

## Formation HAPAT

**Haute disponibilité avec Patroni**



**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
<b>1/ Généralités sur la haute disponibilité</b>	<b>5</b>
1.1 Introduction . . . . .	6
1.2 Définitions . . . . .	7
1.2.1 RTO / RPO . . . . .	7
1.2.2 Haute disponibilité de service . . . . .	8
1.2.3 Haute disponibilité de donnée . . . . .	8
1.3 Sauvegardes . . . . .	10
1.3.1 PITR . . . . .	10
1.3.2 PITR et redondance par réPLICATION physique . . . . .	11
1.3.3 Outils PITR . . . . .	11
1.3.4 Bilan PITR . . . . .	12
1.4 RéPLICATION physique . . . . .	13
1.4.1 RéPLICATION et RPO . . . . .	13
1.4.2 RéPLICATION et RTO . . . . .	14
1.4.3 Bilan sur la réPLICATION . . . . .	14
1.5 Bascule automatisée . . . . .	16
1.5.1 Prise de décision . . . . .	16
1.5.2 Mécanique de <i>fencing</i> . . . . .	17
1.5.3 Mécanique d'un Quorum . . . . .	18
1.5.4 Mécanique du watchdog . . . . .	19
1.5.5 Storage Base Death . . . . .	20
1.5.6 Bilan des solutions anti-split-brain . . . . .	21
1.6 Implication et risques de la bascule automatique . . . . .	23
1.6.1 Questions . . . . .	24
<b>2/ Solutions de réPLICATION</b>	<b>25</b>
2.1 Préambule . . . . .	26
2.1.1 Au menu . . . . .	26
2.1.2 Objectifs . . . . .	27
2.2 Rappels théoriques . . . . .	28
2.2.1 Cluster, primaire, secondaire, <i>standby</i> ... . . . . .	28
2.2.2 RéPLICATION asynchrone asymétrique . . . . .	29
2.2.3 RéPLICATION asynchrone symétrique . . . . .	30
2.2.4 RéPLICATION synchrone asymétrique . . . . .	31
2.2.5 RéPLICATION synchrone symétrique . . . . .	32

2.2.6	Diffusion des modifications . . . . .	33
2.3	RéPLICATION interne physique . . . . .	35
2.3.1	Log Shipping . . . . .	36
2.3.2	Streaming replication . . . . .	37
2.3.3	<i>Warm Standby</i> . . . . .	37
2.3.4	<i>Hot Standby</i> . . . . .	38
2.3.5	Exemple . . . . .	39
2.3.6	RéPLICATION interne . . . . .	40
2.3.7	RéPLICATION en cascade . . . . .	41
2.4	RéPLICATION interne logique . . . . .	42
2.4.1	RéPLICATION logique - Fonctionnement . . . . .	42
2.5	RéPLICATION externe . . . . .	44
2.6	Sharding . . . . .	45
2.7	RéPLICATION bas niveau . . . . .	47
2.7.1	RAID . . . . .	47
2.7.2	DRBD . . . . .	48
2.7.3	SAN Mirroring . . . . .	48
2.8	Conclusion . . . . .	49
2.8.1	Questions . . . . .	49
2.9	Quiz . . . . .	50
<b>3/ RéPLICATION physique : rappels</b>		<b>51</b>
3.1	Introduction . . . . .	52
3.1.1	Au menu . . . . .	52
3.2	Mise en place de la réPLICATION par streaming . . . . .	53
3.2.1	Serveur primaire (1/2) - configuration . . . . .	53
3.2.2	Serveur primaire (2/2) - authentification . . . . .	55
3.2.3	Serveur secondaire (1/3) - copie des données . . . . .	56
3.2.4	Serveur secondaire (2/3) - configuration . . . . .	57
3.2.5	Serveur secondaire (3/3) - démarrage . . . . .	58
3.2.6	Processus . . . . .	59
3.3	Promotion . . . . .	60
3.3.1	Attention au split-brain! . . . . .	60
3.3.2	Vérification avant promotion . . . . .	61
3.3.3	Promotion du standby : méthode . . . . .	62
3.3.4	Promotion du standby : déroulement . . . . .	62
3.3.5	Opérations après promotion du standby . . . . .	63
3.3.6	Retour à l'état stable . . . . .	64
3.3.7	Retour à l'état stable, suite . . . . .	65
3.4	Conclusion . . . . .	67
3.5	Quiz . . . . .	68
3.6	Installation de PostgreSQL depuis les paquets communautaires . . . . .	69
3.6.1	Sur Rocky Linux 8 . . . . .	69
3.6.2	Sur Red Hat 7 / Cent OS 7 . . . . .	71
3.6.3	Sur Debian / Ubuntu . . . . .	71

3.6.4	Accès à l'instance sur le serveur même . . . . .	73
3.7	Travaux pratiques . . . . .	76
3.7.1	RéPLICATION ASYNCHRONE EN FLUX AVEC UN SEUL SECONDAIRE . . . . .	76
3.7.2	PROMOTION DE L'INSTANCE SECONDAIRE . . . . .	76
3.7.3	RETOUR À LA NORMALE . . . . .	77
3.8	Travaux pratiques (solutions) . . . . .	78
3.8.1	RÉPLICATION ASYNCHRONE EN FLUX AVEC UN SEUL SECONDAIRE . . . . .	78
3.8.2	PROMOTION DE L'INSTANCE SECONDAIRE . . . . .	81
3.8.3	RETOUR À LA NORMALE . . . . .	82
<b>4/</b>	<b>Haute disponibilité de service</b>	<b>85</b>
4.1	Préambule . . . . .	86
4.2	Shared storage . . . . .	87
4.3	Share nothing . . . . .	88
4.4	Patroni . . . . .	89
4.5	Pacemaker . . . . .	91
4.6	Accès aux ressources . . . . .	92
4.7	Comparatif des solutions et choix . . . . .	93
4.7.1	Le jeu en vaut-il la chandelle ? . . . . .	93
4.8	Conclusion . . . . .	95
<b>5/</b>	<b>Patroni : Architecture et fonctionnement</b>	<b>97</b>
5.1	Au menu . . . . .	98
5.2	Architecture générale . . . . .	99
5.3	L'algorithme Raft . . . . .	101
5.3.1	Raft : Journal et machine à états . . . . .	102
5.3.2	Élection d'un leader . . . . .	103
5.3.3	RéPLICATION DU JOURNAL . . . . .	104
5.3.4	Sécurité & cohérence . . . . .	105
5.3.5	Majorité et tolérance de panne . . . . .	106
5.3.6	Tolérance de panne : Tableau récapitulatif . . . . .	106
5.3.7	Interaction avec les clients . . . . .	107
5.3.8	Raft en action . . . . .	107
5.4	etcd . . . . .	108
5.4.1	etcd et Raft . . . . .	108
5.4.2	Serveur de configurations distribuées . . . . .	109
5.4.3	Multiples nœuds . . . . .	109
5.4.4	Élection . . . . .	110
5.4.5	Conséquences . . . . .	110
5.4.6	Principaux paramètres . . . . .	111
5.4.7	Questions . . . . .	111
5.5	Installation d'un cluster etcd . . . . .	112
5.5.1	Sur Red Hat 7/CentOS 7 . . . . .	112
5.5.2	Sur Rocky Linux 8 . . . . .	112
5.5.3	Sur Debian/Ubuntu . . . . .	113
5.5.4	Configuration de etcd . . . . .	114

5.5.5	Gestion du service . . . . .	115
5.5.6	Traces . . . . .	115
5.5.7	Tests du cluster etcd . . . . .	115
5.5.8	Configuration du service pour Patroni . . . . .	116
5.5.9	Exemple de sauvegarde d'information . . . . .	117
5.6	Patroni . . . . .	119
5.6.1	Définition . . . . .	119
5.6.2	Mécanismes mis en œuvre . . . . .	119
5.6.3	Bascule automatique . . . . .	120
5.6.4	Définition : <i>split-brain</i> . . . . .	121
5.6.5	Leader lock de Patroni . . . . .	121
5.6.6	Heartbeat . . . . .	122
5.6.7	Bootstrap de noeud . . . . .	122
5.6.8	Cas particulier du multisite . . . . .	123
5.6.9	Deux sites . . . . .	123
5.6.10	Configuration de Patroni . . . . .	125
5.6.11	patronictl . . . . .	155
5.6.12	<i>endpoints</i> de l'API REST . . . . .	163
5.6.13	Proxy, VIP et Poolers de connexions . . . . .	164
5.6.14	Questions . . . . .	170
5.7	Installation de Patroni depuis les paquets communautaires . . . . .	171
5.7.1	Sur Red Hat 7 / CentOS 7 . . . . .	171
5.7.2	Sur Rocky Linux 8 . . . . .	171
5.7.3	Sur Debian / Ubuntu . . . . .	172
5.7.4	Configuration de Patroni . . . . .	172
5.8	Quiz . . . . .	179
5.9	Travaux pratiques . . . . .	180
5.9.1	Raft . . . . .	180
5.9.2	etcd : installation . . . . .	181
5.9.3	etcd : manipulation (optionnel) . . . . .	181
5.9.4	Patroni : installation . . . . .	182
5.9.5	Patroni : utilisation . . . . .	182
5.10	Travaux pratiques (solutions) . . . . .	184
5.10.1	Raft . . . . .	184
5.10.2	etcd : installation . . . . .	185
5.10.3	etcd : manipulation (optionnel) . . . . .	185
5.10.4	Patroni : installation . . . . .	190
5.10.5	Patroni : utilisation . . . . .	195
5.11	Introduction à pgbench . . . . .	202
5.11.1	Installation . . . . .	202
5.11.2	Générer de l'activité . . . . .	202
<b>Les formations Dalibo</b>		<b>205</b>
Cursus des formations . . . . .		205
Les livres blancs . . . . .		206

Téléchargement gratuit . . . . .	206
----------------------------------	-----



## 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.com<sup>1</sup>](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.

---

<sup>1</sup><mailto:formation@dalibo.com>

## Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA<sup>2</sup>**. 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 déposées<sup>3</sup> par PostgreSQL Community Association of Canada.

## Sur ce document

---

<b>Formation</b>	Formation HAPAT
<b>Titre</b>	Haute disponibilité avec Patroni
<b>Révision</b>	23.09
<b>ISBN</b>	N/A
<b>PDF</b>	<a href="https://dali.bo/hapat_pdf">https://dali.bo/hapat_pdf</a>

---

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

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

---

**EPUB** [https://dali.bo/hapat\\_epub](https://dali.bo/hapat_epub)

**HTML** [https://dali.bo/hapat\\_html](https://dali.bo/hapat_html)

**Slides** [https://dali.bo/hapat\\_slides](https://dali.bo/hapat_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/ Généralités sur la haute disponibilité**

## 1.1 INTRODUCTION



- Définition de « Haute Disponibilité »
- Contraintes à définir
- Contraintes techniques
- Solutions existantes

La haute disponibilité est un sujet complexe. Plusieurs outils libres coexistent au sein de l'écosystème PostgreSQL, chacun abordant le sujet d'une façon différente.

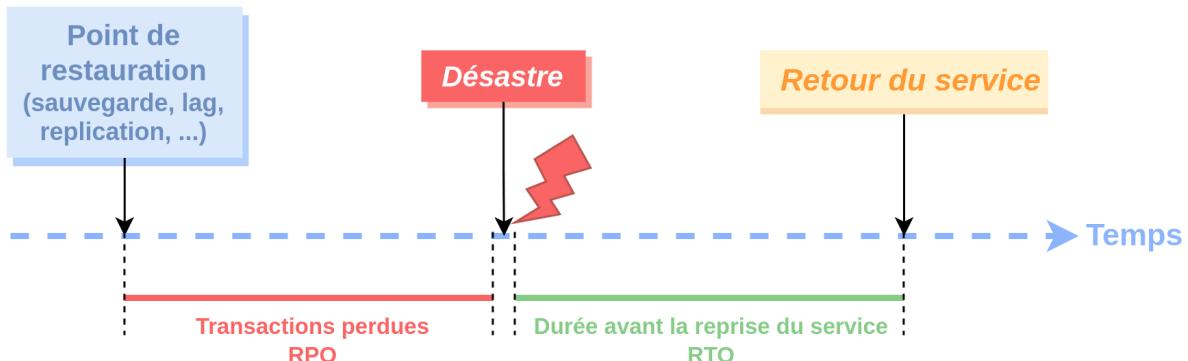
Ce document clarifie la définition de «haute disponibilité», des méthodes existantes et des contraintes à considérer. L'objectif est d'aider la prise de décision et le choix de la solution.

## 1.2 DÉFINITIONS



- RTO / RPO
- Haute Disponibilité de données / de service

### 1.2.1 RTO / RPO



Deux critères essentiels permettent de contraindre le choix d'une solution : le RTO (*recovery time objective*) et le RPO (*recovery point objective*).

Le RTO représente la durée maximale d'interruption de service admissible, depuis la coupure de service jusqu'à son rétablissement. Il inclut le délai de détection de l'incident, le délai de prise en charge et le temps de mise en œuvre des actions correctives. Un RTO peut tendre vers zéro mais ne l'atteint jamais parfaitement. Une coupure de service est le plus souvent **inévitable**, aussi courte soit-elle.

Le RPO représente la durée maximale d'activité de production déjà réalisée que l'on s'autorise à perdre en cas d'incident. Contrairement au RTO, le RPO peut atteindre l'objectif de zéro perte.

Les deux critères sont complémentaires. Ils ont une influence **importante** sur le choix d'une solution **et** sur son coût total. Plus les RTO et RPO sont courts, plus la solution est complexe. Cette complexité se répercute directement sur le coût de mise en œuvre, de formation et de maintenance.

Le coût d'une architecture est exponentiel par rapport à sa disponibilité.

### 1.2.2 Haute disponibilité de service



- Continuité d'activité malgré incident
- Redondance à tous les niveaux
  - réseaux, stockage, serveurs, administrateurs...
- Automatisation des bascules

La **Haute Disponibilité de service** définit les moyens techniques mis en œuvre pour garantir une continuité d'activité suite à un incident sur un service.

La haute disponibilité de service nécessite de redonner tous les éléments nécessaires à l'activité du service : l'alimentation électrique, ses accès réseaux, le réseau lui-même, les serveurs, le stockage, les administrateurs, etc.

En plus de cette redondance, une technique de réPLICATION synchrone ou asynchrone est souvent mise en œuvre afin de maintenir à l'identique ou presque les serveurs redondés.

Pour que la disponibilité ne soit pas affectée par le temps de réaction des humains, on peut rechercher à automatiser les bascules vers un serveur sain en cas de problème.

### 1.2.3 Haute disponibilité de donnée



- Perte faible ou nulle de données après incident
  - redonner les données
  - garantir les écritures à plusieurs endroits
- Contradictoire avec un service très dégradé
  - arbitrage avec disponibilité de service et budget

La **Haute disponibilité des données** définit les moyens techniques mis en œuvre pour garantir une perte faible voire nulle de données en cas d'incident. Ce niveau de disponibilité des données est assuré en redondant les données sur plusieurs systèmes physiques distincts et en assurant que chaque écriture est bien réalisée sur plusieurs d'entre eux.

Dans le cas d'une réPLICATION synchrone entre les systèmes, les écritures sont suspendues tant qu'elles ne peuvent être validées de façon fiable sur au moins deux systèmes.



Autrement dit, la haute disponibilité des données et la haute disponibilité de service sont contradictoires, le premier nécessitant d'interrompre le service en écriture si l'ensemble ne repose que sur un seul système.

Par exemple, un RAID 1 fonctionnant sur un seul disque suite à un incident n'est PAS un environnement à haute disponibilité des données, mais à haute disponibilité de service.

La position du curseur entre la haute disponibilité de service et la haute disponibilité de données guide aussi le choix de la solution. S'il est possible d'atteindre le double objectif, l'impact sur les solutions possibles et le coût est une fois de plus important.

## 1.3 SAUVEGARDES



- Composant déjà présent
- Travail d'optimisation à effectuer
- RTO de quelques minutes possibles
- RPO de quelques minutes (secondes ?) facilement

Les différentes méthodes de sauvegardes de PostgreSQL (logique avec pg\_dump, ou PITR avec pg-BackRest ou d'autres outils) sont souvent sous-estimées. Elles sont pourtant un élément essentiel de toute architecture qui est souvent déjà présent.

Investir dans l'optimisation des sauvegardes peut déjà assurer un certain niveau de disponibilité de votre service, à moindre coût.

Quoi qu'il en soit, la sauvegarde est un élément crucial de toute architecture. Ce sujet doit toujours faire partie de la réflexion autour de la disponibilité d'un service.

### 1.3.1 PITR



- Sauvegarde incrémentale binaire
- Optimiser la sauvegarde complète
- Optimiser la restauration complète (RTO)
- Ajuster l'archivage au RPO désiré

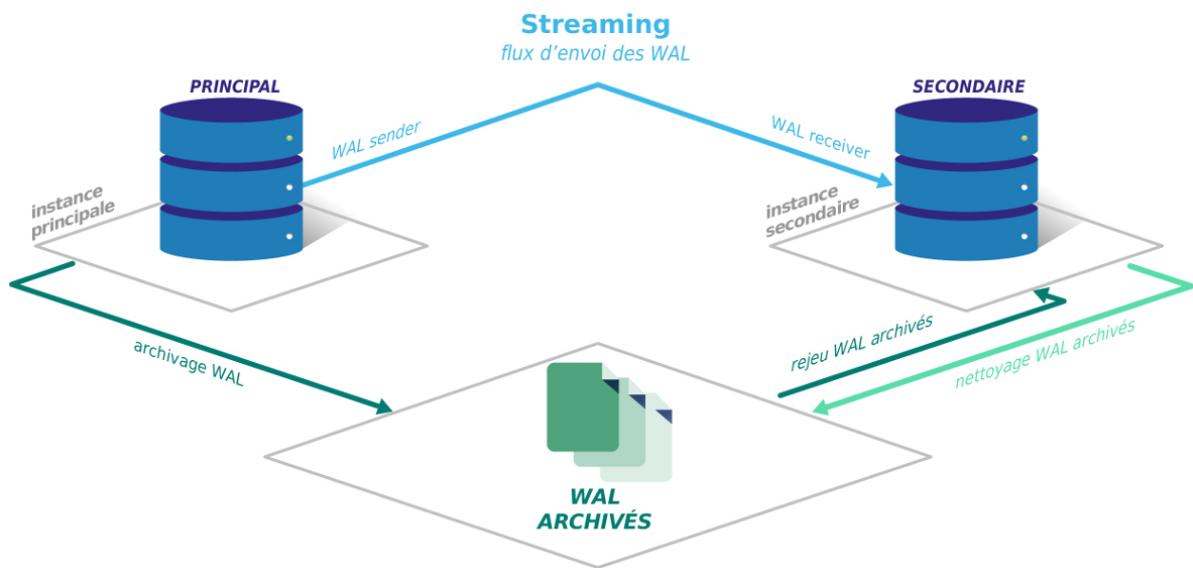
La sauvegarde PITR est une méthode permettant de restaurer une instance PostgreSQL à n'importe quel instant durant la fenêtre de rétention définie, par exemple les dernières 24 heures. Le temps de restauration (RTO) dépend de deux variables : le volume de l'instance et son volume d'écriture.

Avec le bon matériel, les bonnes pratiques et une politique de sauvegarde adaptée, il est possible d'atteindre un RTO de quelques minutes, dans la plupart des cas.

La maîtrise du RPO (perte de données) repose sur la fréquence d'archivage des journaux de transactions. Un RPO d'une minute est tout à fait envisageable. En-dessous, nous entrons dans le domaine de la réPLICATION en *streaming*, soit pour les sauvegardes (outil pg\_receivewal), soit vers une instance secondaire. Nous abordons ce sujet dans un futur chapitre.

