de réPLICATION en flux très complète. Ce module permet de comprendre les principes derrière ce type de réPLICATION, sa mise en place et son administration.

### 5.1.1 Au menu



- Principes
- Mise en place
- Administration

### 5.1.2 Objectifs



- Connaître les avantages et limites de la réPLICATION physique
- Savoir la mettre en place
- Savoir administrer et superviser une solution de réPLICATION physique

## 5.2 CONCEPTS / PRINCIPES



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

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

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

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

### 5.2.1 Principales évolutions de la réPLICATION



- 8.2 : RéPLICATION par journaux (*log shipping*), *Warm Standby*
- 9.0 : RéPLICATION en *streaming*, *Hot Standby*
- 9.1 à 9.3 : RéPLICATION synchrone, cascade, `pg_basebackup`
- 9.4 : Slots de réPLICATION, délai de réPLICATION, décodage logique
- 9.5 : `pg_rewind`, archivage depuis un serveur secondaire
- 9.6 : Rejeu synchrone
- 10 : RéPLICATION synchrone sur base d'un quorum, slots temporaires
- 12 : Déplacement de la configuration du `recovery.conf` vers le `postgresql.conf`
- 13 : Sécurisation des slots

La mise en place de la réPLICATION a été très progressive au fil des versions. Elle pouvait être simpliste au

départ, mais elle est à présent au point, et beaucoup plus complète. L'historique permet d'expliquer certaines particularités et complexités.

La version 8.0, en 2005, contenait déjà tout le code qui permet aujourd'hui, après un crash du serveur, de relire les journaux pour rendre à nouveau cohérents les fichiers de données. Pour répliquer une instance, il a suffit d'automatiser l'envoi des journaux vers un serveur secondaire, qui passe son temps à les rejouer, journal après journal (*log shipping*), pour obtenir un serveur prêt à prendre le relai du primaire (*Warm Standby*).

Le serveur secondaire a ensuite été rendu utilisable pour des requêtes en lecture seule (*Hot Standby*).

La réPLICATION a été ensuite améliorée : elle peut se faire en continu (*streaming replication*) et non plus journal par journal, pour réduire le retard du secondaire. Elle peut être synchrone, avec différents niveaux d'arbitrage entre performance et sécurité, et même s'effectuer en cascade, ou avec un délai, et cela en *log shipping* comme en *streaming*.

Puis, l'ajout des slots de réPLICATION a permis au serveur primaire de connaître la position de ses serveurs secondaires, pour savoir quels journaux de transactions sont encore nécessaires.

En parallèle, différents éléments ont été apportés, permettant l'apparition de la réPLICATION logIQUE en version 10.

L'outil pg\_rewind a été ajouté pour faciliter la reconstruction d'un ancien serveur primaire devenu secondaire. Il est plus flexible depuis PostgreSQL 13 et peut utiliser un secondaire comme référence depuis la version 14.

La version 10 ajoute la possibilité d'appliquer arbitrairement une réPLICATION synchrone à un sous-ensemble d'un groupe d'instances (*\_quorum*), et non plus juste par ordre de priorité, avec ce paramétrage :

```
syncronous_standby_names = [FIRST] | [ANY] num_sync (node1, node2, ...)
```

À partir de PostgreSQL 10, les slots de réPLICATION peuvent être temporaires, et ne durer que le temps de la connexion qui l'a créé.

La version 12 ne change rien sur le fond, mais opère une modification technique lourde : le fichier de paramétrage traditionnel sur le secondaire, `recovery.conf`, disparaît, et ses paramètres sont déplacés dans `postgresql.conf` (ou `postgresql.auto.conf`), ce qui facilite la centralisation de la configuration, et évite d'avoir à redémarrer systématiquement après modification des paramètres concernés.

La version 13 améliore encore ce dernier point. De plus, elle supprime le plus gros inconvénient des slots de réPLICATION en posant un maximum à la volumétrie qu'ils peuvent conserver (`max_slot_wal_keep_size`).

La version 15 accélère le rejeu en *log shipping*.

L'outil de sauvegarde physique pg\_basebackup a profité de toutes ces améliorations et peut suffire aux sauvegardes physiques et construction de serveurs secondaires dans les cas simples. Parallèlement, les différents outils externes ont également beaucoup progressé.

### 5.2.2 Avantages



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

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

L'autre gros avantage est qu'il réplique tout : modification des données comme évolutions de la structure de la base (DDL), séquences, *large objects*, fonctions... C'est une fonctionnalité que tous les systèmes de réPLICATION logique (notamment par trigger) aimeraient avoir.

### 5.2.3 Inconvénients



- RéPLICATION de l'instance complète
- Serveur secondaire uniquement en lecture
- Impossible de changer d'architecture
- Même version majeure de PostgreSQL pour tous les serveurs

De manière assez étonnante, l'avantage de tout répliquer est aussi un inconvénient : avec la réPLICATION interne physique de PostgreSQL, il n'est pas possible de ne répliquer qu'une seule base ou que quelques tables.

De même, il n'est pas possible de créer des objets supplémentaires sur le serveur secondaire, comme des index ou des tables de travail, ce qui serait pourtant bien pratique pour de la création de rapports ou pour stocker des résultats intermédiaires de calculs statistiques. Le serveur secondaire est vraiment réservé aux opérations de lecture seule (sauvegardes, répartition de la charge en lecture...) Ces limites ont motivé le développement de la réPLICATION logique pour certains cas d'usage qui ne relèvent pas de la haute disponibilité.

La réPLICATION se passe au niveau du contenu des fichiers et des journaux de transactions. En conséquence, il n'est pas possible d'avoir deux nœuds du système de réPLICATION avec une architecture différente. Par exemple, ils doivent être tous les deux 32 bits ou 64 bits, mais pas un mélange. De même,

les deux nœuds doivent être *big endian* ou *little endian*, et doivent aussi être à la même version majeure (pas forcément mineure, ce qui facilite les mises à jours mineures). Pour éviter tout problème de librairie, il est même conseillé d'utiliser des systèmes les plus proches possibles (même distribution de même niveau de mise à jour).

## 5.3 MISE EN PLACE DE LA RÉPLICATION PAR STREAMING



- Réplication en flux
- Un processus du serveur primaire discute avec un processus du serveur secondaire
  - d'où un *lag* moins important
- Asynchrone ou synchrone
- En cascade

Le serveur secondaire lance un processus appelé `walreceiver` dont le but est de se connecter au serveur primaire et d'attendre les modifications de la réplication.

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

Cette méthode permet une réplication un peu plus proche du serveur primaire. On peut même configurer un mode synchrone : le client ne récupère pas la main tant que la modification demandée n'est pas enregistrée sur le serveur primaire **et** sur le serveur secondaire synchrone. Cela s'effectue à la validation de la transaction, implicite ou lors d'un COMMIT.

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

### 5.3.1 Serveur primaire (1/2) - configuration



Dans `postgresql.conf`:

- `wal_level = replica (ou logical)`
- `max_wal_senders = X`
  - 1 par client par *streaming*
  - défaut : 10
- `wal_sender_timeout = 60s`

Il faut tout d'abord s'assurer que PostgreSQL enregistre suffisamment d'informations pour que le serveur secondaire puisse rejouer toutes les modifications survenant sur le serveur primaire. Par défaut, PostgreSQL dispose de certaines optimisations qui lui permettent d'éviter certaines écritures quand cela ne pose pas de problème pour l'intégrité des données en cas de crash. Par exemple, il est inutile de tracer toutes les opérations d'une transaction qui commence par vider une table, puis qui la remplit. En cas de crash, l'opération complète est annulée.

Cependant, dans le cas de la réPLICATION, il est nécessaire d'avoir les étapes intermédiaires et il est donc essentiel d'enregistrer ces informations supplémentaires dans les journaux. Pour arbitrer entre les deux (avoir d'excellentes performances si on n'utilise pas la réPLICATION, ou utiliser la réPLICATION en acceptant des performances un peu moindres), utiliser le paramètre `wal_level`. Comme son nom l'indique, il permet de préciser le niveau d'informations que l'on souhaite avoir dans les journaux.

- Le niveau `replica` est celui par défaut, adapté à l'archivage ou la réPLICATION, en plus de la sécurisation contre les arrêts brutaux. C'est le niveau par défaut depuis la version 10.
- Le niveau `minimal` n'offre que la protection contre les arrêts brutaux, mais ne permet ni réPLICATION ni sauvegardes PITR. C'était la valeur par défaut jusqu'en 9.6 incluse. Ce niveau ne sert plus guère qu'aux environnements ni archivés ni répliqués pour réduire la quantité de journaux générés dans certains cas.
- Le niveau `logical` est le plus complet et doit être activé pour l'utilisation du décodage logique, notamment pour utiliser la réPLICATION logique à partir de la version 10.



Avant la version 9.6, existaient deux niveaux intermédiaires : `archive` (archivage pur) et `hot_standby` (pour avoir un serveur secondaire en lecture seule). Depuis, ces deux valeurs sont assimilées à `replica`.

Le serveur primaire accepte un nombre maximum de connexions de réPLICATION : il s'agit du paramètre `max_wal_senders`. Ce paramètre vaut par défaut 10, ce qui devrait suffire dans la plupart des cas. (Jusqu'en 9.6, sa valeur par défaut était 0, et devait être incrémentée d'au moins 1 pour chaque serveur secondaire susceptible de se connecter, sans oublier des outils utilisant le *streaming* comme `pg_basebackup` ou `pg_recvwal`. Il est conseillé de prévoir « large » d'entrée : l'impact mémoire est négligeable, et cela évite d'avoir à redémarrer l'instance primaire à chaque modification.)

Le paramètre `wal_sender_timeout` permet de couper toute connexion inactive après le délai indiqué par ce paramètre. Par défaut, le délai est d'une minute. Cela permet au serveur primaire de ne pas conserver une connexion coupée ou dont le client a disparu pour une raison ou une autre. Le secondaire retentera par la suite une connexion complète.

### 5.3.2 Serveur primaire (2/2) - authentification



- Le serveur secondaire doit pouvoir se connecter au serveur primaire
- Pseudo-base `replication`
- Utilisateur dédié conseillé avec attributs `LOGIN` et `REPLICATION`
- Configurer `pg_hba.conf`:

```
host replication user_repli 10.2.3.4/32 scram-sha-256
```

- Recharger la configuration

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

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

```
host replication user_repli 10.0.0.2/32 scram-sha-256
```

Dans ce cas, l'utilisateur `user_repli` pourra entamer une connexion de réplication vers le serveur primaire à condition que la demande de connexion provienne de l'adresse IP `10.0.0.2` et que cette demande de connexion précise le bon mot de passe (au format `md5` ou `scram-sha-256`).

Un utilisateur dédié à la réplication est conseillé pour des raisons de sécurité. On le créera avec les droits suivants :

```
CREATE ROLE user_repli LOGIN REPLICATION ;
```

et bien sûr un mot de passe complexe.

À partir de la version 10, les connexions locales de réplication sont autorisées par défaut sans mot de passe.

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

### 5.3.3 Serveur secondaire (1/3) - copie des données



Copie des données du serveur primaire (à chaud !) :

- Copie généralement à chaud donc incohérente !
- Le plus simple : pg\_basebackup
  - simple mais a des limites
- Idéal : outil PITR
- Possible : rsync, cp...
  - ne pas oublier pg\_backup\_start()/pg\_backup\_stop() !
  - exclure certains répertoires et fichiers
  - garantir la disponibilité des journaux de transaction

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



Rappelons que généralement cette copie aura lieu à chaud, donc une simple copie directe sera incohérente.

#### **pg\_basebackup :**

L'outil le plus simple est pg\_basebackup. Ses avantages sont sa disponibilité et sa facilité d'utilisation. Il sait ce qu'il n'y a pas besoin de copier et peut inclure les journaux nécessaires pour ne pas avoir à paramétriser l'archivage.

Il peut utiliser la connexion de réPLICATION déjà prévue pour le secondaire, poser des slots temporaires ou le slot définitif.

Pour faciliter la mise en place d'un secondaire, il peut générer les fichiers de configuration à partir des paramètres qui lui ont été fournis (option --write-recovery-conf).

Malgré beaucoup d'améliorations dans les dernières versions, la limite principale de pg\_basebackup reste d'exiger un répertoire cible vide : on doit toujours recopier l'intégralité de la base copiée. Cela peut être pénible lors de tests répétés avec une grosse base, ou avec une liaison instable.

#### **Outils PITR :**

L'idéal est un outil de restauration PITR permettant la restauration en mode delta, par exemple pgBackRest avec l'option --delta. Ne sont restaurés que les fichiers ayant changé, et le primaire n'est pas chargé par la copie.

#### **rsync :**

Un script de copie reste une option possible. Il est possible de le faire manuellement, tout comme pour une sauvegarde PITR.



Une copie manuelle implique que les journaux sont archivés par ailleurs.

Rappelons les trois étapes essentielles :

- le pg\_backup\_start();
- la copie des fichiers : généralement avec rsync --whole-file, ou tout moyen permettant une copie fiable et rapide ;
- le pg\_backup\_stop().

On exclura les fichiers inutiles lors de la copie qui pourraient gêner un redémarrage, notamment le fichier postmaster.pid et les répertoires pg\_wal, pg\_replslot, pg\_dynshmem, pg\_notify, pg\_serial, pg\_snapshots, pg\_stat\_tmp, pg\_subtrans, pgslq\_tmp\*. La liste complète figure dans la documentation officielle<sup>1</sup>.

#### 5.3.4 Serveur secondaire (2/3) - configuration



- <v12 :
  - recovery.conf
  - avec standby\_mode = on & autres paramètres
- v12 :
  - postgresql.conf ou postgresql.auto.conf
  - standby.signal (vide)
- primary\_conninfo (*streaming*)
  - penser à .pgpass
- wal\_receiver\_timeout

Jusqu'en v11, on active le mode *standby* en créant un fichier texte recovery.conf dans le PG-DATA de l'instance (y compris dans le cas des distributions qui mettent leur configuration ailleurs, en premier lieu Debian et Ubuntu) et en y plaçant le paramètre standby\_mode à on, ainsi que les paramètres évoqués ci-dessous.

<sup>1</sup><https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP>

À partir de la v12, les options concernées sont dans `postgresql.conf` ou `postgresql.auto.conf`, et il n'y a plus qu'à créer un fichier vide nommé `standby.signal` dans le répertoire PGDATA.

PostgreSQL doit aussi savoir comment se connecter au serveur primaire. C'est le paramètre `primary_conninfo` qui le lui dit. Il s'agit d'un DSN standard où il est possible de spécifier l'adresse IP de l'hôte ou son alias, le numéro de port, le nom de l'utilisateur, etc. Il est aussi possible de spécifier le mot de passe, mais c'est risqué en terme de sécurité. En effet, PostgreSQL ne vérifie pas si ce fichier est lisible par quelqu'un d'autre que lui. Il est donc préférable de placer le mot de passe dans le fichier `.pgpass`, généralement dans `~postgres/` sur le secondaire, fichier qui n'est utilisé que s'il n'est lisible que par son propriétaire.

On aura donc par exemple une de ces chaînes (la première a été créée automatiquement par `pg_basebackup`, toutes les options ne sont pas nécessaires) :

```
primary_conninfo = 'user=user_repli host=prod port=5433
↪ password=ko3wooPh3evo9tae0Aen
sslmode=prefer sslcompression=0 krbsrvname=postgres target_session_attrs=any'

primary_conninfo = 'user=postgres host=prod passfile=/var/lib/postgresql/.pgpass'
```

De manière optionnelle peuvent être configurés au même endroit `primary_slot_name` (pour sécuriser la réPLICATION avec un slot de réPLICATION) ou `restore_command` (pour sécuriser avec un accès à la sauvegarde PITR).

De manière symétrique à `wal_sender_timeout` sur le primaire, `wal_receiver_timeout` sur le secondaire coupe une connexion inactive. Le secondaire retentera la connexion plus tard.

### 5.3.5 Serveur secondaire (3/3) - démarrage



- Démarrer PostgreSQL
- Suivre dans les traces que tout va bien

Il ne reste plus qu'à démarrer le serveur secondaire.

En cas de problème, le premier endroit où aller chercher est bien entendu le fichier de trace `postgresql.log`

### 5.3.6 Processus



Sur le primaire :

- `walsender replicator ... streaming 0/3BD48728`

Sur le secondaire :

- `walreceiver streaming 0/3BD48728`

## 5.4 PROMOTION



- Attention au *split-brain* !
- Vérification avant promotion
- Promotion : méthode et déroulement
- Retour à l'état stable

### 5.4.1 Attention au split-brain !



- Si un serveur secondaire devient le nouveau primaire
  - s'assurer que l'ancien primaire ne reçoit plus d'écriture
  - Éviter que les deux instances soient ouvertes aux écritures
    - confusion et perte de données !

La pire chose qui puisse arriver lors d'une bascule est d'avoir les deux serveurs, ancien primaire et nouveau primaire promu, ouverts tous les deux en écriture. Les applications risquent alors d'écrire dans l'un ou l'autre...

Quelques histoires « d'horreur » à ce sujet :

- de nombreux exemples sur diverses technologies de réPLICATION<sup>2</sup> ;
- *post mortem* d'un gros problème chez Github en 2018<sup>3</sup>.

---

<sup>2</sup><https://github.blog/2018-10-30-oct21-post-incident-analysis>

<sup>3</sup><https://aphyr.com/posts/288-the-network-is-reliable>

### 5.4.2 Vérification avant promotion



- Primaire :

```
systemctl stop postgresql-14

$ pg_controldata -D /var/lib/pgsql/14/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state: shut down
Latest checkpoint's REDO location: 0/3BD487D0
```

- Secondaire :

```
$ psql -c 'CHECKPOINT;'
$ pg_controldata -D /var/lib/pgsql/14/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state: in archive recovery
Latest checkpoint's REDO location: 0/3BD487D0
```

Avant une bascule, il est capital de vérifier que toutes les modifications envoyées par le primaire sont arrivées sur le secondaire. Si le primaire a été arrêté proprement, ce sera le cas. Après un CHECKPOINT sur le secondaire, on y retrouvera le même emplacement dans les journaux de transaction.

Ce contrôle doit être systématique avant une bascule. Même si toutes les écritures applicatives sont stoppées sur le primaire, quelques opérations de maintenance peuvent en effet écrire dans les journaux et provoquer un écart entre les deux serveurs (divergence). Il n'y aura alors pas de perte de données mais cela pourrait gêner la transformation de l'ancien primaire en secondaire, par exemple.

Noter que pg\_controldata n'est pas dans les chemins par défaut des distributions. La fonction SQL pg\_control\_checkpoint() affiche les mêmes informations, mais n'est bien sûr pas accessible sur un primaire arrêté.

### 5.4.3 Promotion du standby : méthode



- Shell :
  - `pg_ctl promote`
- SQL :
  - fonction `pg_promote()` (v12+)
- Déclenchement par fichier :
  - `promote_trigger_file(v12+)`
  - `trigger_file(<=v11)`

Il existe plusieurs méthodes pour promouvoir un serveur PostgreSQL en mode *standby*. Les méthodes les plus appropriées sont :

- l'action `promote` de l'outil `pg_ctl`, ou de son équivalent dans les scripts des paquets d'installation, comme `pg_ctlcluster` sous Debian ;
- la fonction SQL `pg_promote` (à partir de la version 12).

Historiquement PostgreSQL dispose aussi d'un fichier de déclenchement (*trigger file*), défini par le paramètre `promote_trigger_file` en version 12, ou, dans les versions précédentes, `trigger_file` (dans `recovery.conf`). Un serveur secondaire vérifie en permanence si ce fichier existe. Dès qu'il apparaît, l'instance est promue. Par mesure de sécurité, il est préconisé d'utiliser un emplacement accessible uniquement aux administrateurs.