### 1.3.2 PITR et redondance par réPLICATION physique

#### RÉPLICATION INTERNE POSTGRESQL



Il est possible d'utiliser les journaux de transactions archivés dans le cadre de la réPLICATION physique. Ces archives deviennent alors un second canal d'échange entre l'instance primaire et ses secondaires, apportant une redondance à la réPLICATION elle-même.

### 1.3.3 Outils PITR



- Barman
- pgBackRest

Parmi les outils existants et éprouvés au sein de la communauté, nous pouvons citer les deux ci-dessus.

Un module de nos formations les traite en détail<sup>1</sup>.

<sup>1</sup>[https://dali.bo/i4\\_html](https://dali.bo/i4_html)

### 1.3.4 Bilan PITR



- Utiliser un outil libre issu de l'écosystème PostgreSQL
- Fiabilise l'architecture
- Facilite la mise en œuvre et l'administration
- Couvre déjà certains besoins de disponibilité
- Nécessite une intervention humaine
- Nécessite une supervision fiable

Le principal point faible de la sauvegarde PITR est le temps de prise en compte de l'incident et donc d'intervention d'un administrateur.

Enfin, la sauvegarde PITR doit être surveillée de très près par les équipes d'administration au travers d'une supervision adaptée.

## 1.4 RÉPLICATION PHYSIQUE



- Réplique les écritures via les journaux de transactions
- Entretient une ou plusieurs instances clones
- Intégrée à PostgreSQL
- Facilité de mise en œuvre
- Réduit RPO/RTO par rapport au PITR
- Plus de matériel
- Architecture et maintenance plus complexes
- Haute disponibilité des données

La réPLICATION physique interne de PostgreSQL réPLIQUE le contenu des journaux de transactions. Les instances secondaires sont considérées comme des « clones » de l'instance primaire.

Avec peu de configuration préalable, il est possible de créer des instances secondaires directement à partir de l'instance primaire ou en restaurant une sauvegarde PITR.

La mécanique de réPLICATION est très efficace, car elle ne réPLIQUE que les modifications binaires effectuées dans les tables et les index.

Cette étape assure déjà une haute disponibilité de données, ces dernières étant présentes sur plusieurs serveurs distincts.

La réPLICATION permet d'atteindre un RPO plus faible que celui du PITR, au prix d'investissement plus important (redondance du matériel), d'une complexification de l'architecture et de sa maintenance. Le RPO deviendra également plus lié au temps de réACTION qu'à la bascule technique.

### 1.4.1 RéPLICATION et RPO



- RéPLICATION asynchrone ou synchrone
- Nécessite un réseau très fiable et performant
- Asynchrone : RPO dépendant du volume d'écriture
  - RPO < 1s hors maintenance et chargement en masse
- Synchrone : RPO = 0
  - 2 secondaires minimum
  - impact sur les performances !

PostgreSQL supporte la réPLICATION asynchrone ou synchrone.

La réPLICATION asynchrone autorise un retard entre l'instance primaire et ses secondaires, ce qui implique un RPO supérieur à 0. Ce retard dépend directement du volume d'écriture envoyé par le primaire et de la capacité du réseau à diffuser ce volume, donc son débit. Une utilisation OLTP a un retard typique inférieur à la seconde. Ce retard peut cependant être plus important lors des périodes de maintenance (VACUUM, REINDEX, manipulation en masse de données, etc).

La réPLICATION synchrone s'assure que chaque écriture soit présente sur au moins deux instances avant de valider une transaction. Ce mode permet d'atteindre un RPO de zéro, mais impose d'avoir au minimum trois nœuds dans le cluster, autorisant ainsi la perte complète d'un serveur sans bloquer les écritures. En effet, avec deux nœuds seulement, la disponibilité de données n'est plus assurée : la perte d'un nouveau serveur entraînerait le blocage des écritures qui ne pourraient plus être synchrones.

De plus, le nombre de transactions par seconde dépend directement de la latence du réseau : chaque transaction doit attendre la propagation vers un secondaire et le retour de sa validation.

### 1.4.2 RéPLICATION et RTO



- Bascule manuelle
- Promotion d'une instance en quelques secondes

La réPLICATION seule n'assure pas de disponibilité de service en cas d'incident.

Comme pour les sauvegardes PITR, le RTO dépend principalement du temps de prise en charge de l'incident par un opérateur. Une fois la décision prise, la promotion d'un serveur secondaire en production ne nécessite qu'une commande et ne prend typiquement que quelques secondes.

Reste ensuite à faire converger les connexions applicatives vers la nouvelle instance primaire.

### 1.4.3 Bilan sur la réPLICATION



- $0 \leq \text{RPO} < \text{PITR}$
- $\text{RTO} = \text{prise en charge} + 30s$
- Simple à mettre en œuvre
- Investissement en coût et humain plus important

La réPLICATION nécessite donc au minimum deux serveurs, voire trois en cas de réPLICATION synchrone. À ce coût s'ajoutent plusieurs autres plus ou moins cachés :

- le réseau se doit d'être redondé et fiable surtout en cas de réPLICATION synchrone ;
- la formation des équipes d'administration ;
- la mise en œuvre des procédures de construction et de bascule ;
- une supervision plus fine et maîtrisée des équipes.

## 1.5 BASCULE AUTOMATISÉE



- Détection d'anomalie et bascule automatique
- HA de service :
  - réduit le temps de prise en charge
- Plusieurs solutions en fonction du besoin
- Beaucoup de contraintes !

Une bascule automatique lors d'un incident permet de réduire le temps d'indisponibilité d'un service au plus bas, assurant ainsi une haute disponibilité de service.

Néanmoins, automatiser la détection d'incident et la prise de décision de basculer un service est un sujet très complexe, difficile à bien appréhender et maintenir, d'autant plus dans le domaine des SGBD.

### 1.5.1 Prise de décision



- La détection d'anomalie est naïve !
- L'architecture doit pouvoir éviter un *split-brain*
- Solutions éprouvées :
  - *fencing*
  - *quorum*
  - *watchdog*
  - SBD
- Solutions le plus souvent complémentaires.

Quelle que soit la solution choisie pour détecter les anomalies et déclencher une bascule, celle-ci est toujours très naïve. Contrairement à un opérateur humain, la solution n'a pas de capacité d'analyse et n'a pas accès aux mêmes informations. En cas de non-réponse d'un élément du cluster, il est impossible de déterminer dans quel état il se trouve précisément. Sous une charge importante ? Serveur arrêté brutalement ou non ? Réseau coupé ?

Il y a une forte probabilité de *split-brain* si le cluster se contente d'effectuer une bascule sans se préoccuper de l'ancien primaire. Dans cette situation, deux serveurs se partagent la même ressource (IP ou disque ou SGBD) sans le savoir. Corriger le problème et reconsolider les données est fastidieux et

entraîne une indisponibilité plus importante qu'une simple bascule manuelle avec analyse et prise de décision humaine.

Quatre mécaniques permettent de se prémunir plus ou moins d'un *split-brain* : le *fencing*, le *quorum*, le *watchdog* et le SBD (*Storage Based Death*). La plupart doivent être combinées pour fonctionner de façon optimale.

### 1.5.2 Mécanique de *fencing*



- Isole un serveur/ressource
  - électriquement
  - arrêt via IPMI, hyperviseur
  - coupe les réseaux
- Utile :
  - pour un serveur muet ou fantôme (*rogue node*)
  - lorsque l'arrêt d'une ressource est perturbé
- Déclenché depuis un des nœuds du cluster
- Nécessite une gestion fine des droits
- Supporté par Pacemaker, embryonnaire dans Patroni

Le *fencing* (clôture) isole un serveur ou une ressource de façon active. Suite à une anomalie, et **avant** la bascule vers le secours prévu, le composant fautif est isolé afin qu'il ne puisse plus interférer avec la production.

Il existe au moins deux anomalies où le *fencing* est incontournable. La première concerne le cas d'un serveur qui ne répond plus au cluster. Il est alors impossible de définir quelle est la situation sur le serveur. Est-il encore vivant ? Les ressources sont-elles encore actives ? Ont-elles encore un comportement normal ? Ont-elles encore accès à l'éventuel disque partagé ? Dans cette situation, la seule façon de répondre avec certitude à ces questions est d'éteindre le serveur. L'action définit avec certitude que les ressources y sont toutes inactives.

La seconde anomalie où le *fencing* est essentiel concerne l'arrêt des ressources. Si le serveur est disponible, communique, mais n'arrive pas à éteindre une ressource (problème technique ou timeout), le *fencing* permet « d'escalader » l'extinction de la ressource en extinction du serveur complet.

Il est aussi possible d'isoler un serveur d'une ressource. Le serveur n'est pas éteint, mais son accès à certaines ressources cruciales est coupé, l'empêchant ainsi de corrompre le cluster. L'isolation peut concerner l'accès au réseau Ethernet ou à un disque partagé par exemple.

Il existe donc plusieurs techniques pour un *fencing*, mais il doit toujours être rapide et efficace. Pas de

demi-mesures ! Les méthodes les plus connues soit coupent le courant, donc agissent sur l'UPS<sup>2</sup>, ou le PDU<sup>3</sup> ; soit éteignent la machine au niveau matériel via l'IPMI<sup>4</sup> ; soit éteignent la machine virtuelle rapidement via son hyperviseur ; soit coupent l'accès au réseau, SAN ou Ethernet.

Par conséquent, cette mécanique nécessite souvent de pouvoir gérer finement les droits d'accès à des opérations d'administration lourdes. C'est le cas par exemple au travers des communautés du protocole SNMP, ou la gestion de droits dans les ESX VMware, les accès au PDU, etc.

### 1.5.3 Mécanique d'un Quorum



- Chaque serveur possède un ou plusieurs votes
- Utile en cas de partition réseau
- La partition réseau qui a le plus de votes détient le quorum
- La partition qui détient le quorum peut héberger les ressources
- La partition sans quorum doit arrêter toute ressource
- Attention au retour d'une instance dans le cluster
- Supporté par Pacemaker et Patroni (via DCS)

La mécanique du quorum attribue à chaque nœud un (ou plusieurs) vote. Le cluster n'a le droit d'héberger des ressources que s'il possède la majorité absolue des voix. Par exemple, un cluster à 3 nœuds requiert 2 votes pour pouvoir démarrer les ressources, 3 pour un cluster à 5 nœuds, etc.

Lorsque qu'un ou plusieurs nœuds perdent le quorum, ceux-ci doivent arrêter les ressources qu'ils hébergent.

Il est conseillé de maintenir un nombre de nœuds impair au sein du cluster, mais plusieurs solutions existent en cas d'égalité (par exemple par ordre d'id, par poids, serveurs « témoins » ou arbitre, etc).

Le quorum permet principalement de gérer les incidents liés au réseau, quand « tout va bien » sur les serveurs eux-mêmes et qu'ils peuvent éteindre leurs ressources sans problème, à la demande.

Dans le cadre de PostgreSQL, il faut porter une attention particulière au moment où des serveurs isolés rejoignent de nouveau le cluster. Si l'instance primaire a été arrêtée par manque de quorum, cette dernière pourrait ne pas raccrocher correctement avec le nouveau primaire, voire corrompre ses propres fichiers de données. Effectivement, il est impossible de déterminer le type d'écriture ayant eu lieu sur l'ancien primaire entre la déconnexion réelle du reste du cluster, l'état de sa réPLICATION, de ses backends et de son arrêt total.

Pacemaker intègre la gestion du quorum et peut aussi utiliser un serveur de gestion de vote appelé corosync-qnetd. Ce dernier est utile en tant que tiers pour gérer le quorum de plusieurs clusters Pacemaker à deux nœuds par exemple.

<sup>2</sup>[https://fr.wikipedia.org/wiki/Alimentation\\_sans\\_interruption](https://fr.wikipedia.org/wiki/Alimentation_sans_interruption)

<sup>3</sup>[https://fr.wikipedia.org/wiki/Unit%C3%A9\\_de\\_distribution\\_d%27%C3%A9nergie](https://fr.wikipedia.org/wiki/Unit%C3%A9_de_distribution_d%27%C3%A9nergie)

<sup>4</sup>[https://fr.wikipedia.org/wiki/Intelligent\\_Platform\\_Management\\_Interface](https://fr.wikipedia.org/wiki/Intelligent_Platform_Management_Interface)

Patroni repose sur un DCS<sup>5</sup> extérieur, par exemple etcd<sup>6</sup>, pour stocker l'état du serveur et prendre ses décisions. La responsabilité de la gestion du quorum est donc déléguée au DCS, dont l'architecture robuste est conçue pour toujours présenter des données fiables et de référence à ses clients (ici Patroni).

#### 1.5.4 Mécanique du watchdog



- Équipement matériel intégré partout
  - au pire, softdog (moins fiable)
  - Compte à rebours avant redémarrage complet du serveur
  - Doit être ré-armé par un composant applicatif du serveur, **périodiquement**
  - Permet de déclencher du *self-fencing* rapide et fiable
  - Complémentaire au quorum, « fencing du pauvre »
  - Accélère certaines réactions du serveur
  - Patroni et Pacemaker : oui

Tous les ordinateurs sont désormais équipés d'un *watchdog*. Par exemple, sur un ordinateur portable Dell Latitude, nous trouvons :

iTCO\_wdt : Intel TCO WatchDog Timer Driver v1.11

Sur un Raspberry Pi modèle B :

bcm2835-wdt 20100000.watchdog : Broadcom BCM2835 watchdog timer

Au besoin, il est aussi possible d'ajouter plusieurs autres *watchdog* grâce à des cartes PCI par exemple, bien que ce ne soit pas nécessaire dans notre cas.

Concernant les machines virtuelles, une configuration supplémentaire est souvent nécessaire pour avoir accès à un *watchdog* virtualisé.

En dernier recours, il est possible de demander au noyau Linux lui-même de jouer le rôle de *watchdog* grâce au module softdog. Néanmoins, cette méthode est moins fiable qu'un *watchdog* matériel car il nécessite que le système d'exploitation fonctionne toujours correctement et qu'au moins un des CPU soit disponible. Cet article<sup>7</sup> entre plus en détails.

<sup>5</sup>Distributed Control System

<sup>6</sup><https://etcd.io/>

<sup>7</sup><http://www.beekhof.net/blog/2019/savaged-by-softdog>

Le principe du *watchdog* peut être résumé par : « nourris le chien de garde avant qu'il ait faim et te mange ». En pratique, un *watchdog* est un compte à rebours avant la réinitialisation brutale du serveur. Si ce compte à rebours n'est pas régulièrement ré-armé, le serveur est alors redémarré.

Un *watchdog* surveille donc passivement un processus et assure que ce dernier est toujours disponible et sain. Dans le cadre d'un cluster en haute disponibilité, le processus rendant compte de sa bonne forme au *watchdog* est le clusterware.

Notez qu'un *watchdog* permet aussi de déclencher un *self-fencing* rapide et fiable en cas de besoin. Il permet par exemple de résoudre rapidement le cas de l'arrêt forcé d'une ressource, déjà présenté dans le chapitre consacré au *fencing*.

Patroni et Pacemaker sont tous deux capables d'utiliser un *watchdog* sur chaque nœud. Pour Patroni, il n'est armé que sur l'instance primaire. Pour Pacemaker, il est armé sur tous les nœuds.

### 1.5.5 Storage Base Death



- L'une des méthodes historique de *fencing*
- Un ou plusieurs disques partagés
  - où les nœuds s'échangent des messages
- Un *watchdog* par nœud
- Message *poison pill* pour demander à un nœud distant de s'auto-fencer
- *Self-fencing* en cas de perte d'accès aux disques...
  - ...si Pacemaker confirme lui aussi une anomalie
- Patroni : émulé

Le *Storage Base Death* est une méthode assez ancienne. Elle utilise un ou plusieurs disques partagés (pour la redondance), montés sur tous les nœuds du cluster à la fois. L'espace nécessaire sur chaque disque est très petit, de l'ordre de quelques mégaoctets pour plusieurs centaines de nœuds (le démon *sbd* utilise 1 à 4 Mo pour 255 nœuds). Cet espace disque est utilisé comme support de communication entre les nœuds qui y échangent des messages.

Le clusterware peut isoler un nœud en déposant un message *poison pill* à son attention. Le destinataire s'auto-fence grâce à son *watchdog* dès qu'il lit le message. De plus, un nœud s'auto-fence aussi s'il n'accède plus au stockage et que Pacemaker ou Corosync indiquent eux aussi une anomalie. Ce comportement défensif permet de s'assurer qu'aucun ordre de *self-fencing* ne peut se perdre.

Grâce au SBD, le cluster est assuré que le nœud distant peut effectuer son *self-fencing* soit par perte de son accès au disque partagé, soit par réception du *poison pill*, soit à cause d'une anomalie qui a empêché le clusterware d'assumer le ré-armement du *watchdog*.

Un exemple détaillé de mise en œuvre avec sdb est disponible dans la documentation de Suse, à l'adresse suivante : <https://documentation.suse.com/sle-ha/15-SP4/html/SLE-HA-all/cha-ha-storage-protect.html>.

Pacemaker supporte ce type d'architecture. Patroni ne supporte pas SBD mais a un comportement similaire vis-à-vis du DCS. D'une part les nœuds Patroni s'échangent des messages au travers du DCS. De plus, Patroni doit attendre l'expiration du verrou *leader* avant de pouvoir effectuer une bascule, ce qui est similaire au temps de réaction d'une architecture SBD. Mais surtout, l'instance PostgreSQL est déchue en cas de perte de communication avec le DCS, tout le serveur peut même être éteint si le *watchdog* est actif et que l'opération est trop longue.

### 1.5.6 Bilan des solutions anti-split-brain



À minima, une architecture fiable peut se composer au choix :

- *fencing* actif ou SBD
- 1 *watchdog* par serveur + quorum
- L'idéal : tous les configurer
- Désactiver les services au démarrage

Le *fencing* seul est suffisant pour mettre en œuvre un cluster fiable, même avec deux nœuds. Sans quorum, il est néanmoins nécessaire de désactiver le service au démarrage du cluster, afin d'éviter qu'un nœud isolé ne redémarre ses ressources locales sans l'aval du reste du cluster.

Notez que plusieurs algorithmes existent pour résoudre ce cas, hors quorum, (par exemple les paramètres `two_node`, `wait_for_all` et d'autres de Corosync). Néanmoins, dans le cadre de PostgreSQL, il n'est jamais très prudent de laisser une ancienne instance primaire au sein d'un cluster sans validation préliminaire de son état. Nous conseillons donc toujours de désactiver le service au démarrage, quelle que soit la configuration du cluster.

L'utilisation d'un SBD est une alternative intéressante et fiable pour la création d'un cluster à deux nœuds sans *fencing* actif. Le stockage y joue un peu le rôle du tiers au sein du cluster pour départager quel nœud conserve les ressources en cas de partition réseau. Le seul défaut de SBD par rapport au *fencing* est le temps d'attente supplémentaire avant de pouvoir considérer que le nœud distant est bien hors service. Aussi, attention au stockage iSCSI sur le même réseau que le cluster. En cas d'incident réseau généralisé, comme chaque machine perd son accès au disque ET aux autres machines via Pacemaker, toutes vont s'éteindre.

Une autre architecture possible est le cumul d'un quorum et du *watchdog*. Avec une telle configuration, en cas de partition réseau, la partition détenant le quorum attend alors la durée théorique du *watchdog* (plus une marge) avant de démarrer les ressources perdues. Théoriquement, les nœuds de la partition du cluster perdue sont alors soit redémarrés par leur *watchdog*, soit sains et ont pu arrêter les ressources normalement. Ce type d'architecture nécessite à minima trois nœuds dans le cluster, ou de mettre en place un nœud témoin, utilisé dans le cadre du quorum uniquement (eg. corosync QNetd).

Le cluster idéal cumule les avantages du *fencing*, du quorum et des *watchdogs*.

Comme nous l'avons vu, Pacemaker dispose de toutes les solutions connues. Reste à trouver la bonne combinaison en fonction des contraintes de l'architecture. Patroni, quant à lui, a une architecture similaire au SBD, mais ne force pas à utiliser le *watchdog* sur les nœuds. Pour avoir une architecture aussi fiable que possible, il est recommandé de toujours activer le *watchdog* sur tous les nœuds, au strict minima via *softdog*.

## 1.6 IMPLICATION ET RISQUES DE LA BASCULE AUTOMATIQUE



- Un collègue peu loquace de plus : un automate
- Complexification importante
- Administration importante
  - les opérations après bascule perdurent
- Formation importante
- Tests nécessaires
- Documentation et communication importante
- Sinon : erreurs humaines plus fréquentes
  - est-ce mieux pour la fiabilité ?

L'ajout d'un mécanisme de bascule automatique implique quelques contraintes qu'il est important de prendre en compte lors de la prise de décision.

En premier lieu, l'automate chargé d'effectuer la bascule automatique a tout pouvoir sur vos instances PostgreSQL. Toute opération concernant vos instances de près ou de loin **doit** passer par lui. Il est vital que toutes les équipes soient informées de sa présence afin que toute intervention pouvant impacter le service en tienne compte (mise à jour SAN, coupure, réseau, mise à jour applicative, etc).

Ensuite, il est essentiel de construire une architecture aussi simple que possible.



La complexification multiplie les chances de défaillance ou d'erreur humaine. Il est fréquent d'observer plus d'erreurs humaines sur un cluster complexe que sur une architecture sans bascule automatique !

Pour pallier ces erreurs humaines, la formation d'une équipe est vitale. La connaissance concernant le cluster doit être partagée par plusieurs personnes afin de toujours être en capacité d'agir en cas d'incident. Notez que même si la bascule automatique fonctionne convenablement, il est fréquent de devoir intervenir dessus dans un second temps afin de revenir à un état nominal (eg. reconstruire un nœud).

À ce propos, la documentation de l'ensemble des procédures est essentielle. En cas de maintenance planifiée ou d'incident, il faut être capable de réagir vite avec le moins d'improvisation possible. Quelle que soit la solution choisie, assurez-vous d'allouer suffisamment de temps au projet pour expérimenter, tester le cluster et le documenter.

### 1.6.1 Questions



N'hésitez pas, c'est le moment !

## 2/ 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)

## 2.1 PRÉAMBULE



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

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

### 2.1.1 Au menu



- Rappels théoriques
- RéPLICATION interne
  - réPLICATION physique
  - réPLICATION logique
- 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.

### 2.1.2 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.

## 2.2 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.

### 2.2.1 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* ».

## 2.2.2 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).

## 2.2.3 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.

## 2.2.4 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.

## 2.2.5 RéPLICATION SYNCHRONE SYMÉTRIQUE



### - Symétrique

- écritures sur les différents serveurs primaires
- besoin d'un gestionnaire de conflits
- lectures sur les différents serveurs

### - Synchrone

- les écritures sur les autres serveurs sont immédiates
- le client sait si sa commande est validée sur plusieurs serveurs
- risque important de lenteur !

Ce système est le plus intéressant... en théorie. L'utilisateur peut se connecter à n'importe quel serveur pour des lectures et des écritures. Il n'y a pas de risque de perte de données, vu que la commande ne réussit que si les données sont bien enregistrées sur tous les serveurs. Autrement dit, c'est le meilleur système de réPLICATION et de réPARTITION de charge.

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

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

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

## 2.2.6 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.

## 2.3 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.

### 2.3.1 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\_replay jusqu'en 9.6) qui crée à distance les journaux de transactions à partir d'un flux de réPLICATION.

### 2.3.2 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.

### 2.3.3 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*.

#### 2.3.4 *Hot Standby*

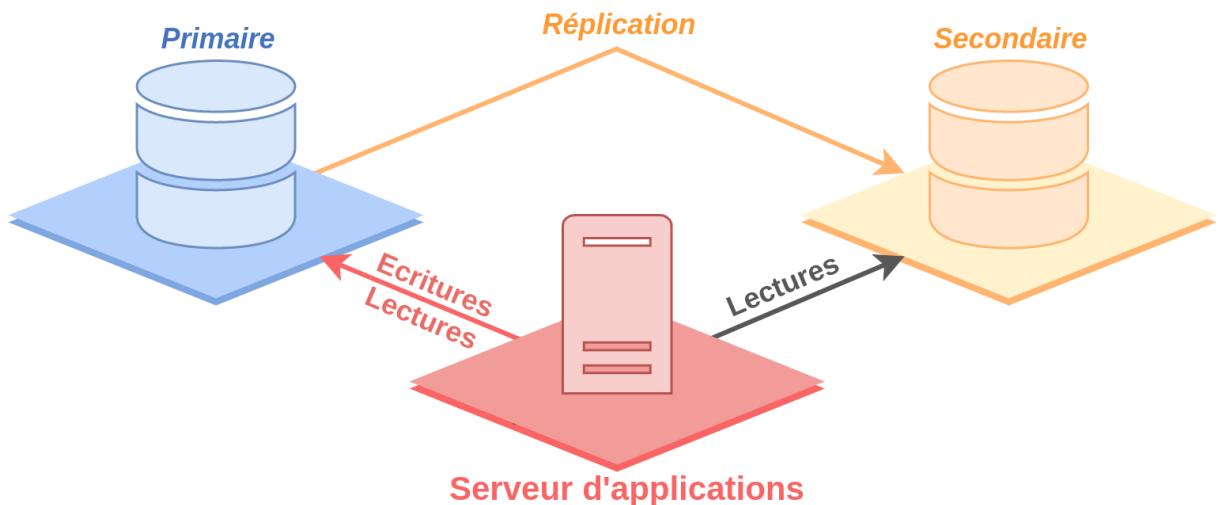


- Serveur secondaire
  - accepte les connexions entrantes
  - requêtes en lecture seule et sauvegardes
  - prêt à prendre le relai du primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée

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...

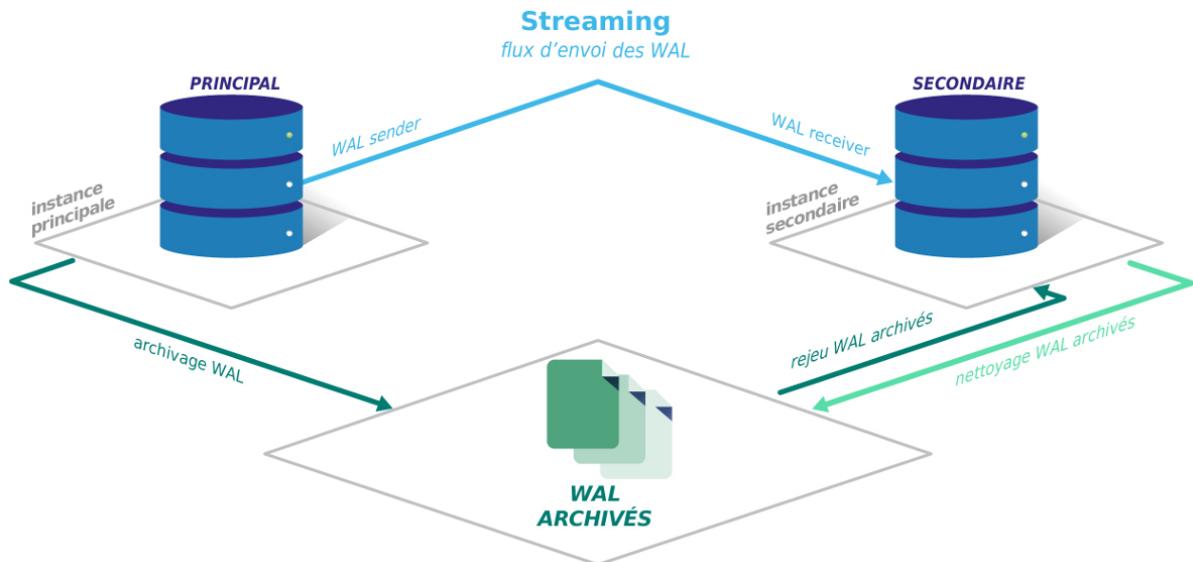
### 2.3.5 Exemple



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.

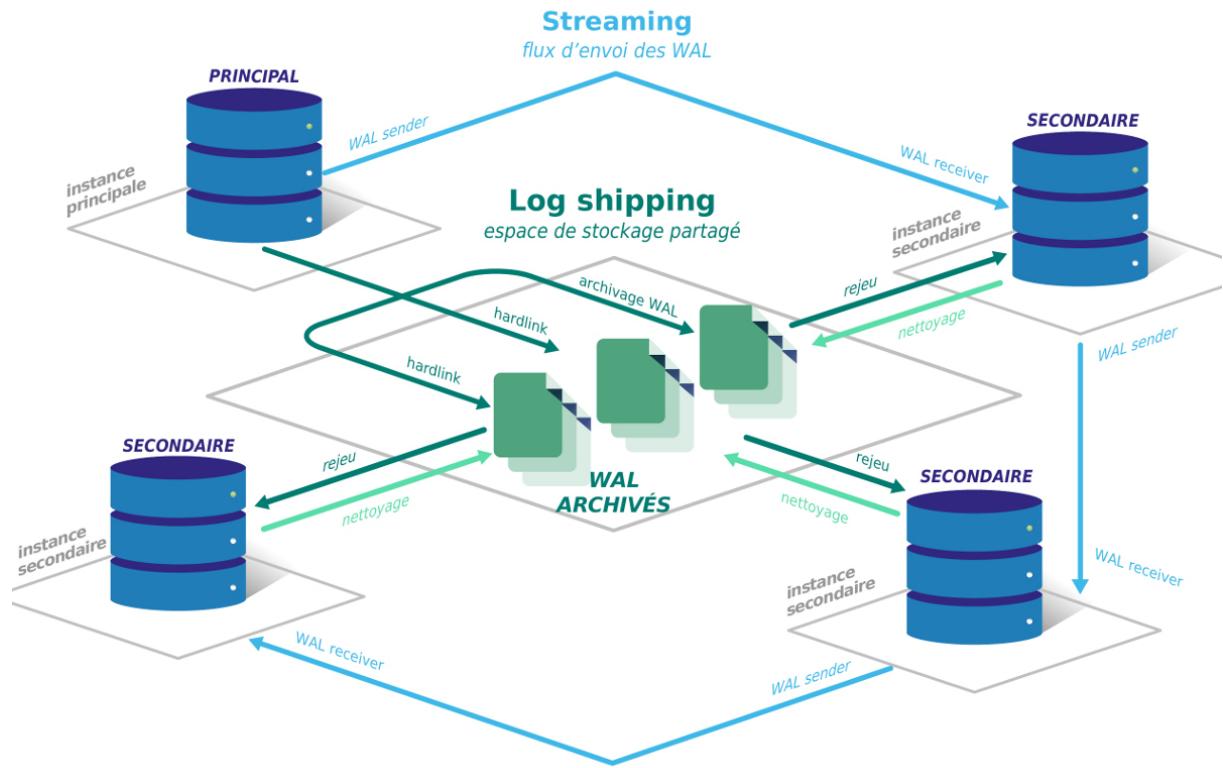
### 2.3.6 RéPLICATION interne

## RÉPLICATION INTERNE POSTGRESQL



### 2.3.7 RéPLICATION EN CASCADE

#### RÉPLICATION POSTGRESQL EN CASCADE



## 2.4 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.

### 2.4.1 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.

## 2.5 RÉPLICATION EXTERNE



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

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

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

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

Pour les détails sur ces outils et d'autres, voir le wiki<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/solutions.pdf](https://public.dalibo.com/exports/conferences/_archives/_2011/201102_haute_disponibilite_avec_postgresql/solutions.pdf)

## 2.6 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>

## 2.7 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.

### 2.7.1 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.

### 2.7.2 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...).

### 2.7.3 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.

## 2.8 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)

#### 2.8.1 Questions



N'hésitez pas, c'est le moment !

---

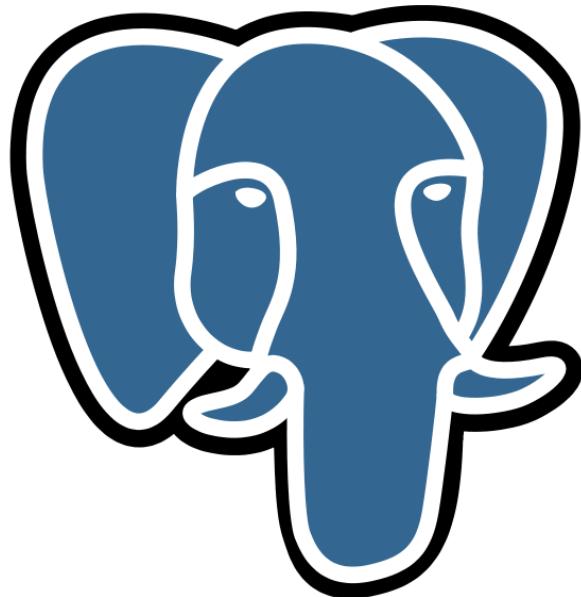
<sup>14</sup><https://docs.postgresql.fr/current/high-availability.html>

## 2.9 QUIZ



| [https://dali.bo/w1\\_quiz](https://dali.bo/w1_quiz)

### **3/ RéPLICATION PHYSIQUE : rappels**



## 3.1 INTRODUCTION

Patroni s'appuie sur la réPLICATION physique native de PostgreSQL pour répliquer les données entre les nœuds d'un agrégat.

Il est important de bien comprendre comment cette réPLICATION est mise en œuvre pour pouvoir régler les anomalies qui pourraient survenir lors de l'exploitation d'un agrégat Patroni.

### 3.1.1 Au menu



- Mise en place de la réPLICATION physique
- Promotion
- Retour à l'état stable

## 3.2 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.

### 3.2.1 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.

### 3.2.2 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).

### 3.2.3 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>.

### 3.2.4 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.

### 3.2.5 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`

### 3.2.6 Processus



Sur le primaire :

- `walsender replicator ... streaming 0/3BD48728`

Sur le secondaire :

- `walreceiver streaming 0/3BD48728`

### 3.3 PROMOTION



- Attention au *split-brain* !
- Vérification avant promotion
- Promotion : méthode et déroulement
- Retour à l'état stable

#### 3.3.1 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>

### 3.3.2 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.

### 3.3.3 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.

### 3.3.4 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
```

### 3.3.5 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.

### 3.3.6 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.

### 3.3.7 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.

### 3.4 CONCLUSION



- La réPLICATION physique native est très robuste
- Patroni va automatiser :
  - la mise en réPLICATION
  - la promotion
  - le raccrochage d'un ancien primaire

La robustesse de la réPLICATION physique a été éprouvée depuis longtemps, c'est à cette base solide que Patroni amène l'automatisation de :

- l'ajout de nœuds supplémentaires ;
- la promotion automatique en fonction du nœud ;
- le raccrochage d'un ancien primaire au nouveau primaire.

Ces mécanismes seront mis en œuvre dans la suite de ce document.

### 3.5 QUIZ



| [https://dali.bo/w2a\\_quiz](https://dali.bo/w2a_quiz)

## 3.6 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.

### 3.6.1 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 <b>postgres</b>	/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.

### 3.6.2 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.

### 3.6.3 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 <b>postgres</b>	/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
```

### 3.6.4 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();'
```

## 3.7 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...).

### 3.7.1 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.

### 3.7.2 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 ?

### 3.7.3 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.

## 3.8 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
```

### 3.8.1 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=''${lib_dir}/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: walsender 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.