### 5.4.4 Promotion du standby : déroulement



Une promotion déclenche :

- déconnexion de la *streaming replication* (bascule programmée)
- rejet des dernières transactions en attente d'application
- choix d'une nouvelle *timeline* du journal de transaction
- suppression du fichier `standby.signal`
- ouverture aux écritures

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

Ensuite, il choisit une nouvelle *timeline* pour son journal de transactions. La timeline est le premier numéro dans le nom du segment (fichier WAL), soit par exemple une timeline 5 pour un fichier nommé 000000050000003200000031).

Enfin, il autorise les connexions en lecture et en écriture.

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



Les *timelines* ne changent pas que lors des promotions, mais aussi lors des restaurations PITR. En général, on désire que les secondaires (parfois en cascade) suivent, si possible, les évolutions de *timeline* avec ce paramétrage (par défaut depuis la version 12) :

```
recovery_target_timeline = latest
```

#### 5.4.5 Opérations après promotion du standby



- VACUUM ANALYZE conseillé
  - calcul d'informations nécessaires pour autovacuum

Il n'y a aucune opération obligatoire après une promotion. Cependant, il peut être intéressant d'exécuter un VACUUM ou un ANALYZE pour que PostgreSQL mette à jour les estimations de nombre de lignes vivantes et mortes. Ces estimations sont utilisées par l'autovacuum pour lutter contre la fragmentation des tables et mettre à jour les statistiques sur les données. Or ces estimations faisant partie des statistiques d'activité, elles ne sont pas répliquées vers les secondaires. Il est donc intéressant de les mettre à jour après une promotion.

#### 5.4.6 Retour à l'état stable



- Si un *standby* a été momentanément indisponible, reconnexion directe possible si :
  - journaux nécessaires encore présents sur primaire (slot, `wal_keep_size/wal_keep_segments`)
  - journaux nécessaires présents en archives (`restore_command`)
- Sinon
  - « décrochage »
  - reconstruction nécessaire

Si un serveur secondaire est momentanément indisponible mais revient en ligne sans perte de données (réseau coupé, problème OS...), alors il a de bonnes chances de se « raccrocher » à son serveur primaire. Il faut bien sûr que l'ensemble des journaux de transaction depuis son arrêt soit accessible à ce serveur, sans exception.

En cas de réPLICATION par *streaming* : le primaire ne doit pas avoir recyclé les journaux après ses *checkpoints*. Il les aura conservés s'il y a un slot de réPLICATION actif dédié à ce secondaire, ou si on a monté `wal_keep_size` (ou `wal_keep_segments` avant la version 13) assez haut par rapport à l'activité en écriture sur le primaire. Les journaux seront alors toujours disponibles sur le principal et le secondaire rattrapera son retard par *streaming*. Si le primaire n'a plus les journaux, il affichera une erreur, et le secondaire tentera de se rabattre sur le *log shipping*, s'il est aussi configuré.

En cas de réPLICATION par *log shipping*, il faut que la `restore_command` fonctionne, que le stock des journaux remonte assez loin dans le temps (jusqu'au moment où le secondaire a perdu contact), et qu'aucun journal ne manque ou ne soit corrompu. Sinon le secondaire se bloquera au dernier journal chargé. En cas d'échec, ou si le dernier journal disponible vient d'être rejoué, le secondaire basculera sur le *streaming*, s'il est configuré.

Si le secondaire ne peut rattraper le flux des journaux du primaire, il doit être reconstruit par l'une des méthodes précédentes.

### 5.4.7 Retour à l'état stable, suite



- Synchronisation automatique une fois la connexion rétablie
- Mais reconstruction obligatoire :
  - si le serveur secondaire était plus avancé que le serveur promu (« divergence »)
- Reconstruire les serveurs secondaires à partir du nouveau principal :
  - rsync, restauration PITR, plutôt que pg\_basebackup
  - pg\_rewind
- Reconstruction : manuelle !
- Tablespaces !

Un secondaire qui a bien « accroché » son primaire se synchronise automatiquement avec lui, que ce soit par *streaming* ou *log shipping*. C'est notamment le cas si l'on vient de le construire depuis une sauvegarde ou avec pg\_basebackup, et que l'archivage ou le *streaming* alimentent correctement le secondaire. Cependant, il y a des cas où un secondaire ne peut être simplement raccroché à un primaire, notamment si le secondaire se croit plus avancé que le primaire dans le flux des journaux.

Le cas typique est un ancien primaire que l'on veut transformer en secondaire d'un ancien secondaire promu. Si la bascule s'était faite proprement, et que l'ancien primaire avait pu envoyer tous ses journaux avant de s'arrêter ou d'être arrêté, il n'y a pas de problème. Si le primaire a été arrêté violemment, sans pouvoir transmettre tous ses journaux, l'ancien secondaire n'a rejoué que ce qu'il a reçu, puis a ouvert en écriture sa propre *timeline* depuis un point moins avancé que là où le primaire était finalement arrivé avant d'être arrêté. Les deux serveurs ont donc « divergé », même pendant très peu de temps. Les journaux non envoyés au nouveau primaire doivent être considérés comme perdus. Quand l'ancien primaire revient en ligne, parfois très longtemps après, il voit que sa *timeline* est plus avancée que la version qu'en a gardée le nouveau primaire. Il ne sait donc pas comment appliquer les journaux qu'il reçoit du nouveau primaire.

La principale solution, et la plus simple, reste alors la reconstruction du secondaire à raccrocher.

L'utilisation de pg\_basebackup est possible mais déconseillée si la volumétrie est importante : cet outil impose une copie de l'ensemble des données du serveur principal, et ce peut être long.

La durée de reconstruction des secondaires peut être optimisée en utilisant des outils de synchronisation de fichiers pour réduire le volume des données à transférer. Les outils de restauration PITR offrent souvent une restauration en mode delta (notamment l'option --delta de pgBackRest) et c'est ce qui est généralement à privilégier. Dans un script de sauvegarde PITR, rsync --whole-file reste une bonne option.

Le fait de disposer de l'ensemble des fichiers de configuration sur tous les nœuds permet de gagner un temps précieux lors des phases de reconstruction, qui peuvent également être scriptées.

Par contre, les opérations de reconstructions se doivent d'être lancées **manuellement** pour éviter tout risque de corruption de données dues à des opérations automatiques externes, comme lors de l'utilisation de solutions de haute disponibilité.

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

Une alternative à la reconstruction est l'utilisation de l'outil `pg_rewind` pour « rembobiner » l'ancien primaire, si tous les journaux nécessaires sont disponibles.

## 5.5 CONCLUSION



- Système de réPLICATION fiable
- Simple à maîtriser et à configurer

La réPLICATION interne à PostgreSQL est le résultat de travaux remontant aussi loin que la version 8.0. Elle est fondée sur des bases solides et saines.

Cette réPLICATION reste fidèle aux principes du moteur de PostgreSQL :

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

## 5.6 QUIZ



[https://dali.bo/w2a\\_quiz](https://dali.bo/w2a_quiz)

## 5.7 TRAVAUX PRATIQUES



Ce TP suppose que les instances tournent sur la même machine. N'oubliez pas qu'il faut un répertoire de données et un numéro de port par serveur PostgreSQL. Dans la réalité, il s'agira de deux machines différentes : l'archivage nécessitera des opérations supplémentaires (montage de partitions réseau, connexion ssh sans mot de passe...).

### 5.7.1 RéPLICATION ASYNCHRONE EN FLUX AVEC UN SEUL SECONDAIRE



**But :** Mettre en place une réPLICATION ASYNCHRONE EN FLUX.

- Créer l'instance principale dans `/var/lib/pgsql/14/instance1`.
- Mettre en place la configuration de la réPLICATION PAR *streaming*.
  - L'utilisateur dédié sera nommé **repli**.
- Créer la première instance secondaire **instance2**, par copie à chaud du répertoire de données avec `pg_basebackup` vers `/var/lib/pgsql/14/instance2`.
  - Penser à modifier le port de cette nouvelle instance avant de la démarrer.
- Démarrer **instance2** et s'assurer que la réPLICATION fonctionne bien avec `ps`.
  - Tenter de se connecter au serveur secondaire.
  - Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

### 5.7.2 PROMOTION DE L'INSTANCE SECONDAIRE



**But :** Promouvoir un serveur secondaire en primaire.

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

### 5.7.3 Retour à la normale



**But :** Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale (/var/lib/pgsql/14/instance1) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant pg\_basebackup.
- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.
- Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

## 5.8 TRAVAUX PRATIQUES (SOLUTIONS)



Cette solution se base sur un système CentOS 7 ou Rocky Linux 8, installé à minima depuis les paquets du PGDG, et en anglais.

Des adaptations seraient à faire pour un autre système (Debian, Windows...). La version de PostgreSQL est la version 14. Adapter au besoin pour une version ultérieure. Noter que les versions 12 et précédentes utilisent d'autres fichiers.



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

La mise en place d'une ou plusieurs instances sur le même poste est décrite plus haut.

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

Ensuite, afin de réaliser l'ensemble des TP, configurer 4 services PostgreSQL « instance[1-4] ».

```
cp /lib/systemd/system/postgresql-14.service \
 /etc/systemd/system/instance1.service

sed -i "s|/var/lib/pgsql/14/data||/var/lib/pgsql/14/instance1|" \
 /etc/systemd/system/instance1.service

cp /lib/systemd/system/postgresql-14.service \
 /etc/systemd/system/instance2.service

sed -i "s|/var/lib/pgsql/14/data||/var/lib/pgsql/14/instance2|" \
 /etc/systemd/system/instance2.service

cp /lib/systemd/system/postgresql-14.service \
 /etc/systemd/system/instance3.service

sed -i "s|/var/lib/pgsql/14/data||/var/lib/pgsql/14/instance3|" \
 /etc/systemd/system/instance3.service

cp /lib/systemd/system/postgresql-14.service \
 /etc/systemd/system/instance4.service

sed -i "s|/var/lib/pgsql/14/data||/var/lib/pgsql/14/instance4|" \
 /etc/systemd/system/instance4.service
```

### 5.8.1 RéPLICATION ASYNCHRONE EN FLUX AVEC UN SEUL SECONDAIRE

- Créer l'instance principale dans /var/lib/pgsql/14/instance1.

```
export PGSETUP_INITDB_OPTIONS='--data-checksums'
/usr/pgsql-14/bin/postgresql-14-setup initdb instance1
Initializing database ... OK
systemctl start instance1
```

- Mettre en place la configuration de la réPLICATION par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

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

Nous allons cependant modifier le fichier `/var/lib/pgsql/14/instance1/pg_hba.conf` pour que l'accès en réPLICATION soit autorisé pour l'utilisateur **repli** :

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

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

```
$ createuser --no-superuser --no-createrole --no-createdb --replication -P repli
Enter password for new role:
Enter it again:
```

Configurer ensuite le fichier `.pgpass` de l'utilisateur système `postgres` :

```
$ echo "*:*:*:repli:confidentiel" > ~/.pgpass
$ chmod 600 ~/.pgpass
```

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

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

- Créer la première instance secondaire **instance2**, par copie à chaud du répertoire de données avec `pg_basebackup` vers `/var/lib/pgsql/14/instance2`.
- Penser à modifier le port de cette nouvelle instance avant de la démarrer.

Utiliser `pg_basebackup` pour créer l'instance secondaire :

```
$ pg_basebackup -D /var/lib/pgsql/14/instance2 -P -R -c fast -h 127.0.0.1 -U repli
25314/25314 kB (100%), 1/1 tablespace
```

L'option `-R` ou `--write-recovery-conf` de `pg_basebackup` a préparé la configuration de la mise en réPLICATION en créant le fichier `standby.signal` ainsi qu'en configurant `primary_conninfo` dans le fichier `postgresql.auto.conf` (dans les versions antérieures à la 11, il renseignerait `recovery.conf`) :

```
$ cat /var/lib/pgsql/14/instance2/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 krb_srvname=postgres target_session_attrs=any'

$ ls /var/lib/pgsql/14/instance2/standby.signal
/var/lib/pgsql/14/instance2/standby.signal
```

Il faut désormais positionner le port d'écoute dans le fichier de configuration, c'est-à-dire `/var/lib/pgsql/14/instance2/postgresql.conf`:

`port=5433`

- Démarrer **instance2** et s'assurer que la réPLICATION fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

Il ne reste désormais plus qu'à démarrer l'instance secondaire :

```
systemctl start instance2
```

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

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

La première partie concerne le serveur secondaire :

```
PID CMD
9671 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
9673 _ postgres: logger
9674 _ postgres: startup recovering 000000010000000000000000
9675 _ postgres: checkpointer
9676 _ postgres: background writer
9677 _ postgres: stats collector
9678 _ postgres: walreceiver streaming 0/3000148
```

La deuxième partie concerne le serveur principal :

```
PID CMD
9564 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
9566 _ postgres: logger
9568 _ postgres: checkpointer
9569 _ postgres: background writer
9570 _ postgres: walwriter
9571 _ postgres: autovacuum launcher
9572 _ postgres: stats collector
9573 _ postgres: logical replication launcher
9679 _ postgres: wal sender repli 127.0.0.1(58420) streaming 0/3000148
```

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

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

Créons quelques données sur le principal et assurons-nous qu'elles soient transmises au secondaire :

```
$ createdb b1
$ psql b1
psql (14.1)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);
CREATE TABLE
b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

On constate que le flux a été transmis :

```
b1=# \! ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"
9674 _ postgres: startup recovering 000000010000000000000006
9678 _ postgres: walreceiver streaming 0/6D4CD28
9679 _ postgres: walsender repli 127.0.0.1(58420) streaming 0/6D4CD28
[...]
```

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

```
$ psql -p 5433 b1
psql (14.1)
Type "help" for help.

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

 1000000
b1=# CREATE TABLE t2(id integer);
ERROR: cannot execute CREATE TABLE in a read-only transaction
```

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

Le comportement est visible dans les logs de l'instance secondaire dans le répertoire /var/lib/pgsql/14/insta

```
... LOG: database system is ready to accept read only connections
```

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

## 5.8.2 Promotion de l'instance secondaire

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

Arrêt de l'instance primaire et vérification de son état :

```
systemctl stop instance1

$ /usr/pgsql-14/bin/pg_controldata -D /var/lib/pgsql/14/instance1/ \
| grep -E '(cluster)|(REDO)'

Database cluster state: shut down
Latest checkpoint's REDO location: 0/6D4E5C8
```

Vérification de l'instance secondaire :

```
$ psql -p 5433 -c 'CHECKPOINT';

$ /usr/pgsql-14/bin/pg_controldata -D /var/lib/pgsql/14/instance2/ \
| grep -E '(cluster)|(REDO)'

Database cluster state: in archive recovery
Latest checkpoint's REDO location: 0/6D4E5C8
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en archive recovery et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/instance2 promote

waiting for server to promote.... done
server promoted
```

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

Connectons-nous à ce nouveau primaire et tentons d'y insérer des données :

```
$ psql -p 5433 b1
psql (14.1)
Type "help" for help.

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

b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Les écritures sont désormais bien possible sur cette instance.

### 5.8.3 Retour à la normale

- Reconstruire l'instance initiale (/var/lib/pgsql/14/instance1) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant pg\_basebackup.

Afin de rétablir la situation, nous pouvons réintégrer l'ancienne instance primaire en tant que nouveau secondaire. Pour ce faire, nous devons re-synchroniser les données. Utilisons pg\_basebackup comme précédemment après avoir mis de côté les fichiers de l'ancien primaire :

```
$ mv /var/lib/pgsql/14/instance1 /var/lib/pgsql/14/instance1.old
$ pg_basebackup -D /var/lib/pgsql/14/instance1 -P -R -c fast \
-h 127.0.0.1 -p 5433 -U repli
104385/104385 kB (100%), 1/1 tablespace
```

Créer le fichier `standby.signal` s'il n'existe pas déjà. Contrôler `postgresql.auto.conf` (qui contient potentiellement deux lignes `primary_conninfo` !) et adapter le port :

```
$ touch /var/lib/pgsql/14/instance1/standby.signal
$ cat /var/lib/pgsql/14/instance1/postgresql.auto.conf
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass'' host=127.0.0.1
↳ port=5433 sslmode=prefer sslcompression=0 gssencmode=prefer krb_srvname=postgres
↳ target_session_attrs=any'
```

Repositionner le port d'écoute dans le fichier `/var/lib/pgsql/14/instance1/postgresql.conf`:

`port=5432`

Enfin, démarrer le service :

```
systemctl start instance1
```

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

Les processus adéquats sont bien présents :

```
$ ps -o pid,cmd fx | egrep "(startup|walreceiver)"
12520 _ postgres: startup recovering 000000020000000000000000A
12524 _ postgres: walreceiver streaming 0/A000148
12525 _ postgres: walsender repli 127.0.0.1(38614) streaming 0/A000148

$ psql -p 5432 b1
psql (14.1)
Type "help" for help.
```

En nous connectant à la nouvelle instance secondaire (port 5432), vérifions que les données précédemment insérées dans la table t2 sont bien présentes :

```
b1=# SELECT COUNT(*) FROM t2;
count

1000000
```

- Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

Afin que l'instance 5432 redevienne primaire et celle sur le port 5433 secondaire, on peut ré-appliquer la procédure de promotion vue précédemment dans l'autre sens.

Arrêt de l'instance primaire et vérification de son état :

```
systemctl stop instance2

$ /usr/pgsql-14/bin/pg_controldata -D /var/lib/pgsql/14/instance2/ \
| grep -E '(cluster)|(REDO)'

Database cluster state: shut down
Latest checkpoint's REDO location: 0/C000060
```

Vérification de l'instance secondaire :

```
$ psql -p 5432 -c 'CHECKPOINT;'
$ /usr/pgsql-14/bin/pg_controldata -D /var/lib/pgsql/14/instance1/ \
| grep -E '(cluster)|(REDO)'

Database cluster state: in archive recovery
Latest checkpoint's REDO location: 0/C000060
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en archive recovery et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/instance1 promote

waiting for server to promote.... done
server promoted
```

Afin que **instance2** redevienne l'instance secondaire, créer le fichier `standby.signal`, démarrer le service et vérifier que les processus adéquats sont bien présents :

```
$ touch /var/lib/pgsql/14/instance2/standby.signal

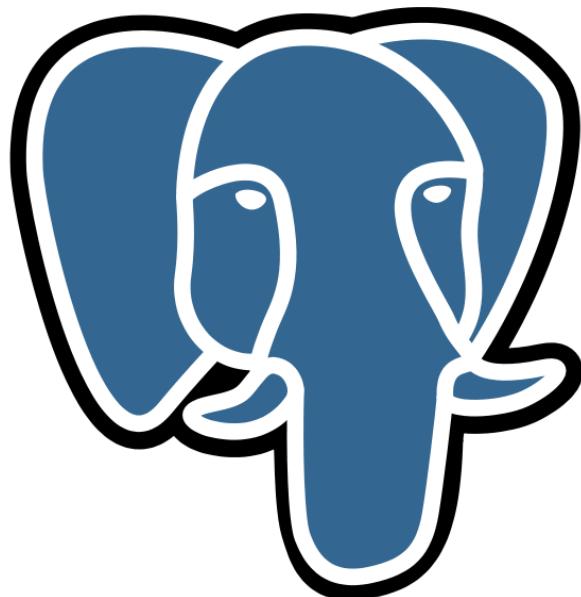
systemctl start instance2

$ ps -o pid,cmd fx | egrep "(startup|walreceiver)"

5844 _ postgres: startup recovering 000000030000000000000000C
5848 _ postgres: walreceiver streaming 0/C0001F0
5849 _ postgres: walsender repli 127.0.0.1(48230) streaming 0/C0001F0
```



## 6/ RéPLICATION PHYSIQUE AVANCÉE



## 6.1 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.

---

### 6.1.1 Au menu



- Supervision
  - Gestion des conflits
  - Asynchrone ou synchrone
  - RéPLICATION en cascade
  - Slot de réPLICATION
  - Log shipping
-

## 6.2 SUPERVISION (STREAMING)



- `pg_is_in_recovery()` : instance en réplication ?
- `pg_stat_replication` : type de réplication, retard des secondaires
  - `dont_lag`
- `pg_replication_slots`
- `pg_stat_wal_receiver`
- Calcul du retard :