### 3.8.2 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.

### 3.8.3 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|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|walreceiving)"
5844 \_ postgres: startup    recovering 000000030000000000000000C
5848 \_ postgres: walreceiver   streaming 0/C0001F0
5849 \_ postgres: walsender repli 127.0.0.1(48230) streaming 0/C0001F0
```

## **4/ Haute disponibilité de service**

## 4.1 PRÉAMBULE



- Quelle architecture pour l'instance primaire et ses secondaires ?
- Plusieurs architectures et solutions possibles
  - *Shared Storage* : Pacemaker
  - *Shared Nothing*
    - Pacemaker
    - Patroni

## 4.2 SHARED STORAGE



- Architecture simple
  - *Cold standby* matériel
  - Disque partagé entre primaire et secours
    - accessible par **un seul** serveur
  - Redondance du stockage nécessaire
    - SAN, DRBD...
  - Alternative : créer/déplacer une machine virtuelle
  - Pacemaker : possible

Une architecture de type *Shared Storage* repose essentiellement sur un disque partagé entre au moins deux serveurs. Les données de l'instance sont situées sur le disque partagé. En cas de panne sur le serveur actif, les disques sont montés sur l'un des serveurs de secours et l'instance PostgreSQL y est démarrée.

L'avantage d'une telle architecture est son apparence simplicité. Les données étant habituellement situées dans un SAN, il est aisément en cas d'incident sur le serveur principal de monter les disques dans un serveur de secours et redémarrer PostgreSQL. La configuration de PostgreSQL reste au plus simple et l'espace occupé par les données de l'instance n'est pas dupliqué à travers plusieurs serveurs.

La haute disponibilité des données doit être prise en charge par une réPLICATION au niveau disque (SAN, DRBD, etc.).

La disponibilité du service dépend de la technologie utilisée : machine virtuelle, lame, serveur physique. Certaines technologies de virtualisation sont capables de « migrer » une machine virtuelle et ses disques en cas de panne sur un hyperviseur. D'autres cluster-wares (comme Pacemaker, rgmanager) sont capables de basculer le disque et le service sur un serveur tiers du cluster.



L'un des points essentiels de cette architecture est de s'assurer que le disque ne peut être disponible que sur un seul serveur à la fois à tout instant de façon excessivement stricte. La fiabilité des données en dépend.

## 4.3 SHARE NOTHING



- Plusieurs nœuds indépendants
  - *ie* plusieurs instances PostgreSQL
  - en réPLICATION
- Notamment : Patroni

L'autre architecture de clustering est appelée *Share Nothing*. Dans une telle architecture, chaque nœud du cluster est totalement indépendant.

Dans le cadre de PostgreSQL, cela signifie que les données sont situées sur plusieurs serveurs grâce à la réPLICATION interne de ce dernier. Les solutions que nous proposons par la suite sont toutes de type *Share Nothing*.

## 4.4 PATRONI



- Fiable : SDB + *watchdog*
- Ne gère que PostgreSQL
  - bascule
  - (re)construction
- 2 nœuds PostgreSQL ou plus par cluster Patroni
- Simple mais formation nécessaire
- 1 cluster DCS de 3 nœuds est recommandé
  - etcd (ou autre)
  - un DCS peut gérer N clusters Patroni
  - +watchdog

Patroni assure l'exploitation d'un cluster PostgreSQL en réPLICATION et est capable d'effectuer une bascule automatique en cas d'incident sur l'instance primaire.

Le projet repose sur un DCS externe comme stockage distribué de sa configuration. etcd<sup>1</sup>, basé sur le protocole Raft, est sans doute le plus simple, mais d'autres sont possibles.

Le DCS nécessite au moins 3 nœuds (donc 3 serveurs ou machines virtuelles) pour assurer son quorum propre, la sécurité et la viabilité de ses données. Néanmoins un même cluster DCS peut ensuite gérer plusieurs clusters Patroni.

Se reposer sur un DCS fiable permet à Patroni plus de robustesse et de se focaliser sur l'expertise PostgreSQL uniquement. Le mécanisme de modification atomique de la base de configuration distribuée permet notamment une élection fiable en cas d'incident.

Depuis la version 2.0, Patroni permet aussi d'utiliser le protocole Raft, sans utiliser de DCS externe. Nous n'avons aucun recul sur cette technologie pour le moment. Nous utiliserons Patroni avec etcd par la suite.

En plus de ce DCS fiable et d'un mécanisme d'élection ne laissant aucune place aux *race conditions*, Patroni supporte l'utilisation d'un *watchdog* matériel. Le couple DCS+*watchdog* permet donc d'éviter les situations de *split-brain*.

Depuis la version 2.0, Patroni offre la possibilité d'exécuter une action pré-promotion, capable d'annuler la promotion si nécessaire. Cette fonctionnalité a été ajoutée afin de permettre l'ajout d'une action de *fencing* de l'ancien primaire. Nous n'avons pour le moment aucun recul sur cette fonctionnalité. Il faut notamment étudier quelles informations sont accessibles depuis le callback pour décider d'un *fencing* ou non.

<sup>1</sup><https://etcd.io/>

Patroni est un projet simple, rapide à déployer et prendre en main. Il est toutefois toujours hautement recommandé de former une équipe à son administration.

## 4.5 PACEMAKER



- Référence de la HA sous Linux
- Principaux contributeurs : RedHat et Suse
- Empaqueté et cohérent sur toutes les distributions principales
- Super fiable : quorum, *watchdog* et *fencing* supportés
- Gère PostgreSQL et tout autre ressource
- Cluster deux nœuds possible avec *fencing*

Pacemaker est la solution de haute disponibilité de référence sur les distributions Linux modernes. Plusieurs entreprises telle que RedHat, Suse ou Linbit investissent du temps de recherche et développement pour la maintenance et l'évolution du projet. RedHat et Suse supportent officiellement les clusters Pacemaker (souvent au travers de souscriptions complémentaires) en imposant certaines contraintes de robustesse à leurs clients (comme un *fencing* obligatoire).

Le projet est très complet, supporte plusieurs types de *fencing*, avec escalade possible, le quorum, l'utilisation de *watchdog*, le clustering étendu (déployé entre deux datacenters). Il est capable de mettre en haute disponibilité plusieurs dizaines de services tels que des bases de données, des serveurs HTTP, mail, des machines virtuelles, des containers, etc.

Cette souplesse a néanmoins un prix en matière d'apprentissage et de maintenance. Il est largement recommandé d'être formé à l'utilisation et la maintenance de Pacemaker avant de le mettre en production.

Concernant la gestion de PostgreSQL, nous recommandons l'utilisation de l'agent PAF<sup>2</sup> (*PostgreSQL Automatic Failover*). Son développement a été lancé au sein de Dalibo, et il est maintenu actuellement par Jehan-Guillaume de Rorthais, afin de corriger ou contourner les problèmes et limitations de l'agent officiel existant. Il est entré au sein de l'organisme ClusterLabs regroupant les différents projets liés à Pacemaker.

---

<sup>2</sup><https://clusterlabs.github.io/PAF/>

## 4.6 ACCÈS AUX RESSOURCES



Comment l'application accède-t-elle à la bonne instance ?

- IP virtuelle
- Proxy
  - HAProxy & autres
- Intelligence dans l'application
  - ex: `target_session_attrs=read-write`

Il existe plusieurs solutions pour gérer l'accès des applications à l'instance principale.

La plus simple est l'utilisation d'une IP virtuelle. Cette solution est à privilégier avec Pacemaker, ce dernier gérant alors à la fois la localisation de l'IP virtuelle et de l'instance primaire. Cette solution est aussi envisageable avec Patroni, à condition d'utiliser un script en callback ou d'ajouter un service supplémentaire dédié à cette tâche. Cette solution n'est cependant pas totalement robuste avec Patroni.

Une autre solution revient à utiliser un tiers faisant office de proxy entre les clients et l'instance principale. HAProxy est une solution populaire, mais il en existe bien d'autres, propriétaires ou non. La solution doit être capable de déterminer où se trouve l'instance primaire au sein du cluster.

Enfin, la dernière solution consiste à intégrer l'intelligence nécessaire directement dans les couches applicatives. La chaîne de connexion à l'instance peut par exemple comporter plusieurs destinataires et désigner le rôle recherché (`target_session_attrs=read-write`). Il est aussi possible d'utiliser un gestionnaire de configuration centralisé aux applications tel que confd.

## 4.7 COMPARATIF DES SOLUTIONS ET CHOIX

Solution	RTO	RPO	Qui	Complexité	Serveurs
PITR	humain	> 1 min	DBA	1	1+
RéPLICATION	humain	< 1 min	DBA	2	2+
Shared disk	< 1mn	0-1 min	SYS	5	2+
Patroni	> 5s	0-1 min	DBA	4	5+
PAF	> 5s	0-1 min	DBA/SYS	5	3 (2+)

- Humain = tâche effectué / déclenché manuellement
- Nombre de Serveurs dédiés aux sauvegardes et HAProxy nom comptés

### 4.7.1 Le jeu en vaut-il la chandelle ?



- Complexité et coût
- Souvent plus sûr de rester sur des procédures :
  - simples
  - manuelles
  - éprouvées
  - maîtrisées
- Patroni ou Pacemaker/PAF
  - reconnus
  - éprouvés
  - éviter le reste

La mise en œuvre de la haute disponibilité apporte un certain nombre de complications et complexifications à l'architecture. Avec une disponibilité exigée de 99,5% (soit 44 h d'indisponibilité par an !), nous déconseillons la mise en œuvre d'une telle architecture. Il est plus sage de reposer sur des procédures connues et éprouvées, outillées, avec une phase de prise de décision humaine optimisée pour être déclenchée le plus vite possible.

Une meilleure maîtrise de l'architecture évite souvent les appels d'astreintes impromptus et se substitue avantageusement à une bascule automatique beaucoup plus difficile à maîtriser et budgétiser.

Si une bascule automatisée est nécessaire, nous recommandons principalement l'utilisation de Patroni ou de Pacemaker/PAF, d'une part pour leur robustesse, d'autre part pour leur adoption importante au sein de la communauté.



Évitez les solutions exotiques, les systèmes de fichiers distribués ou les projets peu populaires ou peu maintenus qui vous exposent à des risques certains d'indisponibilité, voire de corruption (expériences plusieurs fois vécues au support Dalibo).

## 4.8 CONCLUSION



### Points de vigilance

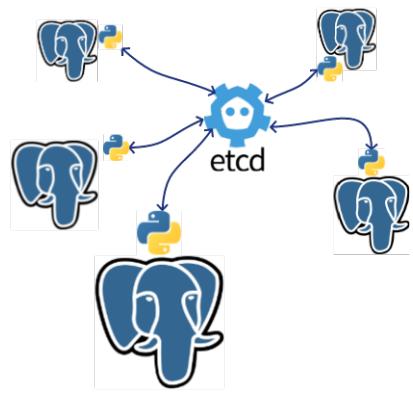
- La supervision
- Les sauvegardes
- La formation des équipes
- La documentation

Si vos instances sont critiques et nécessitent une haute disponibilité de service et/ou de données, rappelons qu'il est **vital** de porter un soin particulier à la **supervision** pour anticiper les incidents et aux **sauvegardes** pour pallier aux incidents majeurs.

Enfin, la **formation des équipes** et la **documentation** à jour du montage utilisé sont indispensables.



## 5/ Patroni : Architecture et fonctionnement



## 5.1 AU MENU



- Architecture générale
- L'algorithme Raft
- ETCD
- Patroni
- Proxy, VIP et poolers de connexions

Sur les bases de la réPLICATION physique de PostgreSQL, Patroni fournit un modèle de gestion d'instances, avec bascule automatique et configuration distribuée.

Les tâches nécessaires pour la bascule ou la promotion, l'ajout d'un nouveau nœud, la resynchronisation suite à une défaillance, deviennent la responsabilité de Patroni qui veillera à ce que ces actions soient effectuées de manière fiable, en évitant toujours la multiplicité de nœuds primaires (*split-brain*).

Dans ce chapitre, nous allons donc décrire l'architecture et la mise en œuvre d'une telle solution de haute disponibilité de service.

## 5.2 ARCHITECTURE GÉNÉRALE



- Haute disponibilité de service
- Au moins deux agrégats de serveurs :
  - DCS (etcd)
  - Patroni
- Instances gérées **uniquement** par Patroni
- Synchronisation des horloges
  - attention aux *snapshots* !

Patroni est destiné à la haute disponibilité d'un service PostgreSQL.

Il s'appuie sur un gestionnaire de configuration distribuée (DCS) pour partager l'état des nœuds de son agrégat et leur configuration. Nous utiliserons etcd pour cela.

Notre but étant la haute disponibilité, l'agrégat etcd ne doit pas devenir un SPOF (*single point of failure*) et doit donc être tolérant aux pannes. Le travail de bascule, d'élection du noeud *leader* ou de retour après défaillance doivent s'effectuer automatiquement au sein de l'agrégat etcd.

L'application des modifications de la configuration de PostgreSQL est effectuée par Patroni qui se charge de la répercuter sur tous les nœuds.



Le démarrage et l'arrêt du service PostgreSQL sur chaque nœud ne doivent plus être générés par le système et doivent être désactivés. Toutes les actions de maintenances (arrêt, démarrage, rechargement de configuration, promotion) doivent être faites en utilisant Patroni plutôt que les moyens traditionnels (pg\_ctl, systemctl, etc).

Nous sommes donc en présence de deux agrégats de serveurs différents :

- l'un pour les nœuds etcd ;
- l'autre pour les nœuds Patroni gérant chacun une instance PostgreSQL.

### Horloges système

La synchronisation des horloges de tous les serveurs est primordiale, elle doit être déléguée à un autre serveur ou agrégat de serveurs NTP<sup>1</sup>.

<sup>1</sup>Network Time Protocol

Un problème pouvant survenir concerne certains ralentissements dus aux sauvegardes par *snapshot* des machines virtuelles. Une indisponibilité (*freeze*) trop grande peut entraîner une bascule automatique et une désynchronisation du nœud fautif. Celui-ci est alors dans l'impossibilité de se raccrocher au nouveau primaire lorsqu'il redevient fonctionnel et sa reconstruction est inévitable.

L'anomalie se signale par exemple dans les traces d'etcd par le message :