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

```
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 procédure stockée appelée `pg_is_in_recovery()`. Elle renvoie la valeur `true` si l'utilisateur se trouve sur un serveur en *hot standby* et `false` sinon.

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`. Elle n'est renseignée que sur le primaire et permet de connaître l'état de tous les serveurs secondaires connectés en *streaming* (mais pas les autres !). Voici un exemple du contenu de cette vue :

```
postgres=# SELECT * FROM pg_stat_replication ;

-[RECORD 1]-----+
pid | 29287
usesysid | 16384
username | repli
application_name | synchrone1
client_addr | 192.168.42.42
client_hostname |
client_port | 56298
backend_start | 2018-02-01 16:35:36.625289+01
backend_xmin |
state | streaming
sent_lsn | 0/353F6720
write_lsn | 0/353F6720
flush_lsn | 0/353F6720
replay_lsn | 0/12CE8EA8
write_lag | 00:00:00.000622
flush_lag | 00:00:00.001785
replay_lag | 00:00:53.239083
sync_priority | 2
sync_state | sync
```

Le calcul de la différence de volumétrie de données entre le primaire et ses secondaires est toujours à faire mais il n'est plus nécessaire de se connecter sur les deux serveurs. Le calcul de différentiel peut-être effectué avec la fonction pg\_wal\_lsn\_diff (nommée pg\_xlog\_location\_diff de 9.3 à 9.6) :

```
postgres=# SELECT pg_wal_lsn_diff(pg_current_wal_lsn(), '0/73D3C1F0');
```

```
-[RECORD 1]-----+-----
pg_wal_lsn_diff | 256
```

Le service **streaming\_delta** de la sonde check\_pgactivity<sup>1</sup> ne fait pas autrement pour suivre les volumétries à recevoir, à appliquer et à rejouer.

La plupart des colonnes se comprennent aisément. La colonne state indique l'état du flux de réPLICATION : streaming indique que le flux est fonctionnel, catchup indique que le serveur secondaire essaie de rattraper son retard, etc. sync\_state peut avoir trois valeurs : async dans le cas d'une réPLICATION ASYNCHRONE, sync et potential dans le cas d'une réPLICATION SYNCHRONE.

Le retard d'un serveur secondaire sur son primaire est toujours exprimé en octets, ce qui n'est pas simple à appréhender si on veut savoir si un serveur secondaire est trop en retard par rapport au primaire. 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 d'un serveur secondaire sous la forme d'une durée. Attention, cela ne fonctionne que si le primaire est actif. En effet, si aucune requête ne s'exécute sur le serveur primaire, le calcul now() - pg\_last\_xact\_replay\_timestamp() aura pour résultat une durée croissante dans le temps, même si le serveur secondaire est identique au serveur primaire (un peu comme Slony et son pseudo lag quand les événements de synchronisation ne passent plus).

Sur le secondaire, on peut consulter aussi la vue pg\_stat\_wal\_receiver pour voir la connexion en cours :

```
postgres=# SELECT * FROM pg_stat_wal_receiver;
```

```
-[RECORD 1]-----+-----
pid | 16789
status | streaming
receive_start_lsn | 7/BA0000000
receive_start_tli | 4
received_lsn | 7/C564A580
received_tli | 4
last_msg_send_time | 2019-01-21 16:49:42.867596+01
last_msg_receipt_time | 2019-01-21 16:49:42.86786+01
latest_end_lsn | 7/C564A580
latest_end_time | 2019-01-21 16:49:42.867596+01
slot_name | abeille2
conninfo | user=replicator passfile=/var/lib/postgresql/.pgpass
dbname=replication host=localhost port=10021
application_name=sec2 fallback_application_name=walreceiver
↳ ...
```

Pour les réPLICATIONS SYNCHRONES, la version 10 introduit trois nouvelles colonnes :

---

<sup>1</sup>[https://github.com/OPMDG/check\\_pgactivity](https://github.com/OPMDG/check_pgactivity)

```
postgres=# SELECT write_lag,flush_lag,replay_lag FROM pg_stat_replication;
-[RECORD 1]-----
write_lag | 00:00:00.000622
flush_lag | 00:00:00.001785
replay_lag| 00:00:53.239083
```

`write_lag` mesure le délai si `synchronous_commit` vaut `remote_write`, c'est-à-dire que chaque COMMIT attend la confirmation de la réception en mémoire de l'enregistrement du COMMIT par le *standby* et l'écriture dans le cache de son système d'exploitation, 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* (`synchronous_commit = on`).

`replay_lag` mesure le délai pour le cas où `synchronous_commit` vaut `remote_apply`, où COMMIT attend que les serveurs secondaires synchrones ont bien rejoué la transaction, la rendant visible aux requêtes des utilisateurs.

Enfin, pour savoir où en sont les serveurs secondaires, éventuellement déconnectés, utilisant un slot de réPLICATION, consulter aussi la vue `pg_replication_slots`:

```
postgres=# 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|
-[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|
```

---

## 6.3 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<sup>2</sup> 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.

---

<sup>2</sup>[https://github.com/OPMDG/check\\_pgactivity](https://github.com/OPMDG/check_pgactivity)

## 6.4 GESTION DES CONFLITS



- Une requête en lecture pose des verrous
  - conflit possible avec changements répliqués !
- Vue pg\_stat\_database\_conflicts (secondaires)
- Traces :
  - log\_recovery\_conflict\_waits (v14+)
- Pour éviter le conflit
  - vacuum\_defer\_cleanup\_age
  - max\_standby\_archive\_delay/wal\_standby\_streaming\_delay
  - hot\_standby\_feedback + wal\_receiver\_status\_interval (10s)
  - gênent le vacuum !

### Source des conflits :

Un conflit peut survenir entre l'application des modifications de la réPLICATION et la connexion sur une base d'un serveur secondaire ou sur l'exécution d'une requête en lecture seule. Comme les modifications de la réPLICATION doivent s'enregistrer dans l'ordre de leur émission, si une requête bloque l'application d'une modification, elle bloque en fait l'application de toutes les modifications suivantes pour 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 du catalogue pg\_stat\_database\_conflicts 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 :

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

```
confl_snapshot | 3
confl_bufferpin | 2
confl_deadlock | 0
```

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.

#### Gestion fine des délais pour réduire les conflits :

Sur les secondaires, `max_standby_archive_delay` et `max_standby_streaming_delay` sont des délais d'attente avant d'appliquer un journal provenant respectivement du *log shipping* ou du *streaming*, journal qui entre en conflit avec des requêtes en cours sur le secondaire. 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.

Du côté primaire, pour limiter le problème au moins pour les VACUUM, `vacuum_defer_cleanup_age` permet de fixer un certain nombre de transactions en écriture avant de recycler des lignes mortes... ce qui est très délicat à calculer et peut impacter certaines opérations. Par défaut, la valeur est 0 et le mécanisme désactivé.

#### 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` (le nom est le même avant la v10), soit 10 s 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`.

---

## 6.5 CONTRÔLE DE LA RÉPLICATION



- `pg_wal_replay_pause()` : mettre en pause le **rejet**
- `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.

(De la 9.1 à la 9.6, ce sont : `pg_xlog_replay_pause()`, `pg_xlog_replay_resume()`, `pg_is_xlog_replay_paused()`.)

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 *rejet* 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.

## 6.6 ASYNCHRONE OU SYNCHRONE



- Par défaut : réPLICATION physique **asynchrone**
- Synchrone avec **synchronous\_standby\_names**
- Plusieurs synchrones simultanés possibles
  - ou un quorum (v 10)
- Mode exact :
  - niveau de synchronicité :

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

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

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

En réPLICATION **asynchrone**, 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 a indiqué une transaction comme enregistrée mais qu'après le *failover*, elle ne soit plus disponible.

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 inconvenients.

Par défaut, la réPLICATION fonctionne en asynchrone. La mise en place d'un mode synchrone est très simple : `synchronous_commit` est déjà à `on` par défaut, donc il ne reste qu'à déclarer les serveurs secondaires synchrones dans `synchronous_standby_names`, en les séparant par des virgules. L'ordre des serveurs secondaires a un sens : le premier secondaire cité sera de priorité 1, le deuxième de priorité 2, etc.

Il est possible d'indiquer le nombre de serveurs synchrones simultanés. (depuis la version 9.6 ; auparavant seul un secondaire était considéré comme synchrone, les autres de la liste n'étant que des remplaçants possibles en cas de défaillance du premier.) Les serveurs surnuméraires sont des synchrones potentiels. Si ce nombre est indiqué, la liste des serveurs est mis entre parenthèses, comme ceci :

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

La version 10 introduit le mot clé FIRST pour la même fonctionnalité :

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

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

Il est à noter que les syntaxes suivantes sont équivalentes :

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

À partir de la version 10, il est possible de se baser sur un quorum pour choisir les serveurs synchrones. 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)'
```

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, utilisé ici pour distinguer les serveurs secondaires. Il doit donc apparaître dans la chaîne de connexion du serveur secondaire au serveur primaire, c'est-à-dire `primary_conninfo` dans le fichier `recovery.conf`. Par exemple :

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

Pour définir le mode de fonctionnement exact, `synchronous_commit` peut prendre plusieurs valeurs :

**off :**

La transaction est directement validée, mais elle pourra être écrite plus tard dans les journaux. Le délai est d'au maximum 3 fois la valeur de `wal_writer_delay` (soit au total 0,6 s par défaut).



Ce paramétrage peut causer la perte des transactions non encore écrites dans les journaux si le serveur se crashe.

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.

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

**remote\_write :**

Le primaire 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.

Cela permet de retrouver beaucoup de la performance perdue avec la valeur `on`, et la fenêtre de perte de données est bien moins importante que le mode asynchrone, mais toujours présente.

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

L'instance primaire ne s'assure pas non plus que le secondaire a **rejoué** la transaction. Le rejet 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.

**on** (défaut) :

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

S'il y a des secondaires synchrônes, la transaction est validée après écriture et synchronisation sur les disques 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, machines).

Le primaire n'attend pas que le secondaire ait rejoué les données pour valider la transaction. Là encore, deux requêtes exécutées au même moment sur le primaire et un secondaire peuvent renvoyer des résultats différents.

**remote\_apply** (apparu en version 9.6) :

C'est le mode de synchronisation le plus poussé. Non seulement les modifications doivent être enregistrées dans les journaux du secondaire, et synchronisées sur le disque, mais elles doivent aussi être rejouées pour que la transaction soit validée.

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 tarde, 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 ! Ce cas est rare car une réPLICATION synchrone est à éviter sur une liaison instable.

**Tableau récapitulatif :**

**Table 6/ .1:** Délai de synchronisation des données vers le disque ou différents supports d'un secondaire, après validation d'une transaction, selon le paramètre `synchronous_commit`

|                     | Sur disque local | En mémoire distante | Sur disque distant | Rejou distant |
|---------------------|------------------|---------------------|--------------------|---------------|
| <b>off</b>          | <0,6 s (défaut)  | Variable            | Variable           | Variable      |
| <b>local</b>        | Au COMMIT        | Variable            | Variable           | Variable      |
| <b>remote_write</b> | Au COMMIT        | Au COMMIT           | Variable           | Variable      |
| <b>on</b>           | Au COMMIT        | Au COMMIT           | Au COMMIT          | Variable      |

`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 local ; -- dans la session
SET LOCAL synchronous_commit TO local ; -- 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 voir quel est 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.

Il faut savoir qu'en cas d'indisponibilité du secondaire, 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 :

```
ALTER SYSTEM SET synchronous_standby_names TO '';
SELECT pg_reload_conf();
```

Une alternative est de passer `synchronous_commit` à `local` de la même manière. Cependant, cela désactivera totalement le mode synchrone vers les secondaires encore en place.

Il faudra aussi envoyer un signal avec `pg_cancel_backend` à toutes les sessions bloquées pour qu'elles se terminent (elles seront bien validées, dans les journaux du primaire au moins).

## 6.7 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.

---

## 6.8 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` 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 paramétrier `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.

(Jusqu'à PostgreSQL 12 inclus, ce paramètre se nommait `wal_keep_segments` et s'exprimait en journaux, soit 1000 pour transposer l'exemple ci-dessus avec les journaux de 16 Mo par défaut).

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.

---

### 6.8.1 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<sup>3</sup>. La `restore_command` est ignorée si le secondaire a rebascué 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` (fichier `recovery.conf` jusqu'en v11). 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`<sup>4</sup>:

```
archive_cleanup_command = '/usr/pgsql-14/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<sup>5</sup> : `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` !

---

<sup>3</sup>[https://dali.bo/i2\\_html](https://dali.bo/i2_html)

<sup>4</sup><https://docs.postgresql.fr/current/pgarchivecleanup.html>

<sup>5</sup>[https://dali.bo/i4\\_html](https://dali.bo/i4_html)

### 6.8.2 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 (ou recovery.conf si <v12)  
`primary_slot_name = 'nomsecondaire'`
  - redémarrage de l'instance (<v13) ou rechargement (v13+)

Le paramètre `max_replication_slots` doit être supérieur à 0. À partir de la version 10, il est par défaut à 10. 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` (ou `recovery.conf` avant la version 12) 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.

### 6.8.3 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 géné
  - 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 :

```
postgres=# SELECT * FROM pg_replication_slots ;
```

```
-[RECORD 1]+-----+
slot_name | defo2
plugin | x
slot_type | physical
datoid | x
database | x
active | t
active_pid | 30708
xmin | 1262
catalog_xmin | x
restart_lsn | 2/55575F28
```

#### 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. En version 11, pg\_basebackup est capable de créer ce slot (option --create-slot) qui sera conservé ensuite. Avant la version 11, il faut créer préalablement le slot dédié au futur secondaire que l'on est en train de construire et lui indiquer avec l'option -S. La version 10 crée au besoin un slot temporaire, mais il reste un risque de décrochage entre la fin de la copie et le lancement du nouveau secondaire.

#### 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.

#### 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.

## 6.9 SYNTHÈSE DES PARAMÈTRES

### 6.9.1 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_connections**        |
|                          | max_replication_slots*   |
| vacuum_defer_cleanup_age | vacuum_defer_cleanup_age |
|                          | wal_keep_size            |
|                          | max_slot_wal_keep_size   |

(\*) paramètres utiles mais pas indispensables.

(\*\*) avant la version 12, max\_wal\_senders compte dans max\_connections.

---

### 6.9.2 Serveur secondaire

| Log shipping              | Streaming                    |
|---------------------------|------------------------------|
| wal_level = replica       | wal_level = replica          |
| standby_mode (<=v11)      | standby_mode (<=v11)         |
| restore_command           |                              |
| archive_cleanup_command*  | primary_conninfo             |
|                           | wal_receiver_timeout         |
|                           | hot_standby*                 |
|                           | promote_trigger_file*        |
|                           | primary_slot_name*           |
| max_standby_archive_delay | max_standby_streaming_delay  |
|                           | hot_standby_feedback*        |
|                           | wal_receiver_status_interval |

(\*) paramètres utiles mais pas indispensables.

---

## 6.10 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.

---

### 6.10.1 Questions



N'hésitez pas, c'est le moment !

---

## 6.11 QUIZ



[https://dali.bo/w2b\\_quiz](https://dali.bo/w2b_quiz)

## 6.12 TRAVAUX PRATIQUES

### 6.12.1 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.

### 6.12.2 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.

### 6.12.3 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 ?

### 6.12.4 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)`

#### 6.12.5 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 rejet 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 rejet puis insertion d'enregistrements sur le principal). Que se passe-t-il ?
-

## 6.13 TRAVAUX PRATIQUES (SOLUTIONS)

### 6.13.1 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 aussi nécessaire de modifier le numéro de port d'écoute 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 000000030000000000000000E    |
| 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 000000030000000000000000E    |
| 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.

### 6.13.2 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);"
```

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 00000003000000000000000000E
-rw----- 1 postgres postgres 16M Nov 27 16:40 00000003000000000000000000F
-rw----- 1 postgres postgres 16M Nov 27 16:40 0000000300000000000000000010
-rw----- 1 postgres postgres 16M Nov 27 16:40 0000000300000000000000000011
-rw----- 1 postgres postgres 16M Nov 27 16:40 0000000200000000000000000012
-rw----- 1 postgres postgres 16M Nov 27 16:40 0000000200000000000000000013
```

(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 `000000030000000000000000E`) 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
username | 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
username | 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 -ahl /var/lib/pgsql/14/instance1/pg_wal/
(...)
-rw----- 1 postgres postgres 16M Nov 27 16:44 000000030000000000000000000000013
-rw----- 1 postgres postgres 16M Nov 27 16:40 000000030000000000000000000000014
-rw----- 1 postgres postgres 16M Nov 27 16:40 000000030000000000000000000000015
-rw----- 1 postgres postgres 16M Nov 27 16:40 000000030000000000000000000000016
-rw----- 1 postgres postgres 16M Nov 27 16:41 000000030000000000000000000000017
-rw----- 1 postgres postgres 16M Nov 27 16:41 000000030000000000000000000000018
-rw----- 1 postgres postgres 16M Nov 27 16:41 000000030000000000000000000000019
```

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');"
```

### 6.13.3 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 00000003000000000000000013
```

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 la 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);"
$ 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 0000000300000000000000000000001B
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001C
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001D
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001E
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001F
-rw-----. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000000000000020
-rw-----. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000000000000021
-rw-----. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000000000000022
-rw-----. 1 postgres postgres 16M Nov 27 17:21 00000003000000000000000000000023

$ ls -alh /var/lib/pgsql/14/archives/
(...)
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001B
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001C
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001D
-rw-----. 1 postgres postgres 16M Nov 27 17:21 0000000300000000000000000000001E
```

- 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 "0000000300000000000000000000001B" from archive
LOG: restored log file "0000000300000000000000000000001C" from archive
LOG: restored log file "0000000300000000000000000000001D" from archive
LOG: restored log file "0000000300000000000000000000001E" from archive
cp: cannot stat '/var/lib/pgsql/14/archives/0000000300000000000000000000001F':
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);"
$ 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//00000003000000000000000000000029" and later
pg_archivecleanup: removing file
 "/var/lib/pgsql/14/archives//00000003000000000000000000000013"
pg_archivecleanup: removing file
 "/var/lib/pgsql/14/archives//00000003000000000000000000000014"
(...)
```

- 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).

#### 6.13.4 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 krb_srvname=postgres
 ↳ target_session_attrs=any'
```

Il est aussi nécessaire de modifier le numéro de port d'écoute de l'instance dans `/var/lib/pgsql/14/instance4`:

```
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 initialisé par le paramètre `synchronous_standby_names` du 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);"
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

```

### 6.13.5 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 rejet 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 rechargeement 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)

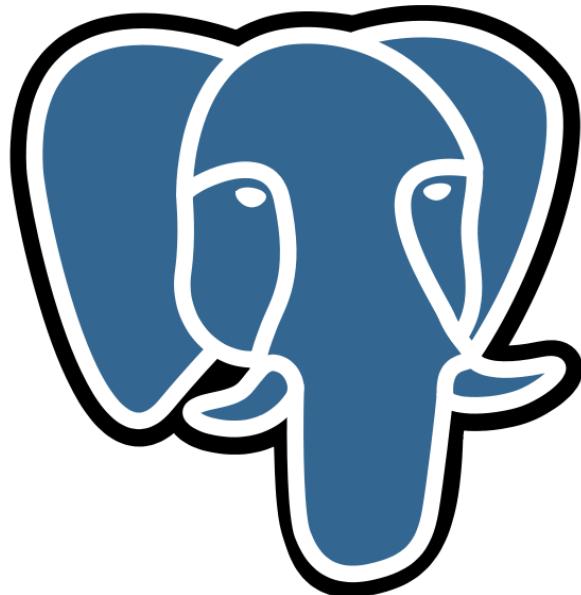
$ psql -p 5432 -c "INSERT INTO t4 VALUES ('2');"
```

Cette fois la requête est bloquée, il faut relancer le rejet sur au moins deux secondaires pour qu'elle puisse s'effectuer puisque nous avons configuré.