```
etcd: the clock difference... is too high
```

## 5.3 L'ALGORITHME RAFT



- Algorithme de consensus
- *Replicated And Fault Tolerant*
- *Leader-based requests*
  - seul à échanger avec Patroni (client)

Raft<sup>2</sup> est un algorithme de consensus répliqué capable de tolérance de panne (*Replicated And Fault Tolerant*). Ce genre d'algorithme vise à permettre à un ensemble de serveurs de fonctionner comme un groupe cohérent qui peut survivre à la disparition de certains de ses membres.

L'élaboration de Raft part du constat que les algorithmes de consensus comme Paxos<sup>3</sup> sont complexes à comprendre et donc à implémenter. C'est un problème important, car ils sont utilisés dans des applications critiques qui nécessitent donc d'être bien comprises et maîtrisées. L'un des objectifs principaux lors de l'élaboration de Raft était donc de le rendre, dans la mesure du possible, simple à comprendre et à implémenter en plus d'être : prouvé, complet et fonctionnel.

Avec Raft, les clients dialoguent uniquement avec le nœud *leader*. Ici, le client est Patroni, et ses requêtes correspondent à :

- un changement d'état du nœud ;
- une modification de configuration d'une instance.

Si une requête est envoyée à un nœud secondaire (*follower*), elle va échouer en renvoyant des informations sur l'adresse du *leader*.

---

<sup>2</sup><https://raft.github.io/raft.pdf>

<sup>3</sup>[https://fr.wikipedia.org/wiki/Paxos\\_\(informatique\)](https://fr.wikipedia.org/wiki/Paxos_(informatique))

### 5.3.1 Raft : Journal et machine à états



- Machine à états
  - *leader/follower/candidate*
- Journal des événements
  - répliqué depuis le *leader*
  - même résultat sur tous les nœuds
- Mandats
  - commence par une élection du *leader* (unique)
  - numérotés, croissants

Raft implémente une machine à états déterministe qui utilise un journal répliqué.

La machine à états joue les modifications les unes après les autres, dans leur ordre d'arrivée. Cela permet de garantir que la machine va d'état stable à état stable et produit le même résultat sur tous les nœuds.

L'algorithme de consensus se charge de maintenir la cohérence du journal répliqué.

Dans Raft les nœuds peuvent avoir trois états :

- *follower* (suiveur) : c'est l'état par défaut lorsqu'on démarre un agrégat. Dans cet état, le nœud est passif. Il s'attend à recevoir régulièrement un message du *leader* (*heart beat via Append Entries RPC*) et potentiellement de nœuds candidats (*Request Vote RPC*). Il se contente de répondre à ces messages.
- *candidate* (candidat) : c'est l'état que prend un nœud *follower* s'il ne reçoit pas de message du *leader* pendant un certain temps. Dans cet état, il va envoyer des demandes de votes (*Request Vote RPC*) aux autres nœuds pour se faire élire comme nouveau *leader*. S'il perd l'élection, il redevient *follower*.
- *leader* : c'est l'état que prend un candidat après avoir remporté une élection. Dans cet état, le nœud va envoyer des messages d'ajout dans les journaux (*Append Entries RPC*) à intervalle régulier. Ces messages servent à la fois pour répliquer les commandes envoyées au *leader* par les clients et pour signifier aux *followers* que le *leader* est toujours vivant (*heart beat*). Si le *leader* découvre un *leader* avec un mandat supérieur au sien, il redevient *follower*.

Les mandats (*terms*) servent à déterminer quand des informations sont obsolètes dans l'agrégat.

Les règles suivantes concernent les mandats :

- chaque mandat commence par une élection ;

- les mandats sont numérotés avec des entiers consécutifs ;
- chaque nœud ne peut voter qu'une fois par mandat ;
- il ne peut y avoir qu'un seul *leader* par mandat. Il est possible qu'un mandat n'ait pas de *leader* si une élection a échoué (*split vote*) ;
- chaque serveur garde une trace du mandat dans lequel il évolue ; l'identifiant de mandat est échangé dans tous les messages.
- si un nœud reçoit un message avec un identifiant de mandat supérieur, il met à jour son identifiant de mandat et redevient *follower* (si ce n'est pas déjà le cas) ;
- si un nœud reçoit un message avec un identifiant de mandat inférieur au sien, il ignore la demande et renvoie une erreur à son expéditeur.

Raft sépare les éléments essentiels pour le consensus en trois parties :

- élection d'un *leader* : il ne doit toujours y avoir qu'un seul *leader*, si un *leader* disparaît un nouveau doit le remplacer ;
- réPLICATION du journal : le *leader* reçoit les modifications depuis les clients et les répliques vers les autres serveurs ; le journal du *leader* fait toujours référence ;
- sécurité & cohérence : si un serveur a appliqué une entrée du journal dans sa machine à état, les autres serveurs doivent appliquer la même entrée provenant de la même position dans le journal.

### 5.3.2 Élection d'un leader



- *Heart beats* depuis le *leader* vers les *followers*
- Si pas de nouvelles du *leader* :
  - démarrage de l'élection d'un nouveau *leader*
  - Promotion par consensus des *followers*
  - Si échec de l'élection, attente aléatoire
  - Tolérance de panne si 3 nœuds

Le *leader* informe régulièrement tous les *followers* de sa présence (*heart beat*) par l'envoi de message d'ajout au journal (*Append Entries RPC*) qui peuvent être vide s'il n'y a pas d'activité.

Si le message tarde à arriver sur un nœud, ce dernier assume que le *leader* a disparu et devient candidat. Il incrémentera alors son numéro de mandat, vote pour lui-même et effectue une demande d'élection (*Request Vote RPC*).

Si le candidat obtient la majorité des votes, il remporte l'élection et devient *leader*. Il envoie alors un message d'*heart beat* aux autres serveurs afin de leur signifier la présence d'un nouveau *leader*.

Si le candidat reçoit un message de *heart beat* en provenance d'un autre nœud alors qu'il est en train d'attendre des votes, il vérifie le numéro de mandat. Si le mandat de ce serveur est supérieur ou égal au

mandat du candidat, cela signifie qu'un autre nœud a été élu *leader*. Dans ce cas, le candidat devient *follower*. Sinon, il renvoie un message d'erreur et continue son élection.

Le dernier cas de figure est qu'aucun candidat ne parvienne à remporter l'élection pendant ce mandat. Dans ce cas, les candidats vont atteindre leur *time out* d'élection et en démarrer une nouvelle. Ce cas de figure est désigné sous le nom de *split vote*.

Le temps d'attente avant le *time out* est différent sur chaque nœud, c'est une valeur aléatoire choisie dans une plage pré définie (par exemple 150 ms-300 ms) désignée sous le nom d'*election time out*. Cette randomisation limite l'occurrence des demandes de vote simultanées et la multiplication de *split votes*.



Pour que la tolérance de panne soit assurée, il faut préserver un nombre de nœuds fonctionnels suffisant pour obtenir une majorité : au minimum **trois**.

### 5.3.3 RéPLICATION DU JOURNAL



- Mécanisme de réPLICATION
- Structure des journaux
- Protection contre les incohérences
  - n° de mandat + index

L'application cliente envoie sa commande au *leader*. Cette commande est ajoutée dans son journal avant d'être envoyée aux *followers* dans un message d'ajout (*Append Entries RPC*).

Si une commande est répliquée dans les journaux de la majorité des nœuds de l'agrégat, elle est exécutée par le *leader* (commité) qui répond ensuite au client. Le *leader* conserve une trace de l'index du dernier enregistrement de log appliqué dans la machine à états (*commit index*), dont la valeur est incluse dans les messages envoyés aux *followers* (*Append Entries RPC*). Cette information permet aux *followers* de savoir quels enregistrements de leur journal peuvent être exécutés par leur machine à états.

Si un *follower* est indisponible, le *leader* tente de renvoyer le message jusqu'à ce qu'il soit reçu.

Raft s'appuie sur les prédictats suivants pour garantir la consistance du journal (*log matching properties*). Si deux entrées de différents journaux ont les mêmes numéros d'index (position dans le journal) et de mandat :

- elles stockent la même commande ;
- les entrées précédentes sont identiques.

Le premier prédictat est garanti par le fait que le *leader* ne produit qu'une seule entrée avec un numéro d'index donné pendant un mandat. De plus les entrées créées dans le journal ne changent pas de position dans le journal.

Le second prédictat est garanti par la présence d'informations supplémentaires dans les demandes d'ajout au journal (*Append Entries RPC*) : les numéros d'index et de mandat de la commande précédente. Le *follower* n'écrit un enregistrement dans son journal que si l'index et le mandat de l'enregistrement précédent correspondent à ce qu'a envoyé le *leader*. Sinon la demande du *leader* est rejetée.

En temps normal, ce contrôle de cohérence ne tombe jamais en échec. Cependant, suite à une élection, plusieurs cas de figure sont possibles :

- le *follower* est à jour, il n'y a rien à faire ;
- le *follower* est en retard, il faut lui communiquer les entrées manquantes ;
- le journal du *follower* contient des entrées qui ne sont pas sur le *leader* : cela peut arriver lorsqu'un ancien *leader* redevient *follower* alors qu'il n'avait pas encore reçu de consensus sur des valeurs qu'il avait écrites dans son journal.

Pour parer à ces situations, le *leader* maintient un index (*next index*) pour chaque *follower* qui correspond à la position du prochain enregistrement qui sera envoyé à ce *follower*. Suite à l'élection, cet index pointe vers la position de l'enregistrement qui suit le dernier enregistrement du journal. De cette manière si le journal d'un *follower* n'est pas cohérent avec celui du *leader*, le prochain message d'ajout dans les journaux sera en échec. Le *leader* va alors décrémenter son compteur d'une position et renvoyer un nouveau message. Ce processus va se répéter jusqu'à ce que le *leader* et le *follower* trouvent un point de consistance entre leurs journaux. Une fois cette position trouvée, le *follower* tronque la suite du journal et applique toutes modifications en provenance du *leader*.

#### 5.3.4 Sécurité & cohérence



- Les nœuds votants refusent de donner leurs votes à un candidat qui est en retard par rapport à eux.

Quelques protections supplémentaires sont nécessaires pour garantir la cohérence du journal distribué en cas d'incident majeur.

La plus importante est mise en place lors du processus d'élection : les nœuds votants refusent de voter pour un candidat qui est en retard par rapport à eux. Cela est rendu possible grâce à la présence du numéro de mandat et d'index du dernier enregistrement du journal du candidat dans le message de demande de vote (*Request Vote RPC*). Le numéro de mandat du candidat doit être supérieur ou égal à celui du votant. Si les numéros de mandat sont égaux, les numéros d'index sont comparés avec le même critère.

### 5.3.5 Majorité et tolérance de panne



- Nombre total de nœuds impair
- Majorité : (nombre de nœuds / 2) + 1
- Tolérance de panne : nombre de nœuds - quorum

Pour éviter le statu quo ou l'égalité entre deux nœuds, il faut pouvoir les départager par un déséquilibre du vote et ceci dans un laps de temps donné.

La promotion d'un nœud candidat ne peut avoir lieu que si la majorité des autres nœuds a voté pour lui.

La tolérance de panne définit combien de nœuds l'agrégat peut perdre, tout en restant opérationnel, c'est-à-dire de toujours pouvoir procéder à une élection.



Un agrégat ayant perdu son quorum ne doit pas répondre aux sollicitations des clients.

Bien que l'ajout d'un nœud à un cluster de taille impaire semble une bonne idée au premier abord, la tolérance aux pannes est pourtant **pire** : le même nombre de nœuds peut défaillir sans que l'on perde le quorum. (Comme il y a plus de nœuds, la probabilité de panne est logiquement plus grande.)

### 5.3.6 Tolérance de panne : Tableau récapitulatif

Nombre de nœuds	Majorité	Tolérance de panne
2	2	0
⇒ 3	2	<b>1</b>
4	3	1
⇒ 5	3	<b>2</b>
6	4	2
⇒ 7	4	<b>3</b>
8	5	3

### 5.3.7 Interaction avec les clients



- Toutes les communications passent par le *leader*
- Protection contre l'exécution en double des requêtes
- Le *leader* valide les entrées non commitées après son élection
- Le *leader* vérifie qu'il est bien *leader* avant de répondre à une demande de lecture

Dans Raft, les clients communiquent uniquement avec le *leader*. Si un autre nœud reçoit un message provenant d'un client, il rejette la demande et renvoie la position du *leader*.

En l'état, Raft peut rejouer deux fois les mêmes commandes : par exemple, si un *leader* plante après avoir validé un enregistrement mais avant d'avoir confirmé son exécution au client. Dans ce cas, le client risque de renvoyer la commande, qui sera exécutée une seconde fois.

Afin d'éviter cela, le client doit associer à chaque commande un numéro unique. La machine à état garde trace du dernier numéro traité, si elle reçoit une commande avec un numéro qui a déjà été traité, elle répond directement sans ré-exécuter la requête.

Immédiatement après l'élection d'un nouveau *leader*, ce dernier ne sait pas quelles entrées sont exécutées dans son journal. Pour résoudre ce problème immédiatement après son élection, le leader envoie un message vide d'ajout dans le journal (*Append Entries RPC*) avec un index (*next\_index*) qui pointe sur l'entrée qui suit la dernière entrée de son journal. De cette manière, tous les *followers* vont se mettre à jour et les entrées du journal du *leader* pourront être exécutées.

Une dernière précaution est que le *leader* envoie toujours un *heart beat* avant de traiter une demande de lecture. De cette manière, on s'assure qu'un autre *leader* n'a pas été élu entre temps.

### 5.3.8 Raft en action



- Démo interactive : Raft en animation<sup>4</sup>

Cette animation permet de simuler le fonctionnement d'un cluster Raft en ajoutant ou enlevant à volonté des nœuds.

Une autre animation plus guidée existe aussi chez \_The Secret Lives of Data<sup>5</sup>.

---

<sup>4</sup><http://thesecretlivesofdata.com/raft/>

## 5.4 ETCD



- Serveur de configurations distribuées
  - clé/valeur
  - Multiples nœuds
  - Haute disponibilité et tolérance de panne
  - Élection par un quorum de 3 nœuds minimum

Dans ce chapitre, nous présenterons etcd, un des gestionnaires de configurations distribuées utilisés par Patroni. Il a été créé à l'origine par CoreOS, puis développé par une communauté de développeurs, sous licence libre (Apache License 2.0).

Les configurations distribuées sont fondamentalement juste des ensembles clé/valeur.

### 5.4.1 etcd et Raft



- Hautement disponible
- Implémente l'algorithme Raft
- Perte du quorum

etcd est un serveur de configuration distribuée, écrit en Go avec pour ligne directrice la haute disponibilité. L'élection du *leader* en son sein ainsi que la modification des configurations distribuées qu'il héberge, sont gérées à l'aide de l'algorithme Raft. On retrouve donc les mêmes caractéristiques déjà exposées précédemment et la cohérence des modifications appliquées via un journal.

La perte du quorum est atteinte si la tolérance de panne est dépassée.

Si vous perdez le quorum etcd, vous devez sauvegarder la base et la configuration de chaque nœud, supprimer votre agrégat et en créer un nouveau. Vous pouvez utiliser un nœud etcd sain pour former un nouvel agrégat, mais vous devez supprimer de toute façon tous les autres nœuds, même sains.



Si un agrégat est dans un état où il ne peut plus tolérer de panne, l'ajout d'un nœud *avant de supprimer des nœuds défaillants est dangereux* : si le nouveau nœud ne parvient pas à s'enregistrer auprès de l'agrégat (suite à une erreur de configuration par exemple), le quorum sera alors définitivement perdu.

### 5.4.2 Serveur de configurations distribuées



Patroni utilise etcd pour :

- l'attribution du jeton *leader* Patroni
- la configuration des instances distribuées

etcd est utilisé par Patroni pour distribuer le jeton propre à l'instance primaire de l'agrégat PostgreSQL, de la même manière que pour le *leader* de l'agrégat etcd, mais ces deux jetons sont différents.

En outre, la configuration commune à tous les nœuds PostgreSQL est stockée dans etcd après leur création (*bootstrap*). La modification des paramètres communs est ainsi simplifiée puisqu'elle se résume à envoyer la modification à l'agrégat etcd, Patroni se charge de sa prise en compte sur toutes les instances.

### 5.4.3 Multiples nœuds



- Multiples nœuds synchrones (journaux)
- De 3 à 7 nœuds
- Peut gérer plusieurs agrégats Patroni distincts

etcd implémente une réPLICATION synchrone des journaux d'événements entre les nœuds. L'application des événements dans la machine à états de chaque nœud se fait de manière asynchrone.

La tolérance de panne est assurée par l'agrégation de serveurs ou nœuds, dont le nombre doit être supérieur ou égal à 3.

Bien qu'aucune limite stricte n'existe, le maximum de 7 nœuds est conseillé.

Un même cluster etcd peut gérer plusieurs agrégats Patroni distincts, voire d'autres services.

#### 5.4.4 Élection



- Chaque *follower* est éligible
- Quorum minimum pour une élection
- Déclenchée après perte du *leader*

Pour qu'une élection ait lieu de manière fiable, 3 votes doivent être acquis par le prétendant, le sien compris. Il faut donc un minimum de 3 nœuds disponibles pour qu'une élection soit possible.

Une élection est lancée lorsque le temps d'attente (*time out*) du nœud a été atteint et qu'aucun *leader* n'a pu lui confirmer sa bonne santé.

Chaque nœud choisit son *time out* au hasard pour ne pas effectuer sa demande d'élection en même temps qu'un autre.

#### 5.4.5 Conséquences



- Élection automatique possible que si 3 nœuds
- Réponse avec 2 nœuds : uniquement si *leader* identifié
- Multi-site ⇒ au moins 3 sites + nœuds en nombre **impair**

L'élection automatique d'un nœud n'est possible que si 2 **autres** nœuds sont disponibles pour dépasser un statu quo qui se produit lorsque seulement deux nœuds sont présents.

Un agrégat de 2 nœuds ne peut répondre aux demandes de modifications de la configuration distribuée que si un *leader* est identifié.

Dans une architecture multi-site, il est donc nécessaire de disposer d'au moins 3 sites, avec chacun un ou plusieurs nœuds permettant d'obtenir un quorum.



Ajouter un nœud à un nombre *impair* de nœuds n'améliore pas la tolérance de panne et ne fait qu'augmenter la complexité. **Pour augmenter la tolérance de panne, l'ajout de nœud doit porter le nombre total de nœuds à un nombre impair.**

#### 5.4.6 Principaux paramètres



- Paramètres pour les nœuds entre eux
- Paramètres pour les clients
- Initialisation (*bootstrap*)

Paramètre	Signification
ETCD_NAME	Nom du nœud
ETCD_LISTEN_PEER_URLS	Adresse, méthode et port pour la communication inter-nœuds
ETCD_LISTEN_CLIENT_URLS	Adresse, méthode et port d'écoute pour les clients du nœud
ETCD_INITIAL_ADVERTISE_PEER_URLS	Adresse, méthode et port d'écoute interne initiaux du nœud
ETCD_ADVERTISE_CLIENT_URLS	Adresse, méthode et port d'écoute externe à communiquer aux autres nœuds
ETCD_INITIAL_CLUSTER	Liste de nœuds de l'agrégat initial
ETCD_INITIAL_CLUSTER_TOKEN	Premier jeton d'identification de l'agrégat
ETCD_INITIAL_CLUSTER_STATE	État initial de l'agrégat
ETCD_ENABLE_V2=true	Activation de l'API v2

#### 5.4.7 Questions



- C'est le moment !

## 5.5 INSTALLATION D'UN CLUSTER ETCD



- Packaging Debian ou Red Hat 7/CentOS 7
  - ...pas 8 !
  - 1 machine par nœud
  - Un peu fastidieux mais simple
  - Ports :
    - 2380 entre nœuds etcd
    - 2379 avec applications clientes
    - penser au firewall
  - API: v2 et v3 différentes !

L'installation est détaillée ci-dessous pour Red Hat/CentOS 7 et Rocky Linux 8, et Debian/Ubuntu. Elle décrit le déploiement d'un *cluster* etcd composé de 3 nœuds.

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

### 5.5.1 Sur Red Hat 7/CentOS 7

Sur chacun des nœuds, installer le paquet etcd :

```
# yum install etcd
```

Puis ouvrir les ports comme décrits ci-dessous.

### 5.5.2 Sur Rocky Linux 8

etcd n'est pas packagé pour Red Hat 8 ou Rocky Linux 8 (!). Il faut procéder à une installation manuelle.

Sur chacun des nœuds, télécharger les sources depuis le dépôt Github<sup>6</sup>, puis copier les exécutables dans les répertoires adéquats :

```
# ETCD_VER=v3.4.14      # à adapter à la dernière version disponible

# curl -L https://github.com/etcd-io/etcd/releases/download/${ETCD_VER}/\
etcd-${ETCD_VER}-linux-amd64.tar.gz -o /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
# mkdir /tmp/etcd-download
# tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C /tmp/etcd-download \
```

<sup>6</sup><https://github.com/etcd-io/etcd/releases/>

```
--strip-components=1
# rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz

# cp /tmp/etcd-download/etcd* /usr/bin/
# chmod +x /usr/bin/etcd*

# groupadd --system etcd
# useradd -s /sbin/nologin --system -g etcd etcd
# mkdir -p /var/lib/etcd && chmod 700 /var/lib/etcd
# chown -R etcd:etcd /var/lib/etcd/
```

Créer le fichier de service :

```
# cat > /usr/lib/systemd/system/etcd.service <<EOF
[Unit]
Description=Etcd Server
After=network.target

[Service]
User=etcd
Type=notify
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=/etc/etcd/etcd.conf
ExecStart=/usr/bin/etcd
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target

EOF

# systemctl daemon-reload
```

Créer le fichier de configuration :

```
# mkdir /etc/etcd
# touch /etc/etcd/etcd.conf
```

Si le firewall est actif, ouvrir les ports utilisés par etcd, à exécuter sur tous les nœuds :

```
$ firewall-cmd --zone=public --add-port=2379/tcp --permanent
$ firewall-cmd --zone=public --add-port=2380/tcp --permanent
$ firewall-cmd --reload
```

### 5.5.3 Sur Debian/Ubuntu

Sur chacun des nœuds, installer le paquet etcd :

```
# apt install etcd
```

Sur Debian, par défaut, les ports ne sont pas bloqués par un firewall.

### 5.5.4 Configuration de etcd



Par simplicité, ce qui suit utilise des connexions en clair (HTTP). En toute rigueur, il faudrait du HTTPS et gérer des règles de *firewall*.

Liste des variables utilisées dans la procédure à ajuster :

Variable	Description
<i>etcd1-ip</i>	Adresse IPv4 du noeud 1
<i>etcd2-ip</i>	Adresse IPv4 du noeud 2
<i>etcd3-ip</i>	Adresse IPv4 du noeud 3
<i>etcd1-name</i>	Nom du nœud 1. Ex: etcd-1
<i>etcd2-name</i>	Nom du nœud 2. Ex: etcd-2
<i>etcd3-name</i>	Nom du nœud 3. Ex: etcd-3
<i>cluster-token</i>	Jeton etcd à partager entre les différents nœuds pour l'initialisation du <i>cluster</i>
<i>root-password</i>	Mot de passe du super-utilisateur root

Sur chacun des nœuds, éditer la configuration :

- généralement /etc/etc/default.etcd;
- sauf Debian/Ubuntu : /etc/default/etc

Sur le nœud 1 :

```
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://127.0.0.1:2380,http://<etcd1-ip>:2380"
ETCD_LISTEN_CLIENT_URLS="http://127.0.0.1:2379,http://<etcd1-ip>:2379"
ETCD_NAME="<etcd1-name>"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://<etcd1-ip>:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://<etcd1-ip>:2379,http://<etcd2-ip>:2379,
    http://<etcd3-ip>:2379"
ETCD_INITIAL_CLUSTER="<etcd1-name>=http://<etcd1-ip>:2380,<etcd2-
    name>=http://<etcd2-ip>:2380,
    <etcd3-name>=http://<etcd3-ip>:2380"
ETCD_INITIAL_CLUSTER_TOKEN="<cluster-token>"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_ENABLE_V2=true
```

Sur le nœud 2 :

```
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://127.0.0.1:2380,http://<etcd2-ip>:2380"
ETCD_LISTEN_CLIENT_URLS="http://127.0.0.1:2379,http://<etcd2-ip>:2379"
ETCD_NAME="<etcd2-name>"
```

```
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://<etcd2-ip>:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://<etcd1-ip>:2379,http://<etcd2-ip>:2379,
    http://<etcd3-ip>:2379"
ETCD_INITIAL_CLUSTER="<etcd1-name>=http://<etcd1-ip>:2380,<etcd2-
    ↵  name>=http://<etcd2-ip>:2380,
    <etcd3-name>=http://<etcd3-ip>:2380"
ETCD_INITIAL_CLUSTER_TOKEN=""
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_ENABLE_V2=true
```

Sur le nœud 3 :

```
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://127.0.0.1:2380,http://<etcd3-ip>:2380"
ETCD_LISTEN_CLIENT_URLS="http://127.0.0.1:2379,http://<etcd3-ip>:2379"
ETCD_NAME=""
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://<etcd3-ip>:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://<etcd1-ip>:2379,http://<etcd2-ip>:2379,
    http://<etcd3-ip>:2379"
ETCD_INITIAL_CLUSTER="<etcd1-name>=http://<etcd1-ip>:2380,<etcd2-
    ↵  name>=http://<etcd2-ip>:2380,
    <etcd3-name>=http://<etcd3-ip>:2380"
ETCD_INITIAL_CLUSTER_TOKEN=""
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_ENABLE_V2=true
```

### 5.5.5 Gestion du service

Démarrage et activation du service, à exécuter sur tous les nœuds :

```
# systemctl start etcd
# systemctl enable etcd
```

Avec **Debian / Ubuntu**, seul un redémarrage est nécessaire :

```
# systemctl restart etcd
```

### 5.5.6 Traces

Les traces de etcd sont accessibles par `journalctl`, par exemple ainsi :

```
# journalctl -u etcd --no-pager --follow
```

### 5.5.7 Tests du cluster etcd

Vérifier l'état du cluster :

```
$ curl -s -XGET http://<etcd1-ip>:2379/health | python -m json.tool
{
    "health": "true"
}
```

(Noter que `python -m json.tool` peut être remplacé par `jq`, disponible dans les paquets du même nom, ou `python3 -m json.tool` si l'alias nécessaire manque.)

### 5.5.8 Configuration du service pour Patroni

Patroni utilise l'API v2 d'`etcd` (pas encore la v3 au moment où ceci est rédigé). Tout ce qui suit implique que cette variable d'environnement soit en place :

```
# export ETCDCTL_API=2
```

Il est nécessaire d'activer l'authentification pour cette version de l'API.

Ajout du super-utilisateur `root`, à exécuter sur un seul nœud (noter le mot de passe) :

```
# ETCDCTL_API=2 etcdctl user add root  
New password:  
User root created
```

Activation de l'authentification, à exécuter sur un seul nœud :

```
# ETCDCTL_API=2 etcdctl auth enable  
Authentication Enabled
```

L'API v2 d'`etcd` fournit un rôle `guest` ayant accès en lecture et écriture à toute l'arborescence des clés. Il est utilisé pour les accès non authentifiés (c'est-à-dire anonymes) à l'API, ainsi n'importe qui peut lire et écrire.

Il est nécessaire de révoquer ses droits pour garantir la sécurité des accès :

```
# ETCDCTL_API=2 etcdctl -u root role revoke --path=/* --rw guest  
Password:  
Role guest updated
```

Après redémarrage, vérifier que l'authentification est bien activée :

```
$ curl -s -XGET http://<etcd1-ip>:2379/v2/auth/enable | python -m json.tool  
{  
    "enabled": true  
}
```

Les clients peuvent être visualisés ainsi, ce qui donne par exemple :

```
$ curl -s http://10.0.3.101:2379/v2/members |jq  
{  
    "members": [  
        {  
            "id": "736293150f1cffb7",  
            "name": "e1",  
            "peerURLs": [  
                "http://10.0.3.101:2380"  
            ],  
            "clientURLs": [  
                "http://10.0.3.101:2379",  
                "http://10.0.3.102:2379",  
            ]  
        }  
    ]  
}
```

```
        "http://10.0.3.103:2379"
    ],
},
{
  "id": "7ef9d5bb55cefbc",
  "name": "e3",
  "peerURLs": [
    "http://10.0.3.103:2380"
  ],
  "clientURLs": [
    "http://10.0.3.101:2379",
    "http://10.0.3.102:2379",
    "http://10.0.3.103:2379"
  ]
},
{
  "id": "97463691c7858a7b",
  "name": "e2",
  "peerURLs": [
    "http://10.0.3.102:2380"
  ],
  "clientURLs": [
    "http://10.0.3.101:2379",
    "http://10.0.3.102:2379",
    "http://10.0.3.103:2379"
  ]
}
]
```

Ou encore sur chaque nœud :

```
$ curl -s http://10.0.3.101:2379/v2/stats/self | jq
```

```
{
  "name": "e1",
  "id": "736293150f1cffb7",
  "state": "StateFollower",
  "startTime": "2022-04-22T12:50:03.049369234Z",
  "leaderInfo": {
    "leader": "7ef9d5bb55cefbc",
    "uptime": "58m28.992415487s",
    "startTime": "2022-04-22T12:50:04.066655006Z"
  },
  "recvAppendRequestCnt": 59,
  "sendAppendRequestCnt": 0
}
```

### 5.5.9 Exemple de sauvegarde d'information

Pour avoir un aperçu de la manière dont Patroni utilise etcd pour stocker une arborescence de valeurs, nous allons essayer d'y stocker une couple clé/valeur.

Pour écrire une valeur de clé sur n'importe quel nœud, il y a deux possibilités :

- soit la ligne de commande sur les nœuds :