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



## 7/ Les outils de réPLICATION



## 7.1 INTRODUCTION



- Les outils à la rescouasse !
  - (Re)construction d'un secondaire
  - Log shipping & PITR
  - Promotion automatique

Nous aborderons dans ce module différents outils externes qui peuvent nous aider à administrer notre cluster de réPLICATION.

### 7.1.1 Au menu



- (Re)construction d'un secondaire
  - outils de copie
  - pg\_rewind
- Log shipping & PITR
  - pgBackRest
  - Barman
- Promotion automatique
  - Patroni
  - repmgr
  - PAF

## 7.2 (RE)CONSTRUIRE UN SECONDAIRE



- pg\_basebackup
- rsync
- outils PITR
- pg\_rewind

### 7.2.1 (Re)construction d'un secondaire : pg\_basebackup



- Simple et pratique
- ...mais recopie tout !

pg\_basebackup est un outil éprouvé pour créer une sauvegarde physique, mais aussi un serveur *standby*. Il est basé sur une réPLICATION par *streaming*.

Rappelons ses qualités et défauts dans le cadre de la réPLICATION :

- il sait générer le `postgresql.auto.conf` et le `standby.signal` (ou le `recovery.conf` avant la v12) pour créer un secondaire en *streaming* (option `--write-recovery-conf`) ;
- l'impact sur le réseau peut être limité grâce à l'option `-r` (`--max-rate`) ;
- il peut poser ou utiliser un slot de réPLICATION permanent pour éviter un décrochage dès le dé-marrage ;
- il peut déplacer des tablespaces à un autre endroit (*mapping*) ;
- les journaux nécessaires peuvent être inclus et leur récupération sécurisée par un slot de réPLICATION temporaire (c'est en général inutile pour (re)construire un secondaire).

Par contre :

- pg\_basebackup s'attend à créer un *standby* de zéro : le réPERTOIRE de données de destination doit être vide, réPERTOIRES des tablespaces compris : un ancien secondaire doit donc être détruit (penser à sauver la configuration !) ;
- il copiera à nouveau toutes les données, même si un ancien secondaire ou un ancien primaire presque identique existe ;
- si la connexion est peu stable, le transfert échouera et sera à reprendre de zéro ;
- les données transférées ne sont pas compressées avant la version 15 de PostgreSQL.

**(Re)construction :**

La sauvegarde générée, et si nécessaire décompressée, ne peut être utilisée directement pour créer un instance secondaire.

Il faut ajouter le fichier `standby.signal` dans le PGDATA, préciser comment récupérer les journaux (`restore_command`) et/ou paramétrer le `streaming` (`primary_conninfo` et `primary_slot_name`) dans `postgresql.conf` ou `postgresql.auto.conf`. (Jusqu'en version 11, tout cela va dans le fichier `recovery.conf`; un modèle tout prêt sous un autre nom peut être déposé dans le PGDATA du primaire).

Pour simplifier, `pg_basebackup` peut générer la configuration nécessaire dans le répertoire de la sauvegarde. Par exemple, il ajoutera ceci dans le `postgresql.auto.conf`:

```
primary_conninfo = 'user=postgres passfile='' /var/lib/postgresql/.pgpass ''
channel_binding=prefer
host=localhost port=5432
sslmode=prefer sslcompression=0 sslnsi=1
ssl_min_protocol_version=TLSv1.2
gssencmode=prefer krb_srvname=postgres
target_session_attrs=any'
```

Cette chaîne réutilise la chaîne fournie en appelant `pg_basebackup`, essaie de prévoir tous les types d'authentification, et positionne nombre de paramètres à leur valeur par défaut. Le plus souvent, utilisateur, port, hôte et `passfile` suffisent.

Si `--slot=nom_du_slot` a été précisé, apparaîtra aussi :

```
primary_slot_name = 'nom_du_slot'
```

Le secondaire possède ainsi immédiatement toutes les informations pour se connecter au primaire. Il est tout de même conseillé de vérifier cette configuration et d'y ajouter la commande de récupération des archives si nécessaire.

### 7.2.2 (Re)construction d'un secondaire : script rsync



- Intérêts :
  - reprendre sur de l'existant
  - reprendre des transferts interrompus
  - compression
- Prévoir :
  - `pg_backup_stop()`/`pg_backup_start ()`
  - `rsync --whole-file`
  - les tablespaces
  - ne pas tout copier !
  - `postgresql.conf` (ou `recovery.conf` si  $\leq v11$ )

Selon les volumes de données mis en jeu, et encore plus avec une liaison instable, il est souvent plus intéressant d'utiliser rsync. En effet, rsync ne transfère que les fichiers ayant subi une modification. Le transfert sera beaucoup plus rapide s'il existe un secondaire, ou ancien primaire qui a « décroché ». Si le transfert a été coupé, rsync permet de le reprendre.

C'est évidemment plus fastidieux qu'un pg\_basebackup direct, mais peut valoir le coup pour les grosses installations.

Voici un exemple d'utilisation :

```
rsync -av -e 'ssh -o Compression=no' --whole-file --ignore-errors \
--delete-before --exclude 'lost+found' --exclude 'pg_wal/*' \
--compress --compress-level=7 \
--exclude='*.pid' $PRIMARY:$PGDATA/ $PGDATA/
```

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

Lorsque la connexion utilisée est lente, il est courant de compresser les données pour le transfert (options -z/--compress et --compress-level, de 1 à 9), à ajuster en fonction du CPU disponible.

L'option --bwlimit limite au besoin le débit réseau.

Il faudra impérativement encadrer l'appel à rsync de pg\_backup\_start() et pg\_backup\_stop(), comme dans une sauvegarde PITR classique.

De nombreux fichiers ne doivent **pas** être copiés. La liste complète figure dans le chapitre sur la sauvegarde PITR<sup>1</sup> ou la documentation officielle<sup>2</sup>. Les principaux sont postmaster.pid, pg\_wal et ses sous-réertoires, pg\_replslot, pg\_dynshmem, pg\_notify, pg\_serial, pg\_snapshots, pg\_stat\_tmp et pg\_subtrans, pgsql\_tmp\*.

Pour créer le fichier postgresql.auto.conf (ou recovery.conf), on peut utiliser un fichier modèle tout prêt dans le répertoire PGDATA du serveur principal.

La liste des réertoires des tablespaces se trouve dans \$PGDATA/pg\_tblspc sous forme de liens symboliques pointant sur le répertoire du tablespace : on peut alors rendre le script générique pour la synchronisation des tablespaces.

Et il faudra bien tester !

---

<sup>1</sup>[https://dali.bo/i2\\_html](https://dali.bo/i2_html)

<sup>2</sup><https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP>

### 7.2.3 (Re)construction d'un secondaire : outil PITR



- Le plus confortable
- Ne charge pas le primaire
- Mode « delta »

```
pgbackrest --stanza=instance --type=standby --delta \
--repo1-host=depot --repo1-host-user=postgres --repo1-host-port=22 \
--pg1-path=/var/lib/postgresql/14/secondaire \
--recovery-option=primary_conninfo='host=principal port=5433
↪ user=replicator' \
--recovery-option=primary_slot_name='secondaire' \
--target-timeline=latest \
restore
```

Si l'on dispose d'un outil PITR, il permet souvent de créer un secondaire, et c'est généralement l'option la plus confortable.

L'exemple ci-dessus utilise pgBackRest pour créer un serveur secondaire dans le répertoire pointé. Les outils concurrents suivent le même principe.

Si le répertoire cible n'est pas vide (secondaire décroché, ancien primaire, restauration échouée...), le paramètre `--delta` permet de ne copier que les différences entre la sauvegarde et le répertoire existant. Cela permet de gagner un temps précieux.

Pour simplifier, une bonne partie de ces options peuvent être définies dans le fichier de configuration local `pgbackrest.conf`.

Les options de `recovery` demandées dans cet exemple apparaîtront dans le `postgresql.auto.conf` (ou `recovery.conf` avant la version 12), ainsi que la `restore_command` nécessaire pour récupérer les journaux :

```
standby_mode = on # avant PG 12
primary_conninfo = 'host=principal port=5433 user=replicator'
primary_slot_name = 'secondaire'
recovery_target_timeline = 'latest'
restore_command = 'pgbackrest --pg1-path=/var/lib/postgresql/14/secondaire
 --repo1-host=depot --repo1-host-port=22 --repo1-host-user=postgres
 --stanza=instance archive-get %f "%p"'
```

Un autre intérêt de créer un secondaire depuis une sauvegarde PITR est de ne pas charger le serveur primaire : les fichiers sont copiés depuis le dépôt.

Il y a un petit inconvénient : la sauvegarde peut dater de plusieurs jours, donc il y aura plus de journaux à restaurer qu'en faisant une nouvelle copie du primaire.

## 7.2.4 pg\_rewind



- Outil inclus
- Évite la reconstruction complète malgré une divergence
- Pré-requis :
  - `data-checksums` (`initdb` à froid)
  - ou `wal_log_hints = on`
  - tous les WAL depuis la divergence
  - `full_page_writes = on` (activé par défaut)
- Revient au point de divergence

Le cas typique d'utilisation de `pg_rewind` est de résoudre la divergence entre un ancien primaire et le nouveau primaire après une bascule. Des informations ont été écrites sur l'ancien primaire qui ne sont pas sur le nouveau, et donc empêchent de le raccrocher simplement comme secondaire. Ou encore, un secondaire a été ouvert brièvement en écriture par erreur.

`pg_rewind` nécessite soit la présence des *checksums* (qui ne peuvent être mises en place qu'au moment de la création de l'instance, ou ajoutés à froid à partir de la version 12), soit l'activation du paramètre `wal_log_hints`, et tout cela suffisamment en avance avant la divergence. La volumétrie des journaux de transactions augmentera un peu. `full_page_writes` est aussi nécessaire mais activé par défaut.

Le serveur à récupérer doit être arrêté proprement. Avec PostgreSQL 13, si ce n'est pas le cas, `pg_rewind` le démarre en mode mono utilisateur puis l'arrête.

Le serveur source (le nouveau primaire) doit être aussi arrêté proprement sur le même serveur, ou accessible par une connexion classique (l'alias `replication` du fichier `pg_hba.conf` ne suffit pas).

L'utilisateur requis pour se connecter à la nouvelle primaire doit avoir l'attribut SUPERUSER. Depuis la version 11, un utilisateur ayant le droit de se connecter (LOGIN) et d'exécuter les fonctions suivantes suffit : `pg_ls_dir()`, `pg_stat_file()`, `pg_read_binary_file(text)` et `pg_read_binary_file(text, bigint, bigint, boolean)`.

Sur l'instance à récupérer, `pg_rewind` analyse les journaux de transactions depuis le checkpoint précédent la divergence et jusqu'au moment où le serveur a été arrêté pour y récupérer la liste des blocs modifiés. Ces journaux doivent se trouver dans le répertoire `pg_wal` de l'instance. La version 13 permet de s'affranchir de cette limitation en permettant de spécifier une commande de restauration pour récupérer les journaux manquants.

Il récupère ensuite de la nouvelle instance primaire tous les blocs dont la liste a été générée précédemment. Puis, il copie tous les autres fichiers, ce qui inclut ceux des nouvelles relations, les WAL, le contenu du répertoire `pg_xact` et les fichiers de configurations présents dans le répertoire de données. Seuls les fichiers habituellement exclus des sauvegardes ne sont pas récupérés.

Pour finir, il crée un fichier `backup_label` et met à jour le LSN du dernier point de cohérence dans le *control file*.

Une fois l'opération réalisée, l'instance est redémarrée et se met à rejouer tous les journaux depuis le point du dernier checkpoint précédent le point de divergence.

## 7.3 LOG SHIPPING & PITR



- Utilisation des archives pour...
  - se prémunir du décrochage d'un secondaire
  - une sauvegarde PITR
- Utilisation des sauvegardes PITR pour...
  - resynchroniser un serveur secondaire

Comparé au slot de réPLICATION, le *log shipping* a l'avantage de pouvoir être utilisé pour réaliser des sauvegardes de type PITR. De plus, il est alors possible de se servir de ces sauvegardes pour reconstruire un serveur secondaire plutôt que de réaliser la copie directement depuis l'instance principale.

### 7.3.1 pgBackRest



- `pgbackrest restore`
  - `--delta`
  - `--type=standby`
  - `--recovery-option`

Il est possible d'utiliser certaines options (en ligne de commande ou fichier de configuration) de l'outil pgBackRest pour resynchroniser une instance secondaire à partir d'une sauvegarde PITR.

Par exemple :

- `--delta` : ne recopiera que les fichiers dont la somme de contrôle (checksum) est différente entre la sauvegarde et le répertoire de destination, les fichiers qui ne sont pas présents dans la sauvegarde sont supprimés du répertoire de l'instance ;
- `--type=standby` : ajoute `standby_mode=on` et la `restore_command` au `recovery.conf` (< v12) ou crée le fichier `standby.signal` et ajoute la `restore_command` au fichier `postgresql.auto.conf` ;
- `--recovery-option` : permet de spécifier des paramètres particuliers pour la configuration de la restauration (`recovery_target_*`,...) ou de la réPLICATION (`primary_conninfo`,...).

### 7.3.2 barman



- `barman recover`
  - `--standby-mode`
  - `--target-*`

Il est possible d'utiliser certaines options de l'outil Barman pour resynchroniser une instance secondaire à partir d'une sauvegarde PITR.

Par exemple :

- `--standby-mode` : ajoute `standby_mode=on` au `recovery.conf` (< v12) ou crée le fichier `standby.signal`. Suivant le type de restauration la `restore_command` sera également ajoutée à la configuration dans le `recovery.conf` (< v12) ou le fichier `postgresql.auto.conf`;
- `--target-*` : permettent de spécifier des paramètres particuliers pour la configuration de la restauration (`recovery_target_*`,...).

## 7.4 PROMOTION AUTOMATIQUE



- L'objectif est de minimiser le temps d'interruption du service en cas d'avarie
- Un peu de vocabulaire :
  - *SPOF* : Single Point Of Failure
  - *Redondance* : pour éviter les SPOF
  - *Ressources* : éléments gérés par un cluster
  - *Split brain* : deux primaires ! et perte de données
  - *Fencing* : isoler un serveur défaillant
  - *STONITH* : Shoot The Other Node In The Head (voir *Fencing*)
  - *Watchdog* : permet à un serveur de s'auto isoler
  - *Quorum* : participe à la résolution des partitions réseau

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

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

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

Les éléments de la plate-forme, aussi matériels que logiciels, dont la défaillance mène à l'indisponibilité du service, sont appelés *SPOF* dans le jargon de la haute disponibilité, ce qui signifie *Single Point Of Failure* ou point individuel de défaillance. L'objectif de la mise en haute disponibilité est d'éliminer ces SPOF. Ce travail concerne le service dans sa globalité : si le serveur applicatif n'est pas hautement-disponible alors que la base de données l'est, le service peut être interrompu car le serveur applicatif est un SPOF. On élimine les différents SPOF possibles par la *redondance*, à la fois matérielle, logicielle mais aussi humaine en évitant qu'une personne détienne toute la connaissance sur un sujet clé.

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

On peut se trouver dans une situation où les deux machines considèrent qu'elles sont le seul primaire de leur cluster (*Split-Brain*, avec écriture de données sur les deux serveurs simultanément, et la perspective d'une réconciliation délicate) ; ou bien entrent en concurrence pour un accès en écriture sur un même disque partagé.

Pour se protéger d'une partition réseau, on peut utiliser un dispositif implémentant un *Quorum*. Chaque machine se voit allouer un poids qui sera utilisé lors des votes. En cas de coupure réseau, la

partition possédant le plus de votes a le droit de conserver les ressources actives, la ou les autres partitions doivent relâcher les ressources ou les éteindre.

Pour éviter d'avoir des serveurs ou ressources dont l'état est inconnu ou incohérent (serveur injougnable ou ressource qui refuse de s'arrêter), on utilise le *Fencing*. Cette technique permet d'isoler une machine du cluster (*Noeud*) en lui interdisant l'accès à certaines ressources ou en l'arrêtant. Cette deuxième solution est plus connue sous le nom de *STONITH* (*Shoot The Other Node In The Head*). Les techniques pour arriver à ce résultat varient : de la coupure de l'alimentation (PDU, IPMI), à la demande faite au superviseur d'arrêter une machine virtuelle. Une alternative est l'utilisation d'un *Watchdog*. Ce dispositif arrête le serveur s'il n'est pas réarmé à intervalle régulier, ce qui permet à un serveur de s'isoler lui-même.

Pour finir, il est possible d'utiliser un dispositif appelé SBD (*Storage Based Death*) qui combine ces concepts et outils (*fencing* et *watchdog*). Un stockage partagé permet aux serveurs de communiquer pour signaler leur présence ou demander l'arrêt d'un serveur. Si un serveur n'a plus accès au disque partagé, il doit arrêter ses ressources ou s'arrêter si c'est impossible.

Une solution de haute disponibilité robuste combine ces mécanismes pour un maximum de fiabilité.

#### 7.4.1 Patroni



- Outil de HA
- Basé sur un gestionnaire de configuration distribué : etcd, Consul, ZooKeeper...
- Contexte physique, VM ou container
- Spilo : image docker PostgreSQL+Patroni

Patroni<sup>3</sup> est un outil de HA développé par Zalando, nécessitant un gestionnaire de configuration distribué (appelé DCS) tel que ZooKeeper, etcd ou Consul.

Le daemon Patroni fait le lien entre le quorum fourni par le DCS, l'éventuelle utilisation d'un module watchdog, la communication avec les autres noeuds Patroni du cluster et la gestion complète de l'instance PostgreSQL locale : de son démarrage à son arrêt.

Le watchdog est un mécanisme permettant de redémarrer le serveur en cas de défaillance. Il est ici utilisé pour redémarrer si le processus Patroni n'est plus disponible (crash, freeze, bug, etc).

Zalando utilise Patroni principalement dans un environnement conteneurisé. À cet effet, ils ont packagé PostgreSQL et Patroni dans une image docker: Spilo<sup>4</sup>.

Cependant, Patroni peut tout à fait être déployé en dehors d'un conteneur, sur une machine physique ou virtuelle.

<sup>3</sup><https://github.com/zalando/patroni>

<sup>4</sup><https://github.com/zalando/spilo>

#### 7.4.2 repmgr



- Outil spécialisé PostgreSQL
- En pratique, fiable pour 2 nœuds
- Gère automatiquement la bascule en cas de problème
  - *health check*
  - *failover et switchover automatiques*
- *Witness*

L'outil repmgr<sup>5</sup> de 2ndQuadrant permet la gestion de la haute disponibilité avec notamment la gestion des opérations de clonage, de promotion d'une instance en primaire et la démotion d'une instance.

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

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

#### 7.4.3 Pacemaker



- Solution de Haute Disponibilité généraliste
- Disponible sur les distributions les plus répandues
- Se base sur Corosync, un service de messagerie inter-nœuds
- Permet de surveiller la disponibilité des machines
- Gère le quorum, le fencing, le watchdog et le SBD
- Gère les ressources d'un cluster et leur interdépendance
- Extensible

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

<sup>5</sup><https://www.repmgr.org/>

extensible avec des scripts.

#### 7.4.4 PAF



- **Postgres Automatic Failover**
- Ressource Agent pour Pacemaker et Corosync permettant de :
  - détecter un incident
  - relancer l'instance primaire
  - basculer sur un autre nœud en cas d'échec de relance
  - élire le meilleur secondaire (avec le retard le plus faible)
  - basculer les rôles au sein du cluster en primaire et secondaire
- Avec les fonctionnalités offertes par Pacemaker & Corosync :
  - surveillance de la disponibilité du service
  - quorum & Fencing
  - gestion des ressources du cluster

PAF est le fruit de la R&D de Dalibo visant à combler les lacunes des agents existants. Il s'agit d'un produit opensource, disponible sur ce dépôt<sup>6</sup> et qui a rejoint la communauté ClusterLabs qui gère aussi Pacemaker.

---

<sup>6</sup><https://clusterlabs.github.io/PAF/>

## 7.5 CONCLUSION



De nombreuses applications tierces peuvent nous aider à administrer efficacement un cluster en réPLICATION.