```
# ETCDCTL_API=2 etcdctl -u root set clef valeur  
Password:  
valeur
```

Pour lire la valeur de la clé sur un autre nœud :

```
# ETCDCTL_API=2 etcdctl -u root get clef  
Password:  
valeur
```

Autre exemple : la configuration stockée par un Patroni fonctionne peut se voir ainsi :

```
# ETCDCTL_API=2 etcdctl -u root ls /service/cluster-test-01/  
Password:  
  
/service/cluster-test-01/initialize  
/service/cluster-test-01/config  
/service/cluster-test-01/leader  
/service/cluster-test-01/status  
/service/cluster-test-01/history  
/service/cluster-test-01/failover  
/service/cluster-test-01/members  
  
# ETCDCTL_API=2 etcdctl -u root get /service/cluster-test-01/config  
Password:  
  
{  
  "loop_wait": 10,  
  "maximum_lag_on_failover": 1048576,  
  "postgresql": {  
    "parameters": {  
      "hot_standby": true,  
      "max_replication_slots": 5,  
      "max_wal_senders": 10,  
      "shared_buffers": "64MB",  
      "wal_level": "replica",  
      "work_mem": "70MB"  
    },  
    "use_pg_rewind": true,  
    "use_slots": true  
  },  
  "retry_timeout": 10,  
  "ttl": 30  
}
```

- soit via curl, via l'API, avec un *endpoint* contenant le nom de la clé, ce qui exige de gérer l'authentification.

L'API v2 est disponible en ligne<sup>7</sup>.

---

<sup>7</sup><https://etcd.io/docs/v2.3/api/>

## 5.6 PATRONI



### 5.6.1 Définition



- Patroni est un outil qui permet de :
  - créer / mettre en réPLICATION
  - maintenir
  - monitorer
- des serveurs PostgreSQL en haute disponibilité.

Patroni est un script écrit en Python. Il permet de maintenir un agrégat d'instances PostgreSQL en condition opérationnelle et de le superviser afin de provoquer une bascule automatique en cas d'incident sur le primaire.

Il s'agit donc d'un outil permettant de garantir la haute disponibilité du service de bases de données.

### 5.6.2 Mécanismes mis en œuvre



- RéPLICATION physique et outils standards (pg\_rewind, pg\_basebackup)
- Sauvegarde PITR (barman, pgBackRest...)
- Gestionnaire de service : Patroni
- DCS : Serveur de configurations distribuées (etcd ou autres)

Patroni est un script Python qui s'appuie sur la capacité de l'écosystème PostgreSQL à répliquer les modifications et les rejouer sur un stockage clé valeur distribué pour garantir la haute disponibilité de PostgreSQL.

**Outils :**

La réPLICATION PHYSIQUE fournie avec PostgreSQL assure la haute disponibilité des données. Des outils fournis avec PostgreSQL comme pg\_rewind<sup>8</sup> et pg\_basebackup<sup>9</sup> sont utilisés pour construire ou reconstruire les instances secondaires. Nous les décrivons dans le module de formation I4<sup>10</sup>.

Cette capacité est étendue grâce à la possibilité d'utiliser des outils de la communauté comme barman<sup>11</sup>, pgBackRest<sup>12</sup> ou WAL-G<sup>13</sup>.

#### **DCS (etcd) :**

Patroni s'appuie sur un gestionnaire de configuration distribué (DCS) pour partager l'état des nœuds de son agrégat et leur configuration commune.

Dans ce document, nous ne mentionnerons que etcd mais il est cependant possible d'utiliser un autre DCS tel que Consul, Zookeeper ou même Kubernetes.



Notre but étant la haute disponibilité, etcd ne doit pas devenir un SPOF (*single point of failure*), et doit donc être lui aussi mis en œuvre en haute disponibilité.

#### **Raft dans Patroni :**

L'algorithme Raft utilisé dans etcd a également été implémenté dans Patroni en version 2... mais cette option est déjà dépréciée dans la version 3 de Patroni et ne doit donc pas être utilisée.

### **5.6.3 Bascule automatique**



- *leader lock*
- *heart beat*
- *split-brain*
- *followers* demandant une élection au quorum
- Promotion automatique

La décision de la bascule automatique est assurée par un mécanisme de verrou partagé appelé *leader lock*, qui est attribué à une seule instance sur la base d'une élection effectuée en fonction de la disponibilité du secondaire choisi et de son LSN (*Log Sequence Number*, ou position dans le flux des journaux de transaction) courant.

<sup>8</sup><https://www.postgresql.org/docs/current/app-pgrewind.html>

<sup>9</sup><https://www.postgresql.org/docs/current/app-pgbasebackup.html>

<sup>10</sup>[https://dali.bo/i4\\_html](https://dali.bo/i4_html)

<sup>11</sup><https://www.pgbarman.org/>

<sup>12</sup><https://pgbackrest.org/>

<sup>13</sup><https://github.com/wal-g/wal-g>

La présence de deux primaires dans un agrégat d'instance nous amène à une situation que nous voulons éviter à tout prix : le *split-brain*.

#### 5.6.4 Définition : *split-brain*



- 2 primaires sollicités en écriture
- Arbitrage très difficile
- Perte de données
- Indisponibilité du service

La situation d'un *split-brain* est obtenue lorsque l'élection automatique d'un primaire est possible sur deux nœuds différents, au même moment.

Les données insérées sur les deux nœuds doivent faire l'objet d'un arbitrage pour départager les-  
quelles seront gardées lors du rétablissement d'une situation normale (un seul primaire).

La perte de données est plus probable lors de cet arbitrage, suivant la quantité et la méthode  
d'arbitrage.

Le service peut souffrir d'une indisponibilité s'il y a nécessité de restauration partielle ou totale des  
données.

#### 5.6.5 Leader lock de Patroni



- Verrou attribué au primaire de manière unique
- Communication entre les nœuds Patroni
  - comparaison des LSN
- Nouveau primaire :
  - nouvelle *timeline*
  - nouvelle chaîne de connexions
  - les secondaires se raccrochent au primaire

L'attribution unique d'un verrou appelé *leader lock* par Patroni permet de se prémunir d'un *split-brain*.  
Ce verrou est distribué et stocké dans le DCS.

Une fois ce verrou obtenu, le futur primaire dialogue alors avec les autres nœuds Patroni référencés dans le DCS, et valide sa promotion en comparant leur type de réPLICATION (synchrone ou asynchrone) et leur LSN courant.

La promotion provoque la création d'une nouvelle *timeline* sur l'instance primaire et la chaîne de connexion (`primary_conninfo`) utilisée par les instances secondaires pour se connecter au primaire est mise à jour.

Les secondaires se raccrochent ensuite à la *timeline* du primaire.

### 5.6.6 Heartbeat



- Tous les nœuds Patroni s'annoncent à etcd
  - primaire ou *follower*
- Conséquences d'une perte de contact avec le *leader* :
  - *time out* sur les *followers* et bascule

Chaque nœud est en communication régulière avec l'agrégat etcd afin d'informer le système de sa bonne santé. Le primaire confirme son statut de *leader* et les secondaires celui de *follower*.

Lorsque la confirmation du *leader* ne vient pas, un *time out* est atteint sur les *followers* ce qui déclenche une procédure de bascule.

### 5.6.7 Bootstrap de nœud



- Création ou recréation de nœud
- Reconstruction :
  - `pg_basebackup` depuis primaire
  - `pgBackRest` depuis sauvegarde PITR (delta !)

L'opération de *bootstrap* consiste à recréer l'instance PostgreSQL d'un nœud, à partir du nœud primaire ou d'une sauvegarde.

Par défaut, Patroni lance un `pg_basebackup` pour récupérer l'instance du primaire. Cependant, sur des gros volumes, cela n'est pas acceptable.

Dans ce genre de situation, il est possible de paramétrier Patroni pour qu'il utilise un autre outil de restauration de sauvegarde PITR tel que pgBackrest.

Pour reconstruire un nœud ayant pris trop de retard, pgBackrest permet de raccrocher rapidement une instance ayant un volume de données important grâce à la restauration en mode delta.



Si l'archivage ou la sauvegarde sont endommagés, Patroni utilise l'alternative pg\_basebackup pour recréer l'instance.

### 5.6.8 Cas particulier du multisite



- Avec deux sites
- Avec trois sites

### 5.6.9 Deux sites



- Prévoir la perte d'un des 2 sites
- Quorum impossible
  - au pire, passage en *read only* !

Le but de configurer un deuxième site est de disposer d'une tolérance de panne à l'échelle d'un site.

En cas d'incident majeur, le deuxième site est censé prendre la relève.

Cependant, dans le cas d'un agrégat à deux sites, il est par exemple impossible de différencier la perte totale d'une salle distante d'une coupure réseau locale. Il est tentant de « favoriser » une salle en y positionnant une majorité de nœuds de l'agrégat, mais cette approche réduit au final la disponibilité de service. Détaillons :

- Cas 1 : primaire et majorité sur le site A
  - si perte du site A : le site B n'a pas le quorum : pas de bascule
  - si perte du site B : aucun impact sur le primaire
- Cas 2 : primaire sur le site A, majorité sur le site B
  - si perte du site A : bascule vers le site B

- si perte du site B : perte du quorum, **l'instance passe en mode stand-by sur le site 0**

Dans le cas d'un agrégat multi-site, la réponse à cette problématique est d'utiliser au minimum trois sites. En cas de perte de réseau d'un des sites, ce dernier perd alors le quorum et les instances PostgreSQL ne peuvent y être démarrées qu'en *stand-by*. Les deux autres sites continuent à communiquer entre eux et conservent ainsi le quorum. L'instance primaire peut être démarrée ou maintenue sur l'un de ces deux sites sans risque de *split-brain*.



En conclusion, sans troisième site, il n'est pas possible de mettre en œuvre une mécanique de bascule automatique fiable et anti *split-brain*.

#### 5.6.9.1 Trois sites



- 3 sites pour un quorum
- Tolérance de panne accrue
- Changement de site lors d'une bascule
- Perte de 2 sites sur 3
  - 3<sup>e</sup> site en *stand-by*

#### Quorum de sites

La présence de trois sites permet de disposer de suffisamment de nœuds pour construire un quorum nécessaire à la bascule automatique.

En effet, on a vu que pour départager deux sites en concurrence, il est nécessaire de disposer d'un arbitre externe et donc d'un troisième site.

#### Tolérance de panne accrue

La tolérance de panne peut être bien plus importante si l'on décide de prévoir la perte totale de deux sites sur les trois, mais elle apporte une complexité supplémentaire et ne permet pas un automatisme complet jusqu'au bout (après la perte de deux sites).

À tout moment, le maintien du service nécessite de disposer de :

- un nombre suffisant de nœuds pour le DCS, plus précisément d'un quorum de nœuds fonctionnels, ce qui dépend du nombre total de nœuds déclarés dans l'agrégat et de leur répartition entre les trois sites ;
- suffisamment de nœuds Patroni sur chaque site, pour que malgré la perte totale d'un ou deux sites, nous puissions disposer d'une tolérance de panne minimale, sur chaque site.

En conséquence, une solution hautement disponible peut être de disposer de trois nœuds pour le DCS et de deux nœuds Patroni au minimum, ceci pour chaque site.

### **Changement de site lors des bascules**

Lors de la bascule automatique, Patroni décide de privilégier les nœuds les plus à jour avec le primaire, puis les plus réactifs. Il n'y a donc aucune garantie que dans certaines conditions particulières, un nœud d'un autre site soit promu plutôt qu'un nœud géographiquement local.

### **Perte de deux sites sur trois**

S'il ne reste plus qu'un site disponible, promouvoir un de ses nœuds secondaires ne pourra être fait qu'après plusieurs opérations **manuelles** visant à **reconstruire un quorum<sup>14</sup> de nœuds dans le DCS** en enlevant les nœuds défaillants de sa liste (les nœuds des deux autres sites).



Pour rappel, il faut toujours retirer les nœuds défaillants du DCS avant d'en ajouter pour les remplacer, sous peine de ne jamais récupérer de quorum malgré l'ajout.

### **3ème site en stand-by**

Une autre possibilité consiste à garder le troisième site en dehors du mécanisme de bascule automatique. Il est prévu de ne le solliciter que manuellement, après avoir constaté la perte totale des deux premiers sites.

## **5.6.10 Configuration de Patroni**



- Configuration dynamique
  - stockée dans le DCS
  - initialisée depuis la section `bootstrap.dcs` du fichier de configuration YAML
  - modifiable ensuite par `patronictl edit-config`
  - prise en compte immédiatement si possible
  - copiée dans `$PGDATA/patroni.dynamic.json` à intervalle régulier
- Configuration statique
  - stockée dans le fichier de configuration YAML de chaque nœud
  - recharge par `patronictl reload`
- Variables d'environnement

<sup>14</sup>Le quorum se calcule ainsi :  $(3 \text{ sites} \times 3 \text{ nœuds}) / 2 + 1$

La configuration de patroni<sup>15</sup> est répartie sur trois niveaux de configuration.

La **configuration dynamique** est initialisée grâce aux données de la section `bootstrap.dcs` du fichier de configuration YAML. Elle est chargée dans le DCS et n'est plus maintenue qu'à cet endroit. Les modifications doivent être faites avec la commande `patronictl edit-config`. Patroni copie cette configuration à intervalle régulier dans le fichier `patroni.dynamic.json` placé dans le répertoire de données de l'instance.

Toutes les informations présentes dans cette section ne sont utilisées que lors de la création du cluster. Cela inclut la configuration dynamique qui est dans la section `bootstrap.dcs`, mais aussi la configuration de PostgreSQL effectuée dans les sections `bootstrap.initdb`, `bootstrap.method`, `bootstrap.pg_hba`, `bootstrap.users`.

La **configuration statique** peut être située dans un fichier YAML ou un répertoire. Si un répertoire est spécifié, tous les fichiers YAML du répertoire seront chargés dans l'ordre où ils apparaissent. Si des paramètres sont définis deux fois, seul la dernière valeur est pris en compte. Les modifications faites dans la configuration statique peuvent être chargées par l'une des méthodes suivantes :

- le rechargement du service (si disponible) ;
- la commande `patronictl reload` ;
- un signal HUP (`reload`) au processus patroni.

Les éléments de configuration suivants sont présents dans ce fichier :

- configuration de l'API REST de Patroni (section `restapi`) ;
- configuration des interactions avec un DCS au choix dont `etcd` (section `etcd` ou `etcd3`) ;
- configuration des interactions avec PostgreSQL (section `postgresql`) et de sa configuration (`postgresql.conf`, `pg_hba.conf`, `pg_ident.conf`) ;
- configuration du `watchdog`.

Les modifications de cette configuration doivent être reportées dans le fichier de configuration de tous les nœuds si on souhaite qu'elles soient prises en compte globalement. Cela peut être fait avec des outils d'industrialisation comme Ansible.

Le fichier de configuration peut être testé avec l'exécutable de Patroni :

```
$ patroni --validate-config /etc/patroni/demo.yaml
```

Pour finir, certains éléments de configuration peuvent être spécifiés via des **variables d'environnement**. Les variables d'environnement ont toujours précédence sur les fichiers de configuration.

L'application de la configuration locale ou dynamique de PostgreSQL suit l'algorithme suivant :

- **SI** `postgresql.custom_conf` est défini dans la configuration YAML de patroni
  - **ALORS** le fichier configuré dans `custom_conf` sera utilisé comme référence de configuration en lieu et place de `postgresql.base.conf` et `postgresql.conf`
  - **SINON**
    - \* **SI** le fichier `postgresql.base.conf` existe

---

<sup>15</sup>[https://patroni.readthedocs.io/en/latest/dynamic\\_configuration.html](https://patroni.readthedocs.io/en/latest/dynamic_configuration.html)

- **ALORS** il sera utilisé comme référence de la configuration
  - **SINON** le fichier `postgresql.conf` est utilisé et renommé en `postgresql.base.conf`
- La configuration dynamique est écrite dans `postgresql.conf` à l'exception de certains paramètres qui seront passés directement en paramètre du postmaster (voir plus loin).
  - Un `include` vers la configuration de référence est ajouté au fichier `postgresql.conf`
  - **SI** des paramètres nécessitent un redémarrage (`pg_settings.context`)
    - **ALORS** un flag `pending_restart` est donné au nœud et sera retiré lors du prochain redémarrage.

L'ordre de prise en compte des paramètres postgresql est donc le suivant :

1. paramètres présents dans `postgresql.base.conf` ou spécifiés dans `custom_conf` ;
2. paramètres présents dans `postgresql.conf` ;
3. paramètres présents dans `postgresql.auto.conf` ;
4. paramètres défini au lancement de PostgreSQL avec l'option `-o --name=value`.

### 5.6.10.1 Configuration globale de Patroni



Paramètres YAML globaux de Patroni :

- name
- namespace
- scope

Les paramètres globaux (de premier niveau) de la configuration de Patroni sont :

- name : le nom de l'hôte, doit être unique au sein du cluster ;
- namespace : le chemin dans lequel sera stockée la configuration de Patroni dans le DCS (défaut: /service) ;
- scope : le nom de l'agrégat, utilisé dans les commandes `patronictl` et pour stocker la configuration de l'agrégat dans le DCS. Ce paramètre est requis et doit être identique sur tous les nœuds.

Exemple de fichier de configuration :

```
name: p1
scope: patroni-demo
<...>
```

Voici un exemple de l'arborescence créée dans le DCS pour la configuration ci-dessus et avec deux serveurs **p1** et **p2** :

```
$ ETCDCTL_API=3 etcdctl get --prefix / --keys-only
/service/patroni-demo/config
/service/patroni-demo/initialize
/service/patroni-demo/leader
/service/patroni-demo/members/p1
/service/patroni-demo/members/p2
/service/patroni-demo/status
```

### 5.6.10.2 Configuration dynamique



- bootstrap.dcs
- bootstrap.dcs.parameters
- bootstrap.dcs.standby\_cluster
- bootstrap.dcs.slots
- bootstrap.dcs.ignore\_slots

Les éléments de configuration du cluster sont stockés dans le DCS sous l'arborescence /<namespace>/<scope>/config. Cette arborescence est initialisée à partir de la section bootstrap.dcs du fichier YAML. Après l'initialisation de l'instance, cette configuration est visible via la commande `patronictl show-config` et modifiable via la commande `patronictl edit-config`.