### 7.5.1 Questions



N'hésitez pas, c'est le moment !

## 7.6 TRAVAUX PRATIQUES



Pour ces travaux pratiques, procéder à :

- la promotion de l'instance **instance2** ;
- la reconstruction de l'ancienne instance primaire via pg\_rewind ;
- la sauvegarde de l'instance **instance2** avec pgBackRest ;
- la promotion “par erreur” de l'instance **instance3** ;
- la reconstruction de l'instance **instance3** en se servant des sauvegardes pgBackRest.

### 7.6.1 (Pré-requis) Environnement

Pour ces TP vous devez avoir 3 instances (1 primaire, 2 secondaires en réPLICATION). Au besoin, repasser en réPLICATION ASYNCHRONE en désactivant le paramètre synchronous\_standby\_names de l'instance primaire.

```
$ ps -o pid,cmd fx

PID CMD
3205 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance3/
3207 _ postgres: logger
3208 _ postgres: startup recovering 0000000300000000000000038
3212 _ postgres: checkpointer
3213 _ postgres: background writer
3215 _ postgres: stats collector
3216 _ postgres: walreceiver streaming 0/38000060
3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
3146 _ postgres: logger
3147 _ postgres: startup recovering 0000000300000000000000038
3151 _ postgres: checkpointer
3152 _ postgres: background writer
3154 _ postgres: stats collector
3155 _ postgres: walreceiver streaming 0/38000060
2896 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
2898 _ postgres: logger
2900 _ postgres: checkpointer
2901 _ postgres: background writer
2902 _ postgres: walwriter
2903 _ postgres: autovacuum launcher
2904 _ postgres: archiver last was 000000030000000000000037.00000028.backup
2905 _ postgres: stats collector
2906 _ postgres: logical replication launcher
3156 _ postgres: walsender repli 127.0.0.1(47494) streaming 0/38000060
3217 _ postgres: walsender repli 127.0.0.1(47502) streaming 0/38000060

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

```
-[RECORD 1]-----+
application_name | instance2
state | streaming
sent_lsn | 0/38000060
write_lsn | 0/38000060
flush_lsn | 0/38000060
replay_lsn | 0/38000060
sync_state | async
-[RECORD 2]-----+
application_name | instance3
state | streaming
sent_lsn | 0/38000060
write_lsn | 0/38000060
flush_lsn | 0/38000060
replay_lsn | 0/38000060
sync_state | async
```

Pour utiliser pg\_rewind, il faut s'assurer que l'instance a été créée avec l'option --data-checksums. Dans le cas contraire, il faut activer wal\_log\_hints = on dans le postgresql.conf.

Pour vérifier les valeurs en cours, exécuter en SQL !

```
SHOW data_checksums ;
```

ou utiliser l'outil shell pg\_controldata :

```
$ /usr/pgsql-14/bin/pg_controldata \
-D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version: 0
```

Par défaut, une instance est initialisée sans sommes de contrôle. Nous devons donc configurer le paramètre wal\_log\_hints à la valeur on. Lançons un *checkpoint* afin de s'assurer que les nouveaux journaux contiendrons les *hints bits* :

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

Si vous avez suivi correctement les travaux pratiques précédents, vous avez activé les *checksums* et obtenez donc le résultat suivant :

```
$ /usr/pgsql-14/bin/pg_controldata \
-D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version: 1
```

## 7.6.2 Promotion d'une instance secondaire



**But :** Réaliser la promotion d'une instance secondaire.

Contrôler que l'archivage est bien configuré sur l'instance **instance2**.

Arrêter l'instance principale pour promouvoir l'instance **instance2**.

Vérifier le répertoire d'archivage. Que contient-il ?

### 7.6.3 Suivi de timeline



**But :** Comprendre la relation entre timeline et réPLICATION.

Nous avons donc :

- l'ancienne instance primaire arrêtée ;
- l'instance **instance2** promue ;
- l'instance **instance3** qui est toujours secondaire de l'instance primaire arrêtée.

Nous allons tenter de raccrocher l'instance **instance3** au nouveau primaire.

- Configurer **instance3** pour qu'il se connecte au **instance2**.
- Observer les traces de l'instance : que se passe-t-il ?

### 7.6.4 pg\_rewind



**But :** « rembobiner » un ancien primaire avec pg\_rewind.

Nous allons maintenant expérimenter le cas où l'instance primaire n'a pas été arrêtée avant la bascule. Il peut donc y avoir des transactions dans la *timeline* 2 qui n'existent pas dans la *timeline* 3 ce qui rend impossible la mise en réPLICATION avec le **instance2**.

Comme décrit plus haut, pour utiliser pg\_rewind, il faut s'assurer que l'instance a été créée avec l'option --data-checksums ou que le paramètre wal\_log\_hints est à on.

- Redémarrer l'ancienne instance primaire (**instance1**).
- Y créer une table t7. Dans quelle *timeline* existe-t-elle ?

- Créer une table t8 sur **instance2**.
  - Arrêter **instance1**.
  - La resynchroniser avec **instance2** à l'aide de pg\_rewind.
  - Ne pas oublier de vérifier les fichiers de configuration.
  - Configurer cette instance resynchronisée en réPLICATION depuis l'instance **instance2**.
  - Ne pas démarrer.
- Démarrer. Observer les traces. Qu'est-il advenu des données insérées avant l'utilisation de pg\_rewind ?
  - Vérifier que **instance2** est bien le serveur principal des deux autres instances.

### 7.6.5 pgBackRest



**But :** Utiliser pgBackRest dans le cadre de la réPLICATION.

- Configurer pgBackRest pour sauvegarder votre nouvelle instance principale (**instance2**).
  - Effectuer une sauvegarde initiale.
  - Insérer des données et vérifier que des archives ont bien été générées.
- Effectuer une promotion « par erreur » du serveur **instance3**, et y insérer des données.
  - Resynchroniser l'instance secondaire **instance3** à l'aide de la sauvegarde de pgBackRest.
  - Vérifier enfin que la réPLICATION entre **instance3** et **instance2** a bien repris.

## 7.7 TRAVAUX PRATIQUES (SOLUTIONS)

### 7.7.1 (Pré-requis) Environnement

Pour ces TP vous devez avoir 3 instances (1 primaire, 2 secondaires en réPLICATION). Au besoin, repasser en réPLICATION ASYNCHRONE en désactivant le paramètre `synchronous_standby_names` de l'instance primaire.

```
$ ps -o pid,cmd fx

PID CMD
3205 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance3/
3207 _ postgres: logger
3208 _ postgres: startup recovering 0000000300000000000000038
3212 _ postgres: checkpointer
3213 _ postgres: background writer
3215 _ postgres: stats collector
3216 _ postgres: walreceiver streaming 0/38000060
3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
3146 _ postgres: logger
3147 _ postgres: startup recovering 0000000300000000000000038
3151 _ postgres: checkpointer
3152 _ postgres: background writer
3154 _ postgres: stats collector
3155 _ postgres: walreceiver streaming 0/38000060
2896 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
2898 _ postgres: logger
2900 _ postgres: checkpointer
2901 _ postgres: background writer
2902 _ postgres: walwriter
2903 _ postgres: autovacuum launcher
2904 _ postgres: archiver last was 0000000300000000000000037.00000028.backup
2905 _ postgres: stats collector
2906 _ postgres: logical replication launcher
3156 _ postgres: walsender repli 127.0.0.1(47494) streaming 0/38000060
3217 _ postgres: walsender repli 127.0.0.1(47502) streaming 0/38000060

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

-[RECORD 1]----+-----
application_name | instance2
state | streaming
sent_lsn | 0/38000060
write_lsn | 0/38000060
flush_lsn | 0/38000060
replay_lsn | 0/38000060
sync_state | async
-[RECORD 2]----+-----
application_name | instance3
state | streaming
sent_lsn | 0/38000060
write_lsn | 0/38000060
flush_lsn | 0/38000060
```

```
replay_lsn | 0/38000060
sync_state | async
```

Pour utiliser pg\_rewind, il faut s'assurer que l'instance a été créée avec l'option --data-checksums. Dans le cas contraire, il faut activer wal\_log\_hints = on dans le postgresql.conf.

Pour vérifier les valeurs en cours, exécuter en SQL !

```
SHOW data_checksums ;
```

ou utiliser l'outil shell pg\_controldata :

```
$ /usr/pgsql-14/bin/pg_controldata \
-D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version: 0
```

Par défaut, une instance est initialisée sans sommes de contrôle. Nous devons donc configurer le paramètre wal\_log\_hints à la valeur on. Lançons un *checkpoint* afin de s'assurer que les nouveaux journaux contiendrons les *hints bits* :

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

Si vous avez suivi correctement les travaux pratiques précédents, vous avez activé les *checksums* et obtenez donc le résultat suivant :

```
$ /usr/pgsql-14/bin/pg_controldata \
-D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version: 1
```

### 7.7.2 Promotion d'une instance secondaire

Contrôler que l'archivage est bien configuré sur l'instance **instance2**.

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

Avec archive\_mode à on, seule l'instance primaire effectue l'archivage des journaux de transaction.

Arrêter l'instance principale pour promouvoir l'instance **instance2**.

```
systemctl stop instance1

$ psql -p 5433 -c "SELECT pg_promote(true, 60);"
pg_promote

t
```

Dans les traces, nous trouvons ceci :

```
LOG: received promote request
LOG: redo done at 0/39000028
cp: cannot stat '/var/lib/pgsql/14/archives/0000000300000000000000039':
 No such file or directory
cp: cannot stat '/var/lib/pgsql/14/archives/00000004.history':
 No such file or directory
LOG: selected new timeline ID: 4
LOG: archive recovery complete
cp: cannot stat '/var/lib/pgsql/14/archives/00000003.history':
 No such file or directory
LOG: database system is ready to accept connections
```

Le serveur a bien reçu la demande de promotion, ensuite (si `restore_command` est spécifié) il cherche à rejouer les derniers journaux de transaction depuis les archives. Puis il vérifie la présence d'un fichier `00000003.history` pour déterminer une nouvelle timeline.

Une fois ces opérations effectuées, il détermine la nouvelle *timeline* (ici, 4) et devient accessible en lecture/écriture :

```
$ psql -p 5433 b1
psql (14.1)
Type "help" for help.

b1=# CREATE TABLE t6 (c1 int);
CREATE TABLE
b1=# INSERT INTO t6 VALUES ('1');
INSERT 0 1
```

### Vérifier le répertoire d'archivage. Que contient-il ?

Vu que le paramètre `archive_mode` était à `on`, seul le serveur primaire effectuait l'archivage. En effectuant la promotion de l'instance **instance2**, celle-ci est devenue primaire et archive donc vers le même répertoire !

Il y a bien un archiver process :

```
3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
3146 _ postgres: logger
3151 _ postgres: checkpointer
3152 _ postgres: background writer
3154 _ postgres: stats collector
3309 _ postgres: walwriter
3310 _ postgres: autovacuum launcher
3311 _ postgres: archiver last was 0000000300000000000000039.partial
3312 _ postgres: logical replication launcher
```

Le répertoire d'archive contient :

```
$ ls -alh /var/lib/pgsql/14/archives/
(..)
-rw-----. 1 postgres postgres 16M Nov 28 16:06 0000000300000000000000039.partial
-rw-----. 1 postgres postgres 84 Nov 28 16:06 00000004.history
(..)
```

Le serveur a archivé le dernier journal avec le suffixe .partial : ceci afin d'éviter d'écraser un fichier journal si le primaire était toujours actif. Avant la version 9.5, il pouvait y avoir un conflit entre l'ancien primaire et le secondaire promu archivant au même emplacement. Le fichier 000000030000000000000002F aurait pu être écrasé. Voir : *Archiving of last segment on timeline after promotion* <http://paquier.xyz/postgresql-2/postgres-9-5-feature-highlight-partial-segment-timeline/>.

### 7.7.3 Suivi de timeline

- Configurer **instance3** pour qu'il se connecte au **instance2**.
- Observer les traces de l'instance : que se passe-t-il ?

Dans les traces de **instance3** :

```
2019-11-28 16:13:17.328 CET [3876] FATAL: could not connect to the primary server:
 could not connect to server: Connection refused
 Is the server running on host "127.0.0.1" and accepting
 TCP/IP connections on port 5432?
```

Le serveur ne sait pas que l'instance **instance2** a été promue, il constate juste que l'instance primaire a été arrêtée.

Modifions le paramètre `primary_conninfo` de l'instance **instance3** pour pointer vers l'instance **instance2** :

```
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli
 password=repli application_name=instance3'
```

Après redémarrage de l'instance, on peut constater que la modification s'est appliquée sur la *timeline* :

```
(...)
LOG: database system is ready to accept read only connections
LOG: started streaming WAL from primary at 0/2F000000 on timeline 4
LOG: redo starts at 0/390000A0
```



À partir de la version 12, `recovery_target_timeline` est désormais à `latest` par défaut. Précédemment, c'était `current`. Grâce à ce changement, l'instance **instance3** a pu suivre la nouvelle *timeline* générée. Avant la version 12, il aurait fallu modifier ce paramètre.

La table t6 existe bien et contient bien les enregistrements :

```
$ psql -p 5434 b1 -c "SELECT * FROM t6;"

c1

1
```

#### 7.7.4 pg\_rewind

- Redémarrer l'ancienne instance primaire (**instance1**).

```
$ sudo systemctl start instance1
```

- Y créer une table t7. Dans quelle *timeline* existe-t-elle ?

Créons la table t7 sur **instance1** :

```
$ psql -c "CREATE TABLE t7 (c1 int)" b1
```

```
$ psql -c "INSERT INTO t7 VALUES ('1')" b1
```

- Créer une table t8 sur **instance2**.

Créons la table t8 sur **instance2** :

```
$ psql -p 5433 -c "CREATE TABLE t8 (c1 int)" b1
```

```
$ psql -p 5433 -c "INSERT INTO t8 VALUES ('1')" b1
```

La table t7 existe dans la timeline 3 mais pas dans la *timeline* 4. Nous allons donc utiliser l'outil pg\_rewind. Il ne lit et ne copie que les données modifiées. L'outil identifie les blocs correspondants aux transactions « perdues ». Dans notre exemple, la table t7 n'est présente que sur la nouvelle instance. Ensuite il copie les blocs correspondants sur l'instance cible.

- Arrêter **instance1**.
- La resynchroniser avec **instance2** à l'aide de pg\_rewind.
- Ne pas oublier de vérifier les fichiers de configuration.
- Configurer cette instance resynchronisée en réplication depuis l'instance **instance2**.
- Ne pas démarrer.

```
$ sudo systemctl stop instance1
```

Maintenant passons à pg\_rewind. L'option -n permet d'effectuer un test sans modification :

```
$ /usr/pgsql-14/bin/pg_rewind -D /var/lib/pgsql/14/instance1/ -n -P \
--source-server="port=5433 user=postgres"
```

```
pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/390000A0 on timeline 3
pg_rewind: rewinding from last common checkpoint at 0/39000028 on timeline 3
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 101 MB (total source directory size is 370 MB)
103727/103727 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

On relance la commande sans l'option -n :

```
$ /usr/pgsql-14/bin/pg_rewind -D /var/lib/pgsql/14/instance1/ -P \
 --source-server="port=5433 user=postgres"

pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/390000A0 on timeline 3
pg_rewind: rewinding from last common checkpoint at 0/39000028 on timeline 3
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 101 MB (total source directory size is 370 MB)
103735/103735 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

On constate qu'il a juste été nécessaire de copier 101 Mo au lieu des 370 Mo de l'instance complète.

pg\_rewind se connecte sur le serveur source et identifie le point de divergence. Ensuite, il liste les fichiers à copier ou supprimer. Puis il copie les blocs modifiés. Enfin il met à jour le fichier backup\_label et le fichier pg\_control.

Les fichiers postgresql.conf et postgresql.auto.conf provenant du **instance2**, il faut modifier le port ainsi que le primary\_conninfo :

```
postgresql.conf
port = 5432

postgresql.auto.conf
primary_conninfo = 'user=repli passfile=''${PGDATA}/.pgpass''
host=127.0.0.1 port=5433 application_name=instance1'
```

Créer le fichier standby.signal :

```
$ touch /var/lib/pgsql/14/instance1/standby.signal
```

Et démarrer l'instance :

```
$ sudo systemctl start instance1
```

- Démarrer. Observer les traces. Qu'est-il advenu des données insérées avant l'utilisation de pg\_rewind ?

Dans les traces :

```
LOG: database system was interrupted while in recovery at log time ...
HINT: If this has occurred more than once some data might be corrupted and
you might need to choose an earlier recovery target.
cp: cannot stat '/var/lib/pgsql/14/archives/00000005.history': No such
file or directory
LOG: entering standby mode
LOG: restored log file "00000004.history" from archive
cp: cannot stat '/var/lib/pgsql/14/archives/000000040000000000000039': No
such file or directory
cp: cannot stat '/var/lib/pgsql/14/archives/00000003.history': No such
file or directory
```

```
LOG: redo starts at 0/390000A0
LOG: started streaming WAL from primary at 0/39000000 on timeline 4
LOG: consistent recovery state reached at 0/3904A8E0
LOG: database system is ready to accept read only connections
```

La table t7 a bien disparu de l'instance resynchronisée :

```
$ psql -p 5432 -c "SELECT * FROM t7;" b1
```

```
ERROR: relation "t7" does not exist
```

- Vérifier que **instance2** est bien le serveur principal des deux autres instances.

```
$ psql -x -p 5433 -c "SELECT * FROM pg_stat_replication;"
```

| -[ RECORD 1 ]----- |            |
|--------------------|------------|
| pid                | 3610       |
| usesysid           | 16384      |
| username           | repli      |
| application_name   | instance3  |
| client_addr        | 127.0.0.1  |
| client_hostname    |            |
| client_port        | 49942      |
| backend_start      | ...        |
| backend_xmin       |            |
| state              | streaming  |
| sent_lsn           | 0/3904B628 |
| write_lsn          | 0/3904B628 |
| flush_lsn          | 0/3904B628 |
| replay_lsn         | 0/3904B628 |
| write_lag          |            |
| flush_lag          |            |
| replay_lag         |            |
| sync_priority      | 0          |
| sync_state         | async      |
| reply_time         | ...        |
| -[ RECORD 2 ]----- |            |
| pid                | 3797       |
| usesysid           | 16384      |
| username           | repli      |
| application_name   | instance1  |
| client_addr        | 127.0.0.1  |
| client_hostname    |            |
| client_port        | 49966      |
| backend_start      | ...        |
| backend_xmin       |            |
| state              | streaming  |
| sent_lsn           | 0/3904B628 |
| write_lsn          | 0/3904B628 |
| flush_lsn          | 0/3904B628 |
| replay_lsn         | 0/3904B628 |
| write_lag          |            |
| flush_lag          |            |
| replay_lag         |            |
| sync_priority      | 0          |
| sync_state         | async      |
| reply_time         | ...        |

### 7.7.5 pgBackRest

- Configurer pgBackRest pour sauvegarder votre nouvelle instance principale (**instance2**).

Commencer par modifier :

```
archive_command = 'pgbackrest --stanza=instance_dev archive-push %p'
```

Ensuite, recharger la configuration :

```
$ sudo systemctl reload instance2
```

Modifier ensuite la configuration de pgBackRest dans /etc/pgbackrest.conf :

```
[global]
start-fast=y
log-level-console=info
process-max=1
repo1-path=/var/lib/pgsql/14/backups
repo1-retention-full=1

[instance_dev]
pg1-path=/var/lib/pgsql/14/instance2
pg1-port=5433
```

Vérifier que le répertoire où les archives et les sauvegardes seront stockées est bien vide (/var/lib/pgsql/14/backups/).

Initialiser le dépôt de pgBackRest et vérifier que l'archivage est fonctionnel :

```
$ pgbackrest --stanza=instance_dev stanza-create
P00 INFO: stanza-create command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev
P00 INFO: stanza-create command end: completed successfully (463ms)

$ pgbackrest --stanza=instance_dev check
P00 INFO: check command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev
P00 INFO: WAL segment 000000040000000000000039 successfully archived to
'/var/lib/pgsql/14/backups/archive/instance_dev/14-1/0000000400000000/
000000040000000000000039-9b054265590ac75cd70ba3ed5b1f47c113a69b84.gz'
P00 INFO: check command end: completed successfully
```

- Effectuer une sauvegarde initiale.
- Insérer des données et vérifier que des archives ont bien été générées.

Pour une première sauvegarde :

```
$ pgbackrest --stanza=instance_dev --type=full backup |grep P00
P00 INFO: backup command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433 --process-max=1
--repo1-path=/var/lib/pgsql/14/backups --repo1-retention-full=1
```

```
--stanza=instance_dev --start-fast --type=full
P00 INFO: execute non-exclusive pg_start_backup() with label "...": backup
begins after the requested immediate checkpoint completes
P00 INFO: backup start archive = 000000040000000000000003B, lsn = 0/3B000060
P00 INFO: full backup size = 274.2MB
P00 INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments
to archive
P00 INFO: backup stop archive = 000000040000000000000003B, lsn = 0/3B000138
P00 INFO: new backup label = 20200110-081833F
P00 INFO: backup command end: completed successfully
P00 INFO: expire command begin 2.20: --log-level-console=info
--repo1-path=/var/lib/pgsql/14/backups --repo1-retention-full=1
--stanza=instance_dev
P00 INFO: expire command end: completed successfully
```

Insérer des données :

```
$ psql -p 5433 -c "CREATE TABLE t9(id int);"
CREATE TABLE
$ psql -p 5433 -c "INSERT INTO t9 SELECT generate_series(0,1000000);"
INSERT 0 1000001
```

Forcer la génération d'un nouveau WAL et vérifier que des archives ont bien été générées :

```
$ psql -p 5433 -c "select pg_switch_wal();"
pg_switch_wal

0/3FD474E8
(1 row)

$ psql -p 5433 -c "select last_archived_wal from pg_stat_archiver;"
last_archived_wal

000000040000000000000003F
(1 row)
```

- Effectuer une promotion « par erreur » du serveur **instance3**, et y insérer des données.