```
$ patronictl -c /etc/patroni/demo.yaml show-config

loop_wait: 10
master_start_timeout: 300
postgresql:
  parameters:
    archive_command: /bin/true
    archive_mode: 'on'
  use_pg_rewind: false
  use_slot: true
retry_timeout: 10
ttl: 30
```

Les paramètres suivant concernent la configuration de Patroni :

- `loop_wait`: temps de pause maximal entre chaque boucle de vérification (défaut: 10s). Cela correspond au temps nominal de la boucle de vérification ;
- `ttl` : temps avant l'initialisation d'un *failover* (défaut : 30s). Cela correspond au temps maximal entre deux mises à jour de la *leader key* ;
- `retry_timeout` : temps avant de retenter une action échouée sur PostgreSQL ou le DCS (défaut: 10s) ;
- `maximum_lag_on_failover` : limite supérieure du délai (*lag*), en octets, pour qu'un nœud *follower* puisse participer à une élection ;
- `max_timelines_history` : quantité maximale de changements de timeline conservés dans l'historique stocké dans le DCS. Pour une valeur de 0, Patroni conserve tout l'historique (défaut: 0) ;
- `master_start_timeout` : le temps maximal autorisé pour qu'une instance primaire redevienne fonctionnelle suite à un incident. Si la valeur est nulle et que l'état du cluster le permet, Patroni déclenche un *failover* immédiat suite à un incident, ce qui peut provoquer une perte de donnée dans le cas d'une réPLICATION asynchrone (défaut: 300) ;

- **master\_stop\_timeout** : le temps maximal autorisé pour l'arrêt de PostgreSQL lorsque le mode synchrone est activé. Si la valeur est supérieure à zéro et que le mode synchrone est activé, Patroni envoie un signal SIGKILL au postmaster s'il met trop de temps à s'arrêter (défaut: 0) ;
- **synchronous\_mode** : activation de la réPLICATION synchrone. Dans ce mode, un ou des répliCAs sont choisis comme *followers* synchrones. Seul ces répliCAs et le *leader* peuvent participer à une élection ;
- **synchronous\_node\_count** : nombre de *followers* synchrones que Patroni doit chercher à maintenir. Pour cela, Patroni choisit des *followers* synchrones parmi les répliCAs disponibles et valorise le paramètre **synchronous\_standby\_names** dans la configuration de PostgreSQL en listant uniquement ces nœuds (ex: 2 (p2, p3)). Si un nœud disparaît de l'agrégat, Patroni le retire de **synchronous\_standby\_names**. Par défaut, Patroni fera en sorte de ne pas bloquer les écritures en allant jusqu'à diminuer le nombre de nœuds synchrones jusqu'à potentiellement désactiver la réPLICATION synchrone. Les *followers* synchrones sont aussi listés dans l'entrée `/namespace/scopE/sync` de etcd ;
- **synchronous\_mode\_strict** : empêche la désactivation du mode synchrone dans le cas où il n'y a plus de *follower* synchrone. Pour cela, Patroni valorise **synchronous\_standby\_names** à \* quand il n'y a plus de candidats, ce qui bloque les écritures en attendant le retour d'un *follower* synchrone. Ce paramètre peut donc provoquer une indisponibilité du service ;
- **maximum\_lag\_on\_syncnode** : limite supérieure du *lag* en octets pour qu'un nœud *follower* synchrone soit considéré comme non sain et remplacé par un *follower* asynchrone sain. Une valeur inférieure ou égale à zéro désactive ce comportement. Une valeur trop basse peut provoquer des changements de *follower* synchrone trop fréquents (défaut : -1) ;
- **check\_timeline** : vérifie que la *timeline* du candidat est bien la plus élevée avant d'effectuer la promotion (défaut: false) ;
- **failsafe\_mode** : ce mode permet d'éviter le déclenchement d'une opération de déMOTE sur l'instance primaire lorsque le DCS est indisponible. Pour cela, une nouvelle clé / **failsafe** est ajoutée au DCS. Elle est maintenue par l'instance primaire et contient la liste des membres autorisés à participer à une élection. En cas de perte du DCS, si tous les membres de la clé **failsafe** sont joignables, l'instance primaire garde son rôle. Si l'un d'entre eux ne répond pas, l'instance primaire perd son rôle de *leader*. (défaut : false).

La configuration de PostgreSQL peut également être chargée dans le DCS au moment de l'initialisation depuis la clé `bootstrap.dcs.postgresql`. Cela permet d'avoir une configuration commune pour toutes les instances. Les paramètres notables sont:

- **use\_pg\_rewind** : activation de l'outil pg\_rewind pour la reconstruction d'une standby après un failover (défaut: false) ;
- **use\_slots** : utilisation des slots de réPLICATION (défaut: true pour les versions de PostgreSQL supérieures à la 9.4). **use\_slots** doit forcément être défini dans cette section ;
- **recovery\_conf** : configuration supplémentaire à ajouter au `recovery.conf`. Même si ce fichier disparaît à partir de la version 12 de PostgresQL, la section peut continuer à être utilisée.

Les sections suivantes sont également présentes :

- **parameters** : paramètres de configuration PostgreSQL;
- **pg\_hba** : liste des règles d'authentification par hôtes ;

- pg\_ident : liste des équivalences entre utilisateur OS et PostgreSQL.

Certains éléments de configuration de PostgreSQL sont initialisés par Patroni et doivent être modifiés et stockés dans le DCS de manière inconditionnelle, soit parce que PostgreSQL requiert qu'ils soient identiques partout, soit par choix d'implémentation. Voici la liste :

- max\_connections : nombre maximal de connexions simultanées (défaut : 100) ;
- max\_locks\_per\_transaction : nombre maximal de verrous par transaction (défaut : 64) ;
- max\_worker\_processes : nombre maximum de processus *worker* (défaut : 8) ;
- max\_prepared\_transactions : nombre maximal de transactions préparées (défaut : 0) ;
- wal\_level : niveau de détail des informations écrites dans les WAL (défaut : replica) ;
- wal\_log\_hints : force l'écriture complète d'une page de données lors de sa première modification après un *checkpoint*, même pour des modifications non critiques comme la mise en place de *hint bits*. Cela permet l'utilisation de pg\_rewind (défaut : on) ;
- track\_commit\_timestamp : permet de tracer l'horodatage des commits dans les journaux de transactions (défaut : off) ;
- max\_wal\_senders : nombre maximal de processus *wal sender* (défaut : 5) ;
- max\_replication\_slots : nombre maximal de slots de réPLICATION (défaut : 5) ;
- wal\_keep\_segments : nombre maximal de WAL conservés pour les instances secondaires afin de les aider à récupérer leur retard (jusque PostgreSQL 12) (défaut : 8) ;
- wal\_keep\_size : quantité de WAL conservés pour les instances secondaires afin de les aider à récupérer leur retard (à partir de PostgreSQL 13) (défaut : 128MB).
- listen\_addresses : la ou les interfaces sur lesquelles PostgreSQL écoute, ce paramètre est défini via le paramètre postgresql.listen ou la variable d'environnement PATRONI\_POSTGRESQL\_LISTEN ;
- port : le port sur lequel écoute l'instance est lui aussi défini via le paramètre postgresql.listen ou la variable d'environnement PATRONI\_POSTGRESQL\_LISTEN ;
- cluster\_name : permet de définir le nom de l'instance qui sera affiché dans la description des processus (ps) et est défini en fonction de la valeur du paramètre scope ou de la variable d'environnement PATRONI\_SCOPE ;
- hot\_standby : permet d'ouvrir les instances secondaires en lecture seule (défaut : on).

Afin de faire en sorte que ces paramètres ne puissent pas être modifiés via les fichiers de configuration, ils sont passés en paramètre de la commande de démarrage de PostgreSQL. Ce n'est le cas **que** pour la liste ci-dessus. Les autres paramètres postgresql.parameters définis dans le DCS sont appliqués de manière classique en suivant la hiérarchie décrite précédemment.

Voici un exemple de configuration de la section bootstrap.dcs incluant la configuration de PostgreSQL :

```
bootstrap:  
  dcs:  
    ttl: 30  
    loop_wait: 10  
    retry_timeout: 10  
    maximum_lag_on_failover: 1048576  
    postgresql:  
      use_pg_rewind: true  
      use_slots: true
```

```
parameters:
  wal_level: replica
  hot_standby: "on"
  wal_keep_segments: 8
  max_wal_senders: 5
  max_replication_slots: 5
  checkpoint_timeout: 30
pg_hba:
  - local all all peer
  - host all all 0.0.0.0/0 scram-sha-256
  - host replication replicator 10.0.30.12/32 scram-sha-256
  - host replication replicator 10.0.30.13/32 scram-sha-256
pg_ident:
  - superusermapping root postgres
  - superusermapping dba postgres
#recovery.conf:
```

Patroni permet de créer un agrégat de secours (*standby cluster*) en utilisant la réPLICATION en cascade de PostgreSQL. Pour se faire, un second agrégat est créé et son *leader* (le *standby leader*) se connecte à un serveur de l'agrégat principal via le protocole de réPLICATION. Les *followers* de ce second agrégat se connectent, eux, au *standby leader*, pour réPLiquer les modifications.

La section `bootstrap.dcs.standby_cluster` est utilisée lorsque l'on crée un agrégat de secours<sup>16</sup>. Les éléments suivants sont définis pour cette section :

- `host` : adresse du serveur primaire ;
- `port` : port du serveur primaire ;
- `primary_slot_name` : indique le slot du serveur primaire à utiliser pour la réPLICATION. Si ce paramètre n'est pas utilisé le nom sera dérivé du nom du serveur primaire. Pour cela, le nom est converti en Unicode, les espaces et tirets sont remplacés par des underscores et le nom est tronqué à 64 caractères. Le slot doit être créé manuellement et ne sera par défaut pas maintenu par Patroni.
- `create_replica_methods` : une liste de méthode à employer pour faire l'initialisation du leader de l'agrégat de secours (voir l'option `bootstrap.method` décrite plus loin pour plus de détails) ;
- `restore_command` : commande utilisée pour récupérer les WAL via le *log shipping* ;
- `archive_cleanup_command` : commande pour supprimer les journaux de transaction du répertoire d'archivage une fois qu'ils ne sont plus nécessaires ;
- `recovery_min_apply_delay` (en millisecondes) : délai d'application des modifications sur le *leader* de l'agrégat de secours, et équivalent au paramètre de même nom de PostgreSQL.

Lorsque le service Patroni est démarré, il initialise le cluster en se connectant à l'instance spécifiée.

---

<sup>16</sup>[https://patroni.readthedocs.io/en/latest/replica\\_bootstrap.html#standby-cluster](https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#standby-cluster)



Afin de s'assurer que les WAL soient toujours disponibles pour l'agrégat de secours, il est possible d'utiliser un slot permanent avec la section `bootstrap.dcs.slots` (voir plus loin).

Il est préférable de disposer d'une IP virtuelle (VIP) sur le serveur primaire de l'agrégat primaire. De cette façon, si on perd le serveur source de la réPLICATION vers le standby leader, la réPLICATION basculera vers un autre nœud. Il est possible qu'il y ait une divergence en cas de *failover* sur l'agrégat primaire.

Voici un exemple de configuration pour la création du leader d'un standby cluster. Le nom de cluster a été changé, une section `bootstrap.dcs.standby_cluster` a été créée et le mot de passe de l'utilisateur de replication a été configuré.

```
scope: patroni-demo-standby
name: p3
<...>
bootstrap:
  dcs:
    <...>
    standby_cluster:
      host: 10.20.89.3
      port: 5432
<...>
postgresql:
  authentication:
    replication:
      username: replicator
      password: repass
<...>
```

On peut confirmer qu'un cluster a été créé dans le DCS pour les nœuds **p3** et **p4** (créé séparément) et que la configuration du *standby cluster* a été adaptée.

Les commandes suivantes sont lancées depuis un serveur etcd.

```
$ ETCDCTL_API=3 etcdctl get --keys-only --prefix /service/patroni-demo-standby
/service/patroni-demo-standby/config
/service/patroni-demo-standby/initialize
/service/patroni-demo-standby/leader
/service/patroni-demo-standby/members/p3
/service/patroni-demo-standby/members/p4
/service/patroni-demo-standby/status

$ ETCDCTL_API=3 etcdctl get --print-value-only /service/patroni-demo-standby/config
↪ | python -m 'json.tool'

{
  "loop_wait": 10,
  "master_start_timeout": 300,
  "postgresql": {
    "parameters": {
      "archive_command": "/bin/true",
```

```

        "archive_mode": "on"
    },
    "use_pg_rewind": false,
    "use_slot": true
},
"retry_timeout": 10,
"standby_cluster": {
    "host": "10.20.89.3",
    "port": 5432
},
"ttl": 30
}

```

Le leader est marqué comme *standby leader* dans patronictl :

```
$ patronictl list
```

Member	Host	Role	State	TL	Lag in MB
<b>Cluster: patroni-demo-standby (7148326914433478989)</b>					
p3	10.20.89.5	Standby Leader	running	1	
p4	10.20.89.6	Replica	running	1	0

Pour conclure, Patroni dispose de deux sections dédiées aux slots de réPLICATION.

Lors d'une bascule, le comportement par défaut de Patroni lorsqu'il rencontre un slot de réPLICATION qui n'est pas maintenu par lui est de le supprimer.

La section `bootstrap.dcs.slots` permet de définir des slots de réPLICATION permanents qui seront préservés lors d'un *switchover* ou *failover*.

- les slots de réPLICATION physique sont créés sur la nouvelle instance primaire au moment de la promotion ou de leur définition dans la configuration ;
- les slots de réPLICATION logique seront créés sur tous les nœuds et leur position sera avancée sur les *followers* à chaque fois que `loop_wait` est écoulé. L'utilisation de cette fonctionnalité requiert la présence du paramètre `bootstrap.dcs.postgresql.use_slots`. De plus le paramètre `hot_standby_feedback` sera activé sur les instances secondaires par Patroni.

Après la mise en place de la configuration, si les slots définis dans cette section n'existent pas, Patroni va tenter de les créer. Si c'est le résultat attendu, il faut vérifier que la création a bien fonctionné. Une erreur dans le paramétrage (nom de base, ou de plugin) ne sera pas vue lors de la validation du paramétrage et fera échouer la création du slot.

Attention, la suppression des slots dans la configuration supprime aussi les slots sur les instances.

Ces slots peuvent être décrits en créant une section portant le nom du slot qui contient les paramètres suivants :

- `type` : le type du slot : `physical` ou `logical` ;
- `database` : le nom de la base de données pour laquelle le slot de réPLICATION logique est créé ;
- `plugin` : le plugin de décodage utilisé par le slot de réPLICATION logique.

Le nommage du slot de réPLICATION persistant doit être fait en gardant à l'esprit qu'il faut éviter les collisions de nom avec les slots créés par Patroni pour les besoins de la réPLICATION en flux utilisée par l'agrégat.

La section `ignore_slots` permet de donner à Patroni une liste de slots de réPLICATION à ignorer.

- `name` : un nom de slot ;
- `type` : un type de slot (`logical` ou `physical`) ;
- `database` : le nom de la base de données sur laquelle un slot de réPLICATION logique est défini ;
- `plugin` : un plugin de décodage logique utilisé par le slot.

Voici un exemple de configuration des slots :

```
bootstrap:  
  dcs:  
    <....>  
  slots:  
    replication_logique:  
      type: logical  
      database: magasin  
      plugin: test_decoding  
    standby_cluster:  
      type: physical  
  ignore_slots:  
    - name: replication_logique_app  
      type: logical  
      database: magasin  
      plugin: test_decoding  
    - name: standby_hors_aggregat  
      type: physical
```

### 5.6.10.3 Paramétrage spécifique à la création de nouvelles instances (bootstrap)



- bootstrap.method et bootstrap.initdb
- bootstrap.pg\_hba
- bootstrap.postgresql
- bootstrap.users

Les sections `bootstrap.method` permet de décrire la manière dont l'instance PostgreSQL sera créée. La méthode `initdb` est la méthode par défaut. Il est possible d'utiliser d'autres commandes<sup>17</sup>.

L'exemple suivant utilise une sauvegarde pgBackRest pour initialiser le cluster. La commande à utiliser est spécifiée avec le paramètre `command`. Si l'option `no_params` est à `False` les paramètres `--scope` et `--datadir` qui définissent respectivement le nom du cluster et le chemin de l'instance seront ajoutés à la commande. Pour finir, le paramètre `keep_existing_recovery_conf` permet de conserver le `recovery.conf` généré par pgBackRest.

```
bootstrap:
  <...>
  method: pgbackrest
  pgbackrest:
    command: /bin/bash -c "pgbackrest --stanza=test restore"
    keep_existing_recovery_conf: True
    no_params: True
<...>
```

Si la méthode `initdb` est choisie, une section spécifique doit être renseignée. Elle permet de spécifier si les sommes de contrôles doivent être activées (`data-checksums`) et de définir une locale et un encodage.

```
bootstrap:
  <...>
  # ici la méthode est facultative, initdb est la valeur par défaut
  method: initdb
  initdb:
    - data-checksums
    - encoding: UTF8
    - locale: UTF8
<...>
```

La section `bootstrap.pg_hba` permet de spécifier une configuration des méthodes d'authentification spécifiques à la création de l'instance.

Voici un exemple :

<sup>17</sup>[https://patroni.readthedocs.io/en/latest/replica\\_bootstrap.html#custom-bootstrap](https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#custom-bootstrap)

```
bootstrap:  
  <...>  
pg_hba:  
  - host all all 0.0.0.0/0 scram-sha-256  
  - host replication replicator 10.0.30.13/32 scram-sha-256  
<...>
```

La section `bootstrap.users` permet de créer et configurer des rôles PostgreSQL supplémentaires.

Voici un exemple :

```
bootstrap:  
  <...>  
users:  
  admin:  
    password: password_admin  
    options:  
      - createrole  
      - createdb  
<...>
```

Pour conclure : des scripts peuvent être déclenchés après l'initialisation de l'instance avec les sections `bootstrap.post_bootstrap` ou `bootstrap.post_init`. Le script recevra en paramètre une URL de connexion avec comme utilisateur le super utilisateur de l'instance. La variable d'environnement `PGPASSFILE` sera également créée.

#### 5.6.10.4 Configuration des traces



- par défaut dans le fichier de trace système
- log
  - level
  - format
  - dir
  - file\_num
  - file\_size

Patroni étant écrit en python, la configuration des traces sera familière aux utilisateurs de ce langage.

Si aucune configuration des traces n'est faite, les traces sont envoyées vers le fichier de traces système.

Les paramètres suivant peuvent être configurés pour contrôler l'affichage des traces :

- level : niveau de trace parmi CRITICAL, ERROR, WARNING, INFO et DEBUG (défaut : INFO) ;
- traceback\_level : niveau de trace à partir duquel les tracebacks seront visibles (défaut : ERROR) ;
- format : format des traces<sup>18</sup> (défaut: %(asctime)s %(levelname)s: %(message)s) ;
- dateformat : format de date<sup>19</sup> ;
- max\_queue\_size : Patroni implémente une trace en deux temps. Il écrit les traces en mémoire et un thread séparé écrit les traces vers un fichier ou la sortie standard. La quantité de traces gardée en mémoire est par défaut de 1000 enregistrements ;
- dir : le répertoire où seront écrites les traces. Patroni doit avoir les droits en écriture sur ce répertoire ;
- file\_num : nombre de fichiers de trace applicative à conserver (défaut : 4) ;
- file\_size : taille maximale du fichier de trace applicative avant qu'un nouveau soit créé (défaut: 25MB) ;
- loggers : cette section permet de définir un niveau de trace par module python.

Exemple :

```
<...>
log:
  level: INFO
  dir: /var/log/patroni
<...>
```

<sup>18</sup><https://docs.python.org/3.10/library/logging.html#logrecord-attributes>

<sup>19</sup><https://docs.python.org/3.10/library/logging.html#logging.Formatter.formatTime>

### 5.6.10.5 Configuration du DCS (etcd / etcd3)



- DCS possibles : etcd, Consul, ZooKeeper, Exhibitor, Kubernetes
- etcd|etcd3
  - host
  - protocol
  - username, password
  - cacert, cert, key

Patroni supporte six types de DCS différents : etcd, Consul, Zookeeper, Exhibitor, Kubernetes.

Dans cette formation, on ne décrira que l'utilisation d'etcd, qui dispose de deux sections en fonction de la version du protocole d'etcd utilisée :

- etcd pour le protocole v2 ;
- etcd3 pour le protocole v3, c'est l'API qui est configurée par défaut dans etcd depuis qu'il est en version version 3.4. Son implémentation dans Patroni est considérée comme prête pour la production depuis sa version 2.1.5. Il est donc recommandé de l'utiliser lorsque les versions utilisées le permettent.

Les paramètres suivants permettent de configurer l'accès au DCS :

- host et hosts : permettent de définir au choix un ou une liste de *endpoints* au format `host:port` ;
- protocol : le protocole utilisé parmi `http` et `https` (défaut : `http`) ;
- username : utilisateur pour l'authentification à etcd ;
- password : mot de passe pour l'authentification à etcd ;
- cacert : le certificat du serveur d'autorité de certification (active le SSL si présent) ;
- cert : certificat du client ;
- key : clé du client (peut être ignorée si elle fait partie du certificat).

D'autres paramètres sont décrits dans la documentation officielle.

Exemple :