```
$ psql -p 5434 -c "SELECT pg_promote(true, 60);"
pg_promote

t

$ psql -p 5434 -c "CREATE TABLE t10(id int);"
CREATE TABLE
```

- Resynchroniser l'instance secondaire **instance3** à l'aide de la sauvegarde de pgBackRest.

De nouvelles données ayant été écrites, nous devons désormais reconstruire l'instance **instance3** pour pouvoir la remettre en réPLICATION avec **instance2**. Pour ce faire, nous pourrions utiliser pg\_rewind comme précédemment. Nous allons ici utiliser les sauvegardes pgBackRest pour reconstruire l'instance.

La sauvegarde de l'instance primaire peut être restaurée sur la seconde instance avec les options `--delta` et `--type=standby` pour écraser les fichiers erronés et ajouter le fichier `standby.signal` dans le répertoire de données. pgBackRest se charge de configurer les options de réPLICATION dans le fichier `postgresql.auto.conf`.

```
$ sudo systemctl stop instance3
$ pgbackrest restore --stanza=instance_dev --delta --type=standby
--recovery-option="primary_conninfo=user=repli passfile='/var/lib/pgsql/.pgpass'"
host=127.0.0.1 port=5433 application_name=instance3"
--pg1-path=/var/lib/pgsql/14/instance3 | grep P00

P00 INFO: restore command begin 2.20: --delta --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance3 --process-max=1
--recovery-option="primary_conninfo=user=repli passfile='/var/lib/pgsql/.pgpass'"
host=127.0.0.1 port=5433 application_name=instance3"
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev --type=standby
P00 INFO: restore backup set ...
P00 INFO: remap data directory to '/var/lib/pgsql/14/instance3'
P00 INFO: remove invalid files/links/paths from '/var/lib/pgsql/14/instance3'
P00 INFO: write updated /var/lib/pgsql/14/instance3/postgresql.auto.conf
P00 INFO: restore global/pg_control (performed last to ensure aborted restores
 cannot be started)
P00 INFO: restore command end: completed successfully
```

- Vérifier enfin que la réPLICATION entre **instance3** et **instance2** a bien repris.

Modifier le port de l'instance **instance3** avant de la démarrer et vérifier que la connexion de réPLICATION est bien fonctionnelle :

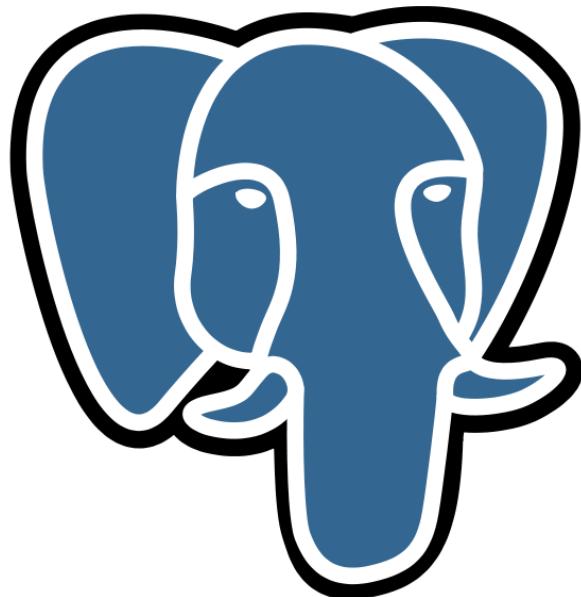
```
port=5434

$ sudo systemctl start instance3
$ psql -x -p 5433 -c "SELECT application_name, state, sync_state FROM
 pg_stat_replication WHERE application_name='instance3';"

-[RECORD 1]-----+
application_name | instance3
state | streaming
sync_state | async
```



## 8/ RéPLICATION logique



## 8.1 OBJECTIFS



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

La réPLICATION logique a été ajoutée dans PostgreSQL depuis la version 10, et est améliorée régulièrement à chaque version.

Nous verrons ici les principes derrière ce type de réPLICATION, sa mise en place, son administration et sa supervision.

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

### 8.1.1 Au menu



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

## 8.2 PRINCIPES DE LA RÉPLICATION LOGIQUE NATIVE



- RéPLICATION logique
  - résout certaines des limitations de la réPLICATION physique
  - native depuis la version 10
  - avant v10 : solutions externes

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

La réPLICATION logique apporte des réponses à ces limitations. Seules des solutions tierces apportaient ce type de réPLICATION à PostgreSQL.

### 8.2.1 RéPLICATION physique vs. logique

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

#### Principe & limites de la réPLICATION physique :

La réPLICATION physique est une réPLICATION au niveau bloc. Le serveur primaire envoie au secondaire les octets à ajouter/remplacer dans des fichiers. Le serveur secondaire n'a aucune information sur les objets logiques (tables, index, vues matérialisées, bases de données). Il n'y a donc pas de granularité possible, c'est forcément l'instance complète qui est répliquée. Cette réPLICATION est par défaut en asynchrone mais il est possible de la configurer en synchrone suivant différents modes.

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

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

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

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

#### RéPLICATION logique :

La réPLICATION logique propose une solution à tous ces problèmes.

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

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

Elle est asymétrique dans le sens où il existe une seule origine des écritures pour une table. Si elle ne permet pas un vrai fonctionnement multi-maîtres, il est possible de réaliser des réPLICATIONS croisées où un ensemble de tables est réPLIQUÉ du serveur 1 vers le serveur 2 et un autre ensemble de tables est réPLIQUÉ du serveur 2 vers le serveur 1.

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

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

#### 8.2.2 Quelques termes essentiels



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

Dans le cadre de la réPLICATION logique, on ne réPLIQUE pas une instance vers une autre. On publie les modifications effectuées sur le contenu d'une table à partir d'un serveur. Ce serveur est le serveur origine, ou publieur (*publisher*). De lui sont enregistrées les modifications que d'autres serveurs

pourront récupérer en s'abonnant (*subscription*) . Ces serveurs abonnés indiquent leur intérêt sur ces modifications en s'abonnant à la publication.

De ceci, il découle que :

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

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

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

### 8.2.3 RéPLICATION en streaming



- Paramètre `wal_level`
- Processus `wal_sender`
  - mais pas de `wal_receiver`
  - un `logical replication worker` à la place
- Asynchrone / synchrone
- Slots de réPLICATION

La réPLICATION logique utilise le même canal d'informations que la réPLICATION physique : les enregistrements des journaux de transactions. Pour que les journaux disposent de suffisamment d'informations, le paramètre `wal_level` doit être configuré avec la valeur `logical`.

Une fois cette configuration effectuée et PostgreSQL redémarré sur le serveur origine, le serveur destination pourra se connecter au serveur origine dans le cadre de la réPLICATION. Lorsque cette connexion est faite, un processus `wal_sender` apparaîtra sur le serveur origine. Ce processus sera en communication avec un processus `logical replication worker` sur le serveur destination.

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

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

## 8.2.4 Granularité



- Par table
  - toutes les tables d'une base
  - toutes les tables d'un schéma (v15+)
  - quelques tables spécifiques
- Granularité d'une table
  - table complète
  - uniquement certaines lignes (v15+)
  - uniquement certaines colonnes (v15+)
- Par opération
  - INSERT, UPDATE, DELETE, TRUNCATE (v11)

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

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

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

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

La granularité peut aussi se voir au niveau des opérations de modification réalisées. On peut très bien ne publier que les opérations d'insertion, de modification ou de suppression. Par défaut, tout est publié. La réPLICATION de la commande TRUNCATE n'a été ajoutée qu'à partir de la version 11.

## 8.2.5 Possibilités sur les tables répliquées



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

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

On peut rajouter ou supprimer des index sur la table répliquée, pourvu que les lignes restent identifiables. Au besoin on peut préciser l'index, qui doit être unique sur colonne NOT NULL servant de clé :

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

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

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

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

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



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



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

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



Pour que la réPLICATION logique fonctionne sans souci, il faut viser au plus simple, avec un modèle de données sur la destination aussi proche que possible de la source, soigneusement maintenu à jour à l'identique. Éviter de modifier les données répliquées. Au plus, se contenter d'ajouter sur la destination des index non uniques ou des colonnes calculées.

Prévoir dès le début le cas où cette réPLICATION devra être arrêtée et reprise de zéro.

## 8.2.6 Limitations de la réPLICATION logique



- Pas de réPLICATION des requêtes DDL
  - à refaire manuellement
  - être rigoureux et surveiller les traces !
- Pas de TRUNCATE en v10
- Pas de réPLICATION des valeurs des séquences
- Pas de réPLICATION des LO (table système)
- Contraintes d'unicité obligatoires pour les UPDATE/DELETE
- Coût en CPU et I/O non négligeable
- RéPLICATION déclenchée uniquement lors du COMMIT (< v14)
- Attention en cas de bascule/restauration !

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

La première, et plus importante, est qu'elle ne réplique que les changements de données des tables (commandes DML), et pas de la définition des objets de la base de données (commandes DDL). L'ajout (ou la suppression) d'une colonne ne sera pas répliqué, causant de ce fait un problème de réPLICATION quand l'utilisateur y ajoutera des données. La mise à jour sera bloquée jusqu'à ce que les tables abonnées soient aussi mises à jour.

D'autres opérations moins évidentes peuvent aussi poser problème, comme une contrainte ou un index supprimé sur l'origine mais pas la cible ; ou un index fonctionnel dont la fonction n'est corrigée que sur la source. Il faut être rigoureux et surveiller les erreurs dans les traces.

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

Le TRUNCATE n'est pas répliqué en version 10, mais il l'est bien à partir de la version 11.

Il n'y a pas non plus de réPLICATION des valeurs des séquences. Les valeurs des séquences sur les serveurs destinations seront donc obsolètes.

Les Large Objects étant stockés dans une table système, ils ne sont pas pris en compte par la réPLICATION logique.

Les opérations UPDATE et DELETE nécessitent la présence d'une contrainte unique pour s'assurer de modifier ou supprimer les bonnes lignes. Une clé primaire est conseillée de toute manière pour bien identifier les lignes.

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

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

## 8.3 MISE EN PLACE



- Cas simple
  - 2 serveurs
  - une seule origine
  - un seul destinataire
  - une seule publication
- Plusieurs étapes
  - configuration du serveur origine
  - configuration du serveur destination
  - création d'une publication
  - ajout d'une souscription

Dans cette partie, nous allons aborder un cas simple avec uniquement deux serveurs. Le premier sera l'origine, le second sera le destinataire des informations de réPLICATION. Toujours pour simplifier l'explication, il n'y aura pour l'instant qu'une seule publication.

La mise en place de la réPLICATION logique consiste en plusieurs étapes :

- la configuration du serveur origine ;
- la configuration du serveur destination ;
- la création d'une publication ;
- la souscription à une publication.

Nous allons voir maintenant ces différents points.

### 8.3.1 Configurer le serveur origine



- Création et configuration de l'utilisateur de réPLICATION
  - et lui donner les droits de lecture des tables à répliquer
- Configuration du fichier `postgresql.conf`
  - `wal_level = logical`
  - `logical_decoding_work_mem = 64MB (v13+)`
- Configuration du fichier `pg_hba.conf`
  - autoriser une connexion de réPLICATION du serveur destination

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

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

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

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

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

Les journaux de transactions doivent disposer de suffisamment d'informations pour que le `wal_sender` puisse envoyer les bonnes informations au `logical replication worker`. Pour cela, il faut configurer le paramètre `wal_level` à `logical` dans le fichier `postgresql.conf`.

La réPLICATION logique, contrairement à la réPLICATION physique, n'est déclenchée que lors d'un COMMIT (voir cet article<sup>1</sup>). Par défaut, il n'y a pas d'envoi des données tant que la transaction est en cours, ce qui peut ajouter beaucoup de délai de réPLICATION pour les transactions longues.

Le paramètre `logical_decoding_work_mem` contrôle la quantité de mémoire allouée à un processus `walsender` pour conserver les modifications en mémoire avant de les stocker sur le disque. Avant PostgreSQL version 13 et l'apparition de ce paramètre, les modifications d'une transaction étaient stockées en mémoire jusqu'à ce que la transaction soit validée par un COMMIT. En conséquence, si cette transaction possédait de nombreuses sous-transactions, chaque `walsender` pouvait allouer énormément de mémoire, menant parfois à un dépassement de mémoire. Avec ce paramètre, il est possible de diminuer sa valeur pour réduire l'utilisation de la mémoire des

<sup>1</sup><http://amitkapila16.blogspot.com/2021/07/logical-replication-of-in-progress.html>

walsender ou définir une valeur plus élevée pour réduire les écritures sur le disque. La valeur par défaut est de 64 Mo.

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

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

en remplaçant XXX.XXX.XXX.XXX/XX par l'adresse CIDR du serveur destination. La méthode d'authentification peut aussi être changée suivant la politique interne. Suivant la méthode d'authentification, il sera nécessaire ou pas de configurer un mot de passe pour cet utilisateur.

Si le paramètre wal\_level a été modifié, il est nécessaire de redémarrer le serveur PostgreSQL. Si seul le fichier pg\_hba.conf a été modifié, seul un rechargement de la configuration est demandé.

### 8.3.2 Configuration du serveur destination



- Création de l'utilisateur de réPLICATION
- Création, si nécessaire, des tables répliquées

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

- Souscription

```
CREATE SUBSCRIPTION ... PUBLICATION pub WITH (streaming = on);
```

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

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

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

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

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

Avant la version 14, le walsender attendait toujours le COMMIT avant d'envoyer une transaction aux abonnés, et de grosses transactions pouvaient entraîner l'apparition d'énormes fichiers dans le répertoire pg\_replslot du serveur d'origine. Depuis la version 14, cela reste le comportement par

défaut, mais il est possible de forcer l'envoi des données au serveur destinataire sans attendre le COMMIT. Le serveur distant stockera les données dans un fichier et ne les rejouera qu'à partir du moment où le COMMIT est reçu.

Cette fonctionnalité doit s'activer explicitement au niveau de la souscription :

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

### 8.3.3 Créer une publication



```
CREATE PUBLICATION nom
 [FOR TABLE [ONLY] nom_table [*] [(nom_colonne [, ...])] [
 ↵ WHERE (expression)] [, ...] |
 FOR TABLES IN SCHEMA nom_schema [*] [, ...] |
 FOR ALL TABLES]
 [WITH (parametre_publication [= valeur] [, ...])]
 - parametre_publication = publish
 - insert, update, delete, et/ou truncate
 - tous par défaut
 - parametre_publication = publish_via_partition_root
```

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

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

À partir de la version 15, la clause FOR TABLES IN SCHEMA permet de répliquer toutes les tables du schéma indiqué sans avoir à nommer les tables spécifiquement. De plus, toute nouvelle table de ce schéma sera répliquée automatiquement dès sa création.

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

Depuis la version 15, il est possible de ne répliquer que certaines colonnes d'une table. Dans ce cas, il faut indiquer les colonnes en question. En voici un exemple :

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

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

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

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

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

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

### 8.3.4 Souscrire à une publication



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

- infos_connexion : chaîne de connexion habituelle
```

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

Le serveur s'indique avec la chaîne infos\_connexion, dont la syntaxe est la syntaxe habituelle des chaînes de connexion. Pour rappel, on utilise les mots clés host, port, user, password, dbname, etc.

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

Les paramètres de souscription sont détaillés dans la slide suivante.

### 8.3.5 Options de la souscription (1/2)



- `copy_data`
  - copie initiale des données (activé par défaut)
- `create_slot`
  - création du slot de réPLICATION (activé par défaut)
- `enabled`
  - activation immédiate de la souscription (activé par défaut)
- `slot_name`
  - nom du slot (par défaut : nom de la souscription)
- `synchronous_commit`
  - surcharge `synchronous_commit`
- `binary` (désactivé par défaut, v14+)
  - envoie les données sous un format binaire (désactivé par défaut)

Les options de souscription sont assez nombreuses et permettent de créer une souscription pour des cas particuliers. Par exemple, si le serveur destination possède déjà les données du serveur origine, il faut placer le paramètre `copy_data` à la valeur `off`.

### 8.3.6 Options de la souscription (2/2)



- `connect`
  - connexion immédiate (activé par défaut)
- `streaming` (désactivé par défaut, v14+)
  - pour envoyer les modifications des transactions en flux
- `two_phase` (désactivé par défaut, v15+)
  - activation de la validation en deux phases
- `disable_on_error` (désactivé par défaut, v15+)
  - désactivation de la souscription en cas d'erreurs détectées

## 8.4 MISE EN PLACE : EXEMPLE



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

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

### 8.4.1 Serveurs et schéma



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

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

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

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

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

INSERT INTO t1 SELECT i, 't1, ligne'||i FROM generate_series(1, 100) i;
INSERT INTO t2 SELECT i, 't2, ligne'||i FROM generate_series(1, 1000) i;
```

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

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

#### 8.4.2 RéPLICATION complète



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

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

#### 8.4.3 Configuration du serveur origine (1/2)



- Crédit et configuration de l'utilisateur de réPLICATION

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

- Fichier postgresql.conf

```
wal_level = logical
```

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

Les fichiers postgresql.conf et pg\_hba.conf sont modifiés pour y ajouter la configuration nécessaire.

#### 8.4.4 Configuration du serveur origine (2/2)



- Fichier pg\_hba.conf

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

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

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

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

#### 8.4.5 Configuration des 4 serveurs destinations



- Création de l'utilisateur de réPLICATION

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

- Création des tables répliquées (sans contenu)

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

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

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

#### 8.4.6 Créer une publication complète



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

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

On utilise la clause ALL\_TABLES pour une réPLICATION complète d'une base.

#### 8.4.7 Souscrire à la publication



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

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

- Un slot de réPLICATION est créé
- Les données initiales sont immédiatement transférées

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

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

Les données initiales de la table **t1** sont envoyées du serveur **s1** vers le serveur **s2**.

#### 8.4.8 Tests de la réPLICATION complète



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

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

Sur le serveur **s1** :

```
b1=# INSERT INTO t1 VALUES (101, 't1, ligne 101');
b1=# UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;
b1=# DELETE FROM t1 WHERE id_t1=11;
b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
```

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

(2 rows)

Sur le serveur **s2** :

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

| count |
|-------|
| 100   |

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

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

(2 rows)

### 8.4.9 RéPLICATION partielle



- Identique à la réPLICATION complète, à une exception...
- Créer la publication partielle

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

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

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

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

#### Mise en place :

Cela donne donc dans notre exemple :

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

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

#### Vérification :

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

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

 100

b1=# SELECT count(*) FROM t2;
count

 1000

b1=# SELECT count(*) FROM t3;
count

 0
```

À noter que nous avons déjà les données précédemment modifiées :

```
b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
+-----+
| id_t1 | label_t1 |
+-----+
| 101 | t1, ligne 101|
| 10 | T1, LIGNE 10 |
+-----+
```

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

```
b1=# INSERT INTO t1 VALUES (102, 't1, ligne 102');
b1=# INSERT INTO t2 VALUES (1001, 't2, ligne 1002');
b1=# INSERT INTO t3 VALUES (-1, 't3, cle 1, ligne -1', 1);
```

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

```
b1=# SELECT * FROM t1 WHERE id_t1=102;
+-----+
| id_t1 | label_t1 |
+-----+
| 102 | t1, ligne 102|
+-----+

b1=# SELECT * FROM t2 WHERE id_t2=1001;
+-----+
| id_t2 | label_t2 |
+-----+
| 1001 | t2, ligne 1002|
+-----+

b1=# SELECT * FROM t3 WHERE id_t3 < 0;
+-----+
| id_t3 | label_t3 | clepartition_t3 |
+-----+
(0 rows)
```

#### 8.4.10 RéPLICATION croisée



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

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

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

Pour rendre cela encore plus intéressant, nous allons utiliser la table **t3** et ses partitions. Le but est de pouvoir écrire dans la partition **t3\_1** sur **s1** et dans la partition **t3\_2** sur **s2**, simulant ainsi une table où il sera possible d'écrire sur les deux serveurs à condition de respecter la clé de partitionnement.

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

- nous allons commencer par mettre en réPLICATION **t3\_1** ;
- et nous finirons en mettant en réPLICATION **t3\_2**.

#### 8.4.11 RéPLICATION de **t3\_1** de **s1** vers **s4**



- Créer la publication partielle sur **s1**

```
CREATE PUBLICATION publi_t3_1
FOR TABLE t3_1;

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

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

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

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

- Fichier **postgresql.conf**:

```
wal_level = logical
```

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

- Fichier **pg\_hba.conf**:

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

(Ne pas oublier de recharger la configuration.)

#### 8.4.12 RéPLICATION de t3\_2 de s4 vers s1



- Créer la publication partielle sur **s4**

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

- Y souscrire sur **s1**

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

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

#### 8.4.13 Tests de la réPLICATION croisée



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

Sur **s1** :

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
 id_t3 | label_t3 | clepartition_t3
-----+-----+-----
(0 rows)

b1=# INSERT INTO t3 VALUES (1001, 't3, ligne 1001', 1);
INSERT 0 1

b1=# SELECT * FROM t3 WHERE id_t3>999;
 id_t3 | label_t3 | clepartition_t3
-----+-----+-----
 1001 | t3, ligne 1001 | 1
```

Sur **s4**:

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
+-----+-----+-----+
| id_t3 | label_t3 | clepartition_t3 |
+-----+-----+-----+
| 1001 | t3, ligne 1001 | 1
+-----+-----+-----+
b1=# INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 2);
INSERT 0 1
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
+-----+-----+-----+
| id_t3 | label_t3 | clepartition_t3 |
+-----+-----+-----+
| 1001 | t3, ligne 1001 | 1
| 1002 | t3, ligne 1002 | 2
+-----+-----+-----+
```

Sur **s1**:

```
b1=# SELECT * FROM t3 WHERE id_t3>999;
+-----+-----+-----+
| id_t3 | label_t3 | clepartition_t3 |
+-----+-----+-----+
| 1001 | t3, ligne 1001 | 1
| 1002 | t3, ligne 1002 | 2
(2 rows)
```

## 8.5 ADMINISTRATION



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

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

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

### 8.5.1 Processus



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

Tout comme il existe un processus `wal_sender` communiquant avec un processus `wal_receiver` dans le cadre de la réPLICATION physique, il y a aussi deux processus discutant ensemble dans le cadre de la réPLICATION logique.

Pour commencer, un serveur en version 10 dispose d'un processus supplémentaire, le `logical replication launcher`. Ce processus a pour but de demander le lancement d'un `logical replication worker` lors de la création d'une souscription. Ce `worker` se connecte au serveur origine et applique toutes les modifications dont `s1` lui fait part (on a aussi le terme de *apply worker*, notamment dans certains messages des traces). Si la connexion se passe bien, un processus `wal_sender` est ajouté sur le serveur origine pour communiquer avec le `worker` sur le serveur destination.

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

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

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

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

### 8.5.2 Synthèse des paramètres (serveur origine)

| Streaming                           | Logique                                 |
|-------------------------------------|-----------------------------------------|
| <code>wal_level = replica</code>    | <code>wal_level = logical</code>        |
|                                     | <code>logical_decoding_work_mem*</code> |
| <code>wal_keep_size*</code>         |                                         |
| <code>max_wal_keep_size*</code>     |                                         |
| <code>wal_sender_timeout*</code>    |                                         |
| <code>max_wal_senders</code>        | <code>max_wal_senders</code>            |
| <code>max_connections **</code>     | <code>max_connections **</code>         |
| <code>max_replication_slots*</code> | <code>max_replication_slots</code>      |

(\*) paramètres utiles mais pas indispensables.

(\*\*) avant la version 12 `max_wal_senders` compte dans `max_connections`.

### 8.5.3 Synthèse des paramètres indispensables

| Streaming            | Logique                         |
|----------------------|---------------------------------|
| primary_conninfo     |                                 |
| wal_level=replica    |                                 |
| standby_mode         |                                 |
| max_worker_processes | max_worker_processes            |
|                      | max_logical_replication_workers |

### 8.5.4 Synthèse des paramètres utiles

| Streaming             | Logique |
|-----------------------|---------|
| restore_command       |         |
| primary_slot_name     |         |
| hot_standby           |         |
| promote_trigger_file  |         |
| max_replication_slots |         |
| hot_standby_feedback  |         |

Ce sont les paramètres utiles mais pas indispensables.

### 8.5.5 Fichiers (serveur origine)



- 2 répertoires importants
- pg\_replslot
  - slots de réPLICATION
  - 1 répertoire par slot (+ slots temporaires)
  - 1 fichier state dans le répertoire
  - fichiers .snap (volumétrie !)
- pg\_logical
  - métadonnées
  - snapshots

La réPLICATION logIQUE maintient des données dans deux réPERTOIRES : pg\_replslot et pg\_logical.

pg\_replslot contient un réPERTOIRE par slot de réPLICATION physique ou logIQUE. On y trouvera aussi des slots temporaires lors de l'initialisation de la réPLICATION logIQUE.

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

pg\_logical contient des métadonnées et une volumétrie beaucoup plus faible.

À cela s'ajoutent les journaux de transaction conservés en fonction de l'avancement des slots de réPLICATION.

### 8.5.6 Empêcher les écritures sur un serveur destination



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

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

```
b1=# CREATE ROLE u1 LOGIN;
CREATE ROLE
b1=# GRANT ALL ON ALL TABLES IN SCHEMA public TO u1;
GRANT
```

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

```
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> INSERT INTO t1 VALUES (103, 't1 sur s2, ligne 103');
INSERT 0 1
```

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

```
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# REVOKE INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM u1;
REVOKE
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> INSERT INTO t1 VALUES (104);
ERROR: permission denied for relation t1
```

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

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

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
+-----+-----+
| id_t1 | label_t1 |
+-----+-----+
| 103 | t1 sur s2, ligne 103|
```

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

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
+-----+-----+
| id_t1 | label_t1 |
+-----+-----+
(0 rows)
```

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

```
b1=> INSERT INTO t1 VALUES (103, 't1 sur s1, ligne 103');
INSERT 0 1
```

Et maintenant, on se trouve avec deux serveurs désynchronisés :

- sur **s1** :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
+-----+-----+
| id_t1 | label_t1 |
+-----+-----+
| 103 | t1 sur s1, ligne 103|
```

- sur **s2** :

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
+-----+-----+
| id_t1 | label_t1 |
+-----+-----+
| 103 | t1 sur s2, ligne 103|
```

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

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

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

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

```
b1=# DELETE FROM t1 WHERE id_t1=103;
DELETE 1

b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 | label_t1
-----+-----
(0 rows)
```

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

```
b1=# SELECT * FROM t1 WHERE id_t1=103;
 id_t1 | label_t1
-----+-----
 103 | t1 sur s1, ligne 103
(1 row)
```

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

### 8.5.7 Que faire pour les DDL ?



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

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

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

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

### 8.5.8 Que faire pour les nouvelles tables ?



- Créer la table sur la cible
- Si publication sans FOR ALL TABLES :
  - ajouter la nouvelle table dans les souscriptions concernées

```
ALTER PUBLICATION ... ADD TABLE ... ;
ALTER SUBSCRIPTION ... REFRESH PUBLICATION ;
```

La création d'une table est une opération DDL. Elle est donc ignorée dans le contexte de la réPLICATION logique.

Si l'on veut la répliquer, il faudra créer la table manuellement dans la base destinataire. Puis, deux cas se présentent : la publication a été déclarée pour certaines tables uniquement, ou avec FOR ALL TABLES.

Si elle ne concerne qu'un sous-ensemble de tables, il faut ajouter la nouvelle table à la publication :

```
ALTER PUBLICATION ... ADD TABLE ... ;
```

Si la publication a été créée avec la clause FOR ALL TABLES, la nouvelle table est immédiatement prise en compte dans la publication.

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

```
ALTER SUBSCRIPTION ... REFRESH PUBLICATION ;
```

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

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

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

Voici un exemple sans FOR ALL TABLES :

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

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
CREATE TABLE

b1=# GRANT SELECT ON TABLE t4 TO logrepli;
GRANT

b1=# INSERT INTO t4 VALUES (1);
INSERT 0 1
```

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

```
b1=# SELECT * FROM t4;
ERROR: relation "t4" does not exist
LINE 1: SELECT * FROM t4;
 ^
```

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

```
b1=# CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
CREATE TABLE

b1=# SELECT * FROM t4;
 id_t4

(0 rows)
```

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

```
b1=# ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;
ALTER SUBSCRIPTION
b1=# SELECT * FROM t4;
 id_t4

 1
(1 row)
```

### 8.5.9 Gérer les opérations de maintenance



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

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

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

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

### 8.5.10 Gérer les sauvegardes & restaurations logiques



- pg\_dumpall et pg\_dump
  - sauvegardent publications et souscriptions
  - options --no-publications et --no-subscriptions
- Restauration d'une publication :
  - nouveau slot de réPLICATION !
  - réconciliation de données à prévoir
- Restauration d'un abonnement :
  - ENABLE et REFRESH PUBLICATION
  - reprendre à zéro la copie... ou copier manuellement ?

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

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

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

#### Restauration d'une publication

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

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

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

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



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

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

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

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

### Restauration d'une souscription

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

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

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



Une réPLICATION restaurée est donc par défaut inactive.

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

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

Ces opérations sont obligatoirement manuelles.



La restauration logique d'un abonnement revient à en créer un nouveau, et ne permet pas de savoir où la copie s'était arrêtée auparavant dans le flux des transactions : la copie des données sera intégralement relancée.

Sans autre opération, et si le contenu des tables répliquées a été restauré, le contenu déjà présent bloquera la réPLICATION (s'il y a une clé primaire) ou de se retrouver en double (sans clé primaire). Il peut être plus simple de ne pas restaurer les données sur la destination, ou de tronquer les tables avant le ENABLE, pour reprendre la copie à zéro. Une alternative est de ne pas effectuer la copie initiale :

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

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

### 8.5.11 Gérer les bascules & les restaurations physiques



Comme pour la réPLICATION physique :

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

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

#### Restauration de l'instance d'origine :

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

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

#### **Bascule de l'instance d'origine :**

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

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

#### **Restauration de l'instance de destination :**

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

En redémarrant, les abonnements vont tenter de se raccrocher au slot de réPLICATION de l'origine, ce qui fonctionnera, mais ils ne recevront que des données jamais répliquées. Là encore des « trous » dans les données répliquées peuvent apparaître si l'instance destination n'a pas été restaurée dans un état suffisamment récent !

#### **Bascule de l'instance destination :**

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

## 8.6 SUPERVISION



- Méta-données
- Statistiques
- Outils

### 8.6.1 Catalogues systèmes - méta-données



- pg\_publication
  - définition des publications
  - \dRp sous psql
- pg\_publication\_tables
  - tables ciblées par chaque publication
- pg\_subscription
  - définition des souscriptions
  - \dRs sous psql

Le catalogue système pg\_publication contient la liste des publications, avec leur méta-données :

```
b1=# SELECT * FROM pg_publication;
```

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

Le catalogue système pg\_publication\_tables contient une ligne par table par publication :

```
b1=# SELECT * FROM pg_publication_tables;
```

| pubname        | schemaname | tablename |
|----------------|------------|-----------|
| publi_complete | public     | t1        |
| publi_complete | public     | t3_1      |
| publi_complete | public     | t3_2      |

|                 |        |      |
|-----------------|--------|------|
| publi_complete  | public | t2   |
| publi_complete  | public | t3_3 |
| publi_complete  | public | t4   |
| publi_partielle | public | t1   |
| publi_partielle | public | t2   |
| publi_t3_1      | public | t3_1 |

On peut en déduire deux versions abrégées :

- la liste des tables par publication :

```
SELECT pubname, array_agg(tablename ORDER BY tablename) AS tables_list
FROM pg_publication_tables
GROUP BY 1
ORDER BY 1;
```

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

- la liste des publications par table :

```
SELECT tablename, array_agg(pubname ORDER BY pubname) AS publications_list
FROM pg_publication_tables
GROUP BY 1
ORDER BY 1 ;
```

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

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

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

### 8.6.2 Vues statistiques



- pg\_stat\_replication
  - statut de réPLICATION
- pg\_stat\_replication\_slots (v14)
  - volumes écrits/envoyés en *streaming* via les slots de réPLICATION logique
- pg\_stat\_subscription
  - état des souscriptions
- pg\_replication\_origin\_status
  - statut des origines de réPLICATION

Comme pour la réPLICATION physique, le retard de réPLICATION est calculable en utilisant les informations de la vue pg\_stat\_replication sur le serveur origine :

```
SELECT * FROM pg_stat_replication ;
-[RECORD 1]-----
pid | 18200
usesysid | 16442
username | logrepli
application_name | subscr_t3_1
client_addr |
client_hostname |
client_port | -1
backend_start | 2017-12-20 10:31:01.13489+01
backend_xmin |
state | streaming
sent_lsn | 0/182D3C8
write_lsn | 0/182D3C8
flush_lsn | 0/182D3C8
replay_lsn | 0/182D3C8
write_lag |
flush_lag |
replay_lag |
sync_priority | 0
sync_state | async
-[RECORD 2]-----
pid | 26606
usesysid | 16442
username | logrepli
application_name | subscr_partielle
client_addr |
client_hostname |
```

```

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

```

Depuis la version 14, une autre vue, `pg_stat_replication_slots` (description complète dans la documentation<sup>2</sup>, permet de suivre les volumétries (octets, nombre de transactions) écrites sur disque (*spilled*) ou envoyées en *streaming* :

```

SELECT * FROM pg_stat_replication_slots \gx

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

```

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

---

<sup>2</sup><https://docs.postgresql.fr/current/monitoring-stats.html#MONITORING-PG-STAT-REPLICATION-SLOTS-VIEW>

```
SELECT * FROM pg_stat_subscription ;
```

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

### 8.6.3 Outils de supervision



- check\_pgactivity
  - replication\_slots
- check\_postgres
  - same\_schema

Peu d'outils ont déjà été mis à jour pour ce type de réPLICATION.

Néanmoins, il est possible de surveiller le retard de réPLICATION via l'état des slots de réPLICATION, comme le propose l'outil `check_pgactivity` (disponible sur [github<sup>3</sup>](#)). Ici, il n'y a pas de retard sur la réPLICATION, pour les trois slots :

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

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

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

```
b1=# INSERT INTO t1 SELECT generate_series(1000000, 2000000);
INSERT 0 1000001
```

<sup>3</sup>[https://github.com/OPMDG/check\\_pgactivity](https://github.com/OPMDG/check_pgactivity)

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

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

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

Il est aussi possible d'utiliser l'action `same_schema` avec l'outil `check_postgres` (disponible aussi sur [github<sup>4</sup>](#)) pour détecter des différences de schémas entre deux serveurs (l'origine et une destination).

---

<sup>4</sup>[https://github.com/bucardo/check\\_postgres/](https://github.com/bucardo/check_postgres/)

## 8.7 MIGRATION MAJEURE PAR RÉPLICATION LOGIQUE



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

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

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

Les étapes sont :

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

Les restrictions liées à la réPLICATION logique subsistent :

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

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

## 8.8 RAPPEL DES LIMITATIONS DE LA RÉPLICATION LOGIQUE NATURE



- Pas de réPLICATION des requêtes DDL
  - ni TRUNCATE en v10
- Pas de réPLICATION des tables partitionnées (< v13)
  - mais réPLICATION possible des partitions
- Pas de réPLICATION vers une table partitionnée (< v13)
- Pas de réPLICATION des valeurs des séquences
- Pas de réPLICATION des LO (table système)
- Contraintes d'unicité obligatoires pour les UPDATE/DELETE
- RéPLICATION déclenchée uniquement lors du COMMIT (< v14)
- Que faire lors des restaurations/bascules ?

La réPLICATION logique native ne réPLIQUE pas les ordres DDL. Elle se base uniquement au niveau des données (donc les ordres DML). Une exception a été faite à partir de la version 11 pour réPLiquer les ordres TRUNCATE car, même s'il s'agit d'un ordre DDL d'après le standard, cet ordre modifie les données d'une table.

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

Toujours avant la version 13, il n'était pas possible d'envoyer des données dans une table partitionnée.

Les valeurs des séquences et les Larges Objects ne sont pas réPLiqués.

Pour les versions inférieures à 14, la réPLICATION logique n'est déclenchée que lors d'un COMMIT. Il n'y a pas d'envoi des données tant que la transaction est en cours, ce qui peut rajouter un délai de réPLICATION pour les transactions longues.



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

Certaines applications supporteront cette limite. Dans d'autres, il sera plus ou moins facile de reprendre la réPLICATION à zéro. Parfois, une réconciliation manuelle sera nécessaire (la présence de clés

primaires peut grandement aider). Dans certains cas, ce problème peut devenir bloquant ou réclamer des développements.

## 8.9 OUTILS DE RÉPLICATION LOGIQUE EXTERNE



- Slony
- Bucardo

### 8.9.1 Slony : Carte d'identité



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

Slony<sup>5</sup> est un très ancien projet libre de réPLICATION pour PostgreSQL. C'était l'outil de choix avant l'arrivée de la réPLICATION native dans PostgreSQL.

### 8.9.2 Slony : Fonctionnalités



- RéPLICATION de tables sélectionnées
- Procédures de bascule
  - *switchover / switchback*
  - *failover / fallback*

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

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

---

<sup>5</sup><http://slony.info/>

### 8.9.3 Slony : Technique



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

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

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

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

### 8.9.4 Slony : Points forts



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

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

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

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

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

### 8.9.5 Slony : Limites



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

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

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

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

### 8.9.6 Slony : Utilisations



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

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

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

Il est fréquent d'utiliser Slony pour des migrations entre deux versions majeures avec une indisponibilité réduite, voire avec un retour en arrière possible. La réPLICATION logique native n'a pas encore atteint le niveau de robustesse de Slony.

Pour plus d'informations sur Slony, n'hésitez pas à lire un de nos articles disponibles sur notre site<sup>6</sup>.

<sup>6</sup>[https://www.dalibo.org/hs44\\_slony\\_la\\_rePLICATION\\_des\\_donnees\\_par\\_trigger](https://www.dalibo.org/hs44_slony_la_rePLICATION_des_donnees_par_trigger)

Le thème des réplications complexes a aussi été abordé lors du PostgreSQL Sessions 2012<sup>7</sup>.

### 8.9.7 Bucardo : Carte d'identité



- Projet libre (BSD)
- Triggers et démons externes
- Intérêt :
  - cluster maître/maître **simple**
  - Plusieurs méthodes de résolution des conflits
  - *Failover* manuel

Bucardo<sup>8</sup> est un des rares outils à proposer du multi-maîtres, avec plusieurs primaires possibles. Il s'agit aussi de réPLICATION logique par triggers, mais sa mise en place est vraiment différente par rapport à Slony.

Bucardo offre plusieurs types de réPLICATION, mais le plus intéressant sur un PostgreSQL moderne est le mode de synchronisation multi-maîtres.

Le problème de toute réPLICATION logique étant la résolution d'éventuels conflits, Bucardo offre plusieurs méthodes de résolution :

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

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

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

<sup>7</sup>[https://www.postgresql-sessions.org/assets/archives/pgsessions3\\_slony.pdf](https://www.postgresql-sessions.org/assets/archives/pgsessions3_slony.pdf)

<sup>8</sup><https://bucardo.org/Bucardo/>

### 8.9.8 Bucardo : Limites



- Aucune technique de propagation des DDL
- Limité à deux nœuds en mode multi-maîtres
- Le réseau doit être fiable
  - peu de retard, pas ou peu de coupures
- Sous Linux/Unix uniquement

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

Le projet est porté surtout par Greg Sabino Mulane, développeur très connu (et apprécié) dans la communauté PostgreSQL. Cela explique un développement apparemment peu actif<sup>9</sup>, même si la correction des bugs est généralement très rapide.

La mise en place de Bucardo est intéressante quand on tient à un cluster maître/maître sur les mêmes tables. Le réseau doit être fiable. La moindre erreur peut tout de même poser des soucis de réconciliation des données. En dehors de cela, il est préférable de se baser sur des solutions comme la réPLICATION interne ou Slony, surtout si l'on peut séparer nettement les données dont chaque serveur est responsable.

---

<sup>9</sup><https://bucardo.org/pipermail/bucardo-announce/>

## 8.10 CONCLUSION



- RéPLICATION logique simple et pratique
  - ...avec ses subtilités

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

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

### 8.10.1 Questions



N'hésitez pas, c'est le moment !

## 8.11 QUIZ



[https://dali.bo/w5\\_quiz](https://dali.bo/w5_quiz)

## 8.12 TRAVAUX PRATIQUES

### 8.12.1 Pré-requis

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

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

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

rm -rf /var/lib/pgsql/15/instance1
rm -rf /var/lib/pgsql/15/instance2
rm -rf /var/lib/pgsql/15/instance3
rm -rf /var/lib/pgsql/15/instance4

export PGSETUP_INITDB_OPTIONS='--data-checksums'
/usr/pgsql-15/bin/postgresql-14-setup initdb instance1
/usr/pgsql-15/bin/postgresql-14-setup initdb instance2
/usr/pgsql-15/bin/postgresql-14-setup initdb instance3
/usr/pgsql-15/bin/postgresql-14-setup initdb instance4

sed -i "s/#port = 5432/port = 5433/" /var/lib/pgsql/15/instance2/postgresql.conf
sed -i "s/#port = 5432/port = 5434/" /var/lib/pgsql/15/instance3/postgresql.conf
sed -i "s/#port = 5432/port = 5435/" /var/lib/pgsql/15/instance4/postgresql.conf

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

$ ps -o pid,cmd fx

 PID CMD
7077 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance4/
7079 _ postgres: logger
7081 _ postgres: checkpointer
7082 _ postgres: background writer
7083 _ postgres: walwriter
7084 _ postgres: autovacuum launcher
7085 _ postgres: logical replication launcher
7056 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance3/
7058 _ postgres: logger
7060 _ postgres: checkpointer
7061 _ postgres: background writer
7062 _ postgres: walwriter
7063 _ postgres: autovacuum launcher
7064 _ postgres: logical replication launcher
7035 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance2/
7037 _ postgres: logger
7039 _ postgres: checkpointer
7040 _ postgres: background writer
```

```

7041 _ postgres: walwriter
7042 _ postgres: autovacuum launcher
7043 _ postgres: logical replication launcher
7015 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
7016 _ postgres: logger
7018 _ postgres: checkpointer
7019 _ postgres: background writer
7020 _ postgres: walwriter
7021 _ postgres: autovacuum launcher
7022 _ postgres: logical replication launcher

```

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

```

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

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

INSERT INTO t1 SELECT i, 't1, ligne'||i FROM generate_series(1, 100) i;
INSERT INTO t2 SELECT i, 't2, ligne'||i FROM generate_series(1, 1000) i;
INSERT INTO t3 SELECT i, 't3, ligne'||i, 1 FROM generate_series(1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne'||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne'||i, 3 FROM generate_series(301, 600) i;

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

```

### 8.12.2 RéPLICATION complète d'une base



**But :** Mettre en place la réPLICATION complète d'une base avec la réPLICATION logIQUE.

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

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

Sur **instance1**, modifier la configuration du paramètre **wal\_level** dans le fichier **postgresql.conf**.

- | Sur **instance1**, modifier la configuration des connexions dans le fichier pg\_hba.conf.
- | Redémarrer **instance1**.
- | Sur **instance2**, créer l'utilisateur de réPLICATION.
- | Sur **instance2**, créer la base **b1**.
- | Sur **instance2**, ajouter dans la base **b1** les tables répliquées (sans contenu).
- | Sur **instance1**, créer la publication pour toutes les tables.
- | Sur **instance2**, créer la souscription.
- | Vérifier sur **instance1**, dans la vue pg\_stat\_replication l'état de la réPLICATION logique.
- | Sur **instance2**, consulter pg\_stat\_subscription.
- | Vérifier que les tables ont le même contenu que sur **instance1** et que les modifications sont également répliquées.

### 8.12.3 RéPLICATION partielle d'une base



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

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

- | Sur **instance1**, créer la publication pour t1 et t2.
- | Sur **instance3**, créer la base **b1**, les tables à répliquer, puis souscrire à la nouvelle publication de **instance1**.
- | Vérifier sur **instance1**, dans la vue pg\_stat\_replication l'état de la réPLICATION logique.

Sur **instance3**, consulter pg\_stat\_subscription.

#### 8.12.4 RéPLICATION croisée



**But :** Mettre en place une réPLICATION croisée avec la réPLICATION logique.

Pour répliquer la partition t3\_1 du serveur **instance1** vers le serveur **instance4**, puis répliquer la partition t3\_2 du serveur **instance4** vers le serveur **instance2** :

Sur **instance1**, créer la publication pour la partition t3\_1.

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

Sur **instance4**, souscrire à cette nouvelle publication de **instance1**. Pour créer la table t3\_1, il faut aussi créer la table mère t3.

Sur **instance4**, adapter la valeur du paramètre wal\_level dans postgresql.conf.

Sur **instance4**, adapter les autorisations dans pg\_hba.conf pour permettre une réPLICATION depuis **instance4**.

Redémarrer **instance4**.

Sur **instance4**, créer la publication pour t3\_4. Il faudra importer la partition t3\_4 et donner les droits de lecture à **logrepli**.

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

Insérer des données dans t3\_4 sur **instance4** et vérifier que la réPLICATION se fait de **instance4** à **instance1**.

#### 8.12.5 RéPLICATION et partitionnement



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

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

- Sur **instance1**, créer une base `bench_part`.
- Sur **instance2**, créer une base `bench_part`.
- Sur **instance1**, utiliser pgbench pour créer la table `pgbench_account`
  - avec un partitionnement de type *hash* et cinq partitions.
- Sur **instance2**, utiliser pgbench pour créer la table `pgbench_account`
  - avec un partitionnement de type *range* et trois partitions,
  - mais sans insérer de données.
- Sur **instance1**, autoriser l'utilisateur de réPLICATION à accéder aux tables.
  - CréeR une publication pour toutes les tables de la base.
- Sur **instance2**, créer la souscription associée. Que constatez-vous ?
- Sur **instance1**, supprimer la publication et la recréer avec l'option `publish_via_partition_root`.
- Sur **instance2**, recréer la souscription.
- Sur **instance1** et **instance2**, compter les lignes dans chaque partition de `pgbench_accounts`. Qu'observez-vous ?

## 8.13 TRAVAUX PRATIQUES (SOLUTIONS)

### 8.13.1 Pré-requis

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

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

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

rm -rf /var/lib/pgsql/15/instance1
rm -rf /var/lib/pgsql/15/instance2
rm -rf /var/lib/pgsql/15/instance3
rm -rf /var/lib/pgsql/15/instance4

export PGSETUP_INITDB_OPTIONS='--data-checksums'
/usr/pgsql-15/bin/postgresql-14-setup initdb instance1
/usr/pgsql-15/bin/postgresql-14-setup initdb instance2
/usr/pgsql-15/bin/postgresql-14-setup initdb instance3
/usr/pgsql-15/bin/postgresql-14-setup initdb instance4

sed -i "s/#port = 5432/port = 5433/" /var/lib/pgsql/15/instance2/postgresql.conf
sed -i "s/#port = 5432/port = 5434/" /var/lib/pgsql/15/instance3/postgresql.conf
sed -i "s/#port = 5432/port = 5435/" /var/lib/pgsql/15/instance4/postgresql.conf

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

$ ps -o pid,cmd fx

 PID CMD
7077 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance4/
7079 _ postgres: logger
7081 _ postgres: checkpointer
7082 _ postgres: background writer
7083 _ postgres: walwriter
7084 _ postgres: autovacuum launcher
7085 _ postgres: logical replication launcher
7056 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance3/
7058 _ postgres: logger
7060 _ postgres: checkpointer
7061 _ postgres: background writer
7062 _ postgres: walwriter
7063 _ postgres: autovacuum launcher
7064 _ postgres: logical replication launcher
7035 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/14/instance2/
7037 _ postgres: logger
7039 _ postgres: checkpointer
7040 _ postgres: background writer
```

```

7041 _ postgres: walwriter
7042 _ postgres: autovacuum launcher
7043 _ postgres: logical replication launcher
7015 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
7016 _ postgres: logger
7018 _ postgres: checkpointer
7019 _ postgres: background writer
7020 _ postgres: walwriter
7021 _ postgres: autovacuum launcher
7022 _ postgres: logical replication launcher

```

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

```

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

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

INSERT INTO t1 SELECT i, 't1, ligne '||i FROM generate_series(1, 100) i;
INSERT INTO t2 SELECT i, 't2, ligne '||i FROM generate_series(1, 1000) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 1 FROM generate_series(1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 3 FROM generate_series(301, 600) i;

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

```

### 8.13.2 RéPLICATION complète d'une base

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

```

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

```

Sur **instance1**, modifier la configuration du paramètre **wal\_level** dans le fichier **postgresql.conf**.

```
wal_level = logical
```

Sur **instance1**, modifier la configuration des connexions dans le fichier **pg\_hba.conf**.

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

Redémarrer **instance1**.

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

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

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

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

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

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

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

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

Sur **instance2**, créer la souscription.

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

Vérifier sur **instance1**, dans la vue pg\_stat\_replication l'état de la réPLICATION logIQUE. Sur **instance2**, consulter pg\_stat\_subscription.

Sur **instance1**:

```
b1=# SELECT * FROM pg_stat_replication \gx
-[RECORD 1]-----+
pid | 7326
usesysid | 16451
username | logrepli
application_name | subscr_complete
client_addr | 127.0.0.1
client_hostname |
client_port | 48094
backend_start | ...
backend_xmin |
state | streaming
sent_lsn | 0/195BF78
write_lsn | 0/195BF78
flush_lsn | 0/195BF78
replay_lsn | 0/195BF78
write_lag |
flush_lag |
replay_lag |
sync_priority | 0
sync_state | async
reply_time | ...
```

Sur **instance2**:

```
b1=# select * from pg_stat_subscription \gx
-[RECORD 1]-----+
subid | 16521
subname | subscr_complete
pid | 7325
relid |
received_lsn | 0/195BF78
last_msg_send_time | ...
last_msg_receipt_time | ...
latest_end_lsn | 0/195BF78
latest_end_time | ...
```

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

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

Sur **instance1**:

```
b1=# INSERT INTO t1 VALUES (101, 't1, ligne 101');
INSERT 0 1

b1=# UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;
UPDATE 1

b1=# DELETE FROM t1 WHERE id_t1=11;
DELETE 1

b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
+-----+
| id_t1 | label_t1 |
+-----+
| 101 | t1, ligne 101 |
| 10 | T1, LIGNE 10 |
+-----+
(2 rows)
```

Sur **instance2**:

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

100

b1=# SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
+-----+
| id_t1 | label_t1 |
+-----+
| 101 | t1, ligne 101 |
| 10 | T1, LIGNE 10 |
+-----+
```

### 8.13.3 RéPLICATION partielle d'une base

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

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

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

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

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

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

Vérifier sur **instance1**, dans la vue pg\_stat\_replication l'état de la réPLICATION logIQUE.

Sur **instance1**:

```
b1=# SELECT * FROM pg_stat_replication \gx
-[RECORD 1]-----+
pid | 7326
usesysid | 16451
username | logrepli
application_name | subscr_complete
client_addr | 127.0.0.1
client_hostname |
client_port | 48094
backend_start | ...
backend_xmin |
state | streaming
sent_lsn | 0/1965548
write_lsn | 0/1965548
flush_lsn | 0/1965548
replay_lsn | 0/1965548
write_lag |
flush_lag |
replay_lag |
sync_priority | 0
sync_state | async
reply_time | ...
-[RECORD 2]-----+
pid | 7511
usesysid | 16451
username | logrepli
application_name | subscr_partielle
client_addr | 127.0.0.1
client_hostname |
client_port | 48124
```

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

Sur **instance3**, consulter pg\_stat\_subscription.

Sur **instance3**:

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

#### 8.13.4 RéPLICATION croisée

Sur **instance1**, créer la publication pour la partition t3\_1.

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

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

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

Sur **instance4**, souscrire à cette nouvelle publication de **instance1**. Pour créer la table t3\_1, il faut aussi créer la table mère t3.

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

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

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

```
wal_level = logical
```

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

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

Redémarrer **instance4**.

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

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

```
GRANT SELECT ON t3_4 TO logrepli;
```

```
CREATE PUBLICATION publi_t3_4
FOR TABLE t3_4;
```

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

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

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

Sur **instance1**:

```
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
 id_t3 | label_t3 | clepartition_t3
-----+-----+-----
(0 rows)

b1=# INSERT INTO t3 VALUES (1001, 't3, ligne 1001', 1);
INSERT 0 1

b1=# SELECT * FROM t3 WHERE id_t3>999;
 id_t3 | label_t3 | clepartition_t3
-----+-----+-----
 1001 | t3, ligne 1001 | 1
```

Sur **instance4**:

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

```

id_t3 | label_t3 | clepartition_t3
-----+-----+-----+
 1001 | t3, ligne 1001 | 1
b1=# INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 4);
INSERT 0 1
b1=# SELECT * FROM t3 WHERE id_t3 > 999;
id_t3 | label_t3 | clepartition_t3
-----+-----+-----+
 1001 | t3, ligne 1001 | 1
 1002 | t3, ligne 1002 | 4

```

Sur **instance1**:

```

b1=# SELECT * FROM t3 WHERE id_t3>999;
id_t3 | label_t3 | clepartition_t3
-----+-----+-----+
 1001 | t3, ligne 1001 | 1
 1002 | t3, ligne 1002 | 4

```

### 8.13.5 RéPLICATION ET PARTITIONNEMENT

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

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

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

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

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

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

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

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

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

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

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

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

La commande échoue avec le message suivant :

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

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

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

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

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

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

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

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

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

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

tableoid	count
pgbench_accounts_1	19851
pgbench_accounts_2	20223
pgbench_accounts_3	19969


```

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

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

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

tableoid	count
pgbench_accounts_1	33334
pgbench_accounts_2	33334
pgbench_accounts_3	33332
¤	100000

(4 rows)
```

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

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

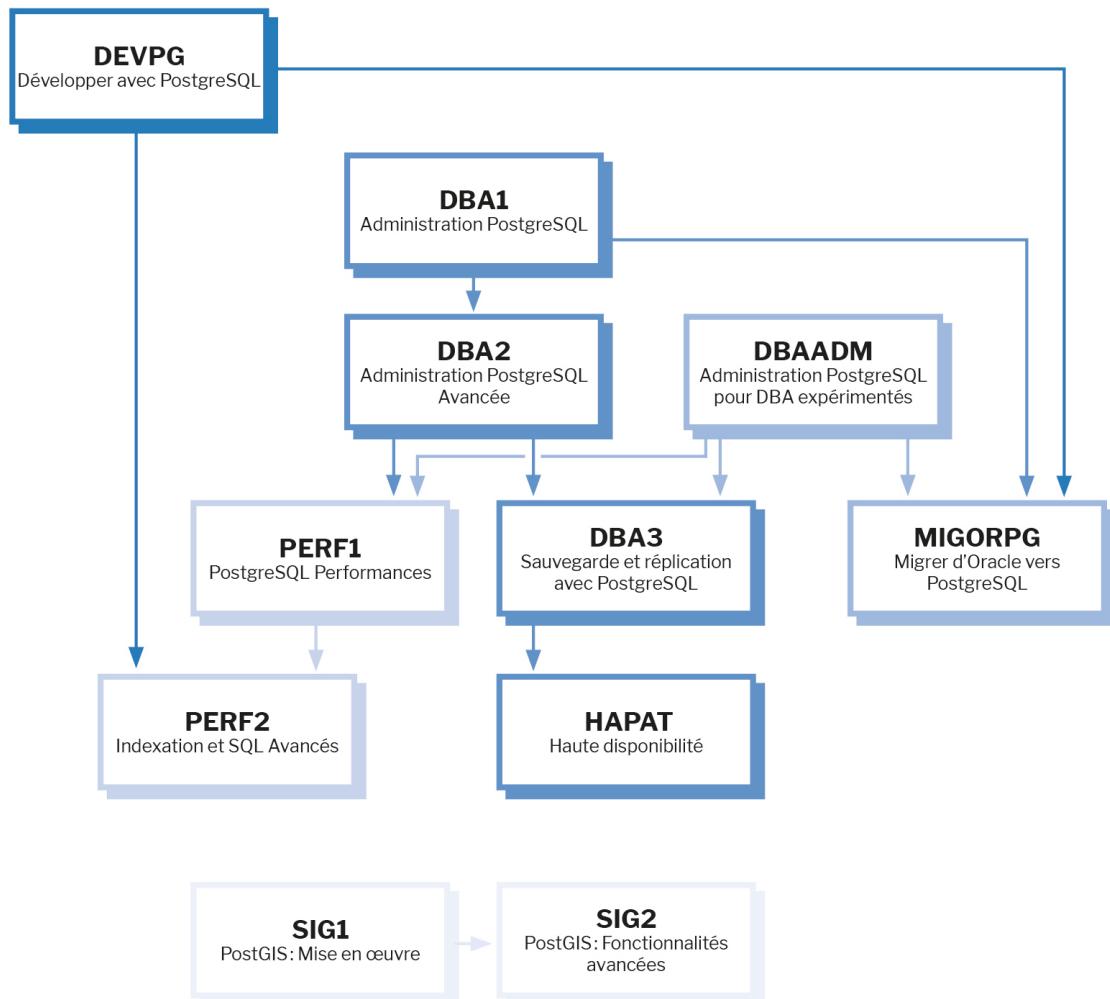


# Les formations Dalibo

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

Pour toute information ou question, n'hésitez pas à nous écrire sur [contact@dalibo.com](mailto: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.