```
<...>
etcd3:
  hosts:
    - 10.20.89.56:2379
    - 10.20.89.57:2379
    - 10.20.89.58:2379
  username: patroniaccess
  password: secret
<...>
```

### 5.6.10.6 Configuration de PostgreSQL



- postgresql
- postgresql.authentication
- postgresql.callbacks
- postgresql.parameters
- postgresql.pg\_hba
- postgresql.pg\_ident
- postgresql.create\_replica\_methods

La section `postgresql` du fichier de configuration permet de faire la configuration de l'instance PostgreSQL.

Les paramètres suivants sont disponibles à ce niveau :

- `connect_address`: adresse IP au travers de laquelle PostgreSQL est accessible pour les autres nœuds et applications sous la forme [IP|nom d'hôte]:port ;
- `data_dir` : l'emplacement du répertoire de données ;
- `config_dir` : l'emplacement des fichiers de configuration (défaut : la valeur de `data_dir`) ;
- `bin_dir` : chemin vers les binaires de PostgreSQL (`pg_ctl`, `pg_rewind`, `pg_basebackup`, `postgres`). Si cette valeur n'est pas configurée, la variable d'environnement PATH sera utilisée pour trouver les exécutables ;
- `listen` : une liste d'adresses sur lesquelles PostgreSQL écoute. Elles doivent être accessibles par les autres nœuds. Il est possible d'utiliser une liste sous la forme [IP|nom d'hôte],...:port, dans ce cas la première adresse sera utilisée par Patroni pour ses connexions au nœud local. Ce paramètre est utilisé pour valoriser les paramètres `listen_addresses` et `port` de PostgreSQL. Il est possible d'utiliser \* plutôt qu'une liste d'IP ;
- `use_unix_socket` : permet de dire à Patroni de préférer une connexion par socket pour accéder au nœud local (défaut: false) ;
- `use_unix_socket_repl` : permet de dire à Patroni de préférer une connexion par socket pour la réPLICATION (défaut: false) ;
- `pgpass` : chemin vers un fichier `.pgpass`. Il est préférable de laisser Patroni gérer ce fichier et de ne pas ajouter nos propres entrées car elles pourraient être supprimées ;
- `recovery_conf` : permet d'ajouter des paramètres spécifiques lors de la configuration d'un nœud *follower* ;
- `custom_conf` : chemin vers un fichier de configuration qui sera utilisé à la place de `postgresql.base.conf`. Le fichier doit exister et être accessible par Patroni et PostgreSQL. Patroni ne surveille pas ce fichier pour détecter des modifications et ne le sauvegarde pas. Il est simplement inclus dans le `postgresql.conf` ;
- `pg_ctl_timeout` : temps d'attente pour les actions effectuées avec `pg_ctl` (start, stop, restart) (défaut: 60) ;

- `use_pg_rewind` : utilise `pg_rewind` pour permettre à un ancien *leader* de rejoindre le cluster comme réplica après un *failover* (défaut: `false`) ;
- `remove_data_directory_on_rewind_failure` : force la suppression du répertoire de donnée de l'instance en cas d'erreur lors du `pg_rewind` (défaut: `false`) ;
- `remove_data_directory_on_diverged_timelines` : supprime le répertoire de données si les timelines divergent entre le réplica et son leader ;
- `pre_promote` : permet d'exécuter un script destiné à faire du *fencing*. Il est exécuté pendant un failover après l'acquisition du leader lock et avant la promotion d'un réplica. Si le script renvoie un code différent de zéro, Patroni n'effectue pas la promotion et relâche la leader key.

Exemple de configuration basique :

```
<...>
postgresql:
  listen: "*:5432"
  connect_address: 10.20.89.54:5432
  data_dir: /var/lib/pgsql/11/data
  bin_dir: /usr/pgsql-11/bin
  pgpass: /var/lib/pgsql/.pgpass_patroni
<...>
```

La première sous-section de cette partie de la configuration permet de configurer les utilisateurs utilisés par Patroni pour ses tâches courantes (`superuser`), de configurer la réPLICATION (`replication`) ou d'exécuter un `pg_rewind` (`rewind`).

Pour chaque type d'utilisateur, les éléments de configuration suivants sont disponibles et correspondent aux paramètres de connexion éponymes : `username`, `password`, `sslmode`, `sslkey`, `sslpASSWORD`, `sslcert`, `sslrootcert`, `sslCrl`, `sslCrlDir`, `gssencmode` et `channel_binding`.

Exemple de section `authentification` :

```
<...>
postgresql:
  <...>
    authentification:
      superuser:
        username: patronidba
        password: secret
      replication:
        username: replicator
        password: secretaussi
      rewind:
        username: rewinder
        password: sectrettoujours
  <...>
```

Patroni permet de définir des *callbacks* pour exécuter des scripts lors du changement d'état du cluster. Trois paramètres sont communiqués aux scripts : l'action, le rôle du nœud et le nom du cluster.

Les actions suivantes sont disponibles et sont regroupées dans la section `postgresql.callbacks`:

- `on_reload` : rechargement de la configuration ;

- `on_restart`: redémarrage de PostgreSQL ;
- `on_role_change` : promotion ou passage de primaire à standby ;
- `on_start`: démarrage de PostgreSQL ;
- `on_stop`: arrêt de PostgreSQL.

Les sous-sections suivantes sont aussi disponibles dans cette section et ont un fonctionnement identique à ce qui a été décrit précédemment :

- `parameters` : paramètres de configuration ;
- `pg_hba` : liste des règles d'authentification par hôtes ;
- `pg_ident` : liste des équivalences entre utilisateur OS et PostgreSQL.

Pour finir, la section `create_replica_method` permet de définir la façon dont les instances secondaires sont créées<sup>20</sup>. Elle contient une liste de toutes les méthodes qui seront testées dans l'ordre d'apparition. Patroni permet l'utilisation de `pg_basebackup` (défaut), `pgBackRest`, `WAL-G`, `barman` ou d'un script utilisateur pour réaliser cette opération.

Il est possible de fournir des paramètres à l'outil que l'on souhaite utiliser. Tous les paramètres configurés seront communiqués sous la forme `--<nom>=<valeur>` ou `--<nom>`, à l'exception des 3 paramètres suivants :

- `no_master` : permet d'utiliser une méthode même si aucune instance n'est démarrée dans l'agrégat (défaut : `false`) ;
- `no_params` : permet de ne pas utiliser les paramètres supplémentaires décrits ci-après (défaut : `false`) ;
- `keep_data` : permet de dire à Patroni de ne pas vider le répertoire de données de l'instance avant la réinitialisation (défaut : `false`).

Les paramètres suivant seront également fourni au script si l'option `no_params` reste à `False` :

- `scope` : nom de l'agrégat ;
- `datadir` : chemin vers le répertoire de donnée de l'instance secondaire ;
- `role` : toujours valorisé à `replica` ;
- `connstring` : chaîne de connexion vers le nœud depuis lequel la copie va être réalisée.

Exemple de configuration des méthodes de création des réplicas :

```
<...>
postgresql:
  <...>
  create_replica_methods:
    - pgbackrest
    - basebackup
  pgbackrest:
    command: /usr/bin/pgbackrest --stanza=patroni_demo --delta restore
    keep_data: True
    no_master: True
    no_params: True
  basebackup:
    keep_data: False
```

<sup>20</sup>[https://patroni.readthedocs.io/en/latest/replica\\_bootstrap.html#custom-replica-creation](https://patroni.readthedocs.io/en/latest/replica_bootstrap.html#custom-replica-creation)

```
no_master: False
no_params: False
verbose
max-rate: '100M'
waldir: /pg_wal/14/pg_wal
<...>
```

### 5.6.10.7 Configuration de l'API REST



- restapi
  - connect\_address
  - listen
  - authentication(username/password)
  - SSL (certfile, keyfile, keyfile\_password, cafile, verify\_client)

L'accès à l'API REST peut être contrôlé grâce aux paramètres suivants de la section `restapi` :

- `listen` : permet de définir les adresses et le port sur lesquelles Patroni écoute pour son API REST. Elle est notamment utilisée par les autres membres de l'agrégat afin de pouvoir vérifier la santé du nœud lors d'une élection. Cette adresse peut également servir pour les *health checks* d'outils comme HAProxy, la supervision et les connexions utilisateurs.
- `connect_address` : adresse IP et port fournis aux autres membres pour l'accès à l'API REST de Patroni. Cette information est stockée dans le DCS.
- 'proxy\_address' : adresse et port pour joindre le pool de connexion ou proxy qui permet d'accéder à PostgreSQL. Une entrée `proxy_url` est créée dans etcd afin de faciliter la découverte de service ;
- `authentication` : permet de définir un `username` et `password` pour le accès au endpoint de l'API REST.
- `certfile` : certificat au format PEM (active le SSL si présent) ;
- `keyfile` : clé secrète au format PEM ;
- `keyfile_password` : mot de passe pour déchiffrer la clé ;
- `cafile` : spécifie le certificat de l'autorité de certification ;
- `verify_client` : définit quand la clé est requise
  - `none` (défaut) : l'API REST ne vérifie pas les certificats ;
  - `requiered` : les certificats clients sont requis pour tous les accès à l'API REST ;
  - `optional` : les certificats ne sont requis que pour les accès marqués comme sensibles (appels PUT, POST, PATCH et DELETE).
- `allowlist` : liste d'hôtes autorisés à accéder aux endpoints définis comme sensibles. Les noms d'hôtes, adresses IP ou adresse de réseau sont autorisés. Par défaut tout est autorisé ;
- `allowlist_include_members` : permet d'autoriser les membres de l'agrégat à accéder aux API sensibles. L'adresse IP est récupérée à partir du paramètre `api_url` stocké dans le DCS : attention à ce que ce soit bien l'IP utilisée pour accéder à l'API REST !

Il existe également du paramétrage pour adapter les en-têtes HTTP ou HTTPS.

Voici un exemple qui autorise les accès aux endpoints sensibles uniquement depuis les membres de l'agrégat :

```
name: p1
scope: patroni-demo
<...>
restapi:
  listen: 0.0.0.0:8009
  connect_address: 10.20.89.54:8009
  allowlist_include_members: true
<...>
```

On peut voir que l'adresse définie dans `connect_address` est reportée dans le DCS sous le nom `api_url`.

```
[root@e1 ~]# ETCDCTL_API=3 etcdctl get
> --print-value-only '/service/patroni-demo/members/p1' | python -m json.tool

{
  "api_url": "http://10.20.89.54:8008/patroni",
  "conn_url": "postgres://10.20.89.54:5432/postgres",
  "pending_restart": true,
  "role": "master",
  "state": "running",
  "timeline": 1,
  "version": "2.1.4",
  "xlog_location": 50331968
}
```

Si on lance la commande suivante de modification de la configuration depuis le serveur **p2** vers l'API REST du serveur **p1**, la modification aboutit :

```
[root@p2 ~]# curl -s -XPATCH -d '{
>   "loop_wait": 10,
>   "master_start_timeout": 300,
>   "postgresql": {
>     "parameters": {
>       "archive_command": "/bin/true",
>       "archive_mode": "on",
>       "max_connections": 101
>     },
>     "use_pg_rewind": false,
>     "use_slot": true
>   },
>   "retry_timeout": 10,
>   "ttl": 30
> }' 10.20.89.54:8008/config | python -m json.tool

{
  "loop_wait": 10,
  "master_start_timeout": 300,
  "postgresql": {
    "parameters": {
      "archive_command": "/bin/true",
      "archive_mode": "on",
      "max_connections": 101
    },
    "use_pg_rewind": false,
    "use_slot": true
  }
}
```

```
 },
"retry_timeout": 10,
"ttl": 30
}
```

La même commande lancée depuis un autre serveur se termine avec le message suivant :

Access is denied

### 5.6.10.8 Configuration des accès à l'API REST par patronictl



- `ctl`
  - `insecure`
  - `certfile`
  - `keyfile`
  - `keyfile_password`
  - `cacert`

La sécurisation des accès à l'API REST par patronictl est également possible grâce aux paramètres suivants dans la section `ctl` :

- `insecure` : autorise les connexions l'API REST sans vérification des certificats SSL ;
- `certfile` : certificat au format PEM (active le SSL si présent) ;
- `keyfile` : clé secrète au format PEM ;
- `keyfile_password` : mot de passe pour déchiffrer la clé ;
- `cacert` : certificat d'autorité de certification.

### 5.6.10.9 Configuration du watchdog



- watchdog
  - mode : required
  - device
  - safety\_margin

Afin d'éviter une situation de *split brain*, Patroni doit s'assurer que l'instance PostgreSQL d'un nœud n'accepte plus de transaction une fois que la *leader key* qui lui est associées expire. En temps normal, Patroni essaie d'obtenir cette garantie en arrêtant l'instance. Cependant cette opération peut échouer si :

- Patroni plante à cause d'un bug, un problème de mémoire ou le processus est tué par une source extérieure ;
- l'arrêt de PostgreSQL est trop lent ;
- Patroni ne fonctionne pas à cause d'une charge trop importante sur le serveur, ou un autre problème d'infrastructure ou d'hyperviseur.

Afin que le cluster réagisse correctement dans ce genre de cas, Patroni supporte l'utilisation d'un watchdog. Un watchdog est un composant doté d'un compte à rebours qui s'il expire éteint ou redémarre physiquement le serveur *sur-le-champs*. Un logiciel, ici Patroni, doit donc recharger continuellement ce *watchdog timeout (WDT)* avant qu'il n'expire.

#### Activation et Désactivation

Patroni tente d'armer le watchdog sur le nœud qui devient leader avant la promotion de l'instance PostgreSQL. Si l'utilisation d'un watchdog est requise (voir `watchdog.mode`) et que le watchdog ne s'active pas, le nœud refuse de devenir leader.

Un test est également réalisé lorsqu'un nœud décide de participer à l'élection du primaire et que le watchdog est requis sur ce dernier. Dans ce cas Patroni vérifie que le *device* associé au watchdog existe (voir `watchdog.device`) et est accessible. Il contrôle également que le timeout du watchdog est supérieur ou égal à la durée nominale d'une boucle (`loop_wait`).

Lorsqu'une instance perd le statut de leader ou que Patroni est mis en pause, le watchdog est désactivé.

#### Mécanique et paramétrage

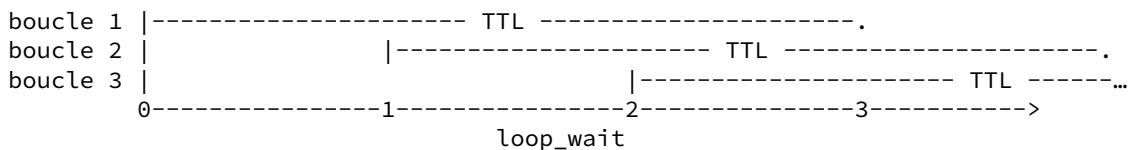
Par défaut Patroni configure le watchdog pour expirer 5 secondes avant que le `ttl` n'expire, c'est le paramètre `safety_margin`. Patroni calcule donc le *watchdog timeout* grâce à la formule suivante:  $WDT = ttl - safety\_margin$ .

Ces 5 secondes laissent une marge de sécurité avant que le *leader key* n'expire. Elles permettent de garantir que l'ancien primaire est bien arrêté au moment où le `ttl` expire, ce qui déclenche alors une

nouvelle élection. Nous évitons ainsi une situation de *split brain* en cas d'incident ou de blocage.

Pour bien comprendre comment configurer `ttl`, `safety_margin`, `loop_wait` et `retry_timeout`, intéressons nous au fonctionnement interne de Patroni.

La boucle de haute disponibilité est exécutée au moins toutes les 10 secondes par défaut, c'est le paramètre `loop_wait`. À la fin de chaque exécution, le processus calcule combien de temps il doit patienter avant sa prochaine exécution pour respecter au mieux cette période de `loop_wait` secondes. Dans les cas extrêmes (charge, lenteur, etc), il se ré-exécute sur-le-champs pour rattraper son retard. Le `ttl` étant de 30s, Patroni a l'équivalent de trois exécutions de boucles pour le recharger.

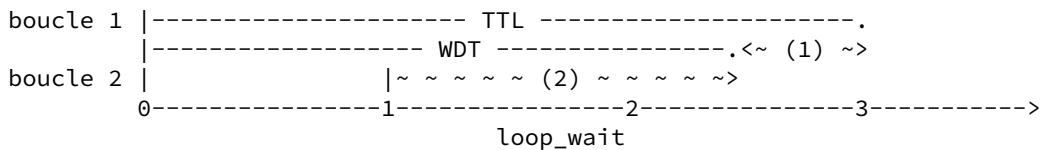


À chaque exécution de la boucle, après avoir déterminé que l'instance est primaire, Patroni doit:

1. recharger/valider le TTL du *leader key*
2. en cas de succès, recharger immédiatement le *watchdog timeout*
3. en cas d'échec, immédiatement déclasser l'instance en *standby*

En temps normal, le temps écoulé entre les actions 1 et 2 est négligeable. Le watchdog expire donc environ `safety_margin` secondes *avant* le TTL de la *leader key*, ce qui est désiré.

Il faut aussi tenir compte qu'au début d'une boucle, avant que toute action n'ai eu lieu, le WDT a dans le meilleur des cas été recharge il y a déjà `loop_wait` secondes, lors de l'exécution de la précédente boucle. Par défaut, la boucle a donc 15 secondes pour effectuer ses toutes premières actions.



$$(1) \approx \text{safety\_margin}$$

$$(2) \approx \text{TTL} - \text{loop\_wait} - \text{safety\_margin} = 30 - 10 - 5 = 15\text{s}$$

Ces différents paramètres et calculs en tête, intéressons-nous maintenant à deux scénarios d'incidents.

Le premier cas concerne une coupure de service avec le DCS. Dans cette situation, nous devons ici considérer le paramètre `retry_timeout` qui, dans le cas du leader, exprime le temps d'attente maximal d'une réponse du DCS avant de le considérer comme perdu. Par défaut, cette attente est de 10 secondes. Au moment où Patroni tente de mettre à jour le TTL de son *leader lock*, sans réponse du DCS après ces 10 secondes, Patroni déclenche une opération de *demote* pour déclasser l'instance en mode *standby* sur-le-champs. Notez qu'il ne recharge alors **pas** le watchdog, la situation du leader lock étant indéfinie. En conséquence, à ce moment précis, avec le paramétrage par défaut, Patroni n'a plus que 5 secondes pour effectuer l'opération de *demote*:  $\text{WDT} - \text{loop\_wait} - \text{retry\_timeout} = 25 - 10 - 10 = 5\text{s}$

```
boucle 1 |----- TTL -----.
          |----- WDT -----.
boucle 2 |           | ~ ~ ~ (1) ~ ~ ~ ><(2)>
          0-----1-----2-----3----->
                           loop_wait

(1) retry_timeout
(2) temps de demote = WDT - loop_wait - retry_timeout = 25 - 10 - 10 = 5s
```

Si l'instance ne s'arrêtait pas proprement dans les temps, l'arrêt brutal par le watchdog est de toute façon équivalent au kill -SIGKILL effectué par Patroni dans ce genre de situation lorsque la réPLICATION synchrone est activée (voir paramètre master\_stop\_timeout).

Le second scénario implique une réponse du DCS anormalement longue (charge, coupure réseau, etc) ou un incident gelant la machine entre les deux recharges des TTL et WDT. Or, si le watchdog est rechargeé plus que safety\_margin secondes après le TTL, alors le WDT expire malheureusement **après** le prochain TTL.

```
boucle 1 |----- TTL -----.
          |<~ ~ (1) ~ ~>----- WDT -----.
          |                               <(2)>
          0-----1-----2-----3----->
                           loop_wait

(1) gel > safety_margin
(2) gel + WDT > TTL, le watchdog expire après le TTL
```

Ce cas est relativement peu probable, mais possible. De plus, il peut très bien survenir sans aucune conséquence pour le cluster. Rappelez-vous que la boucle est de 10 secondes par défaut, les TTL et WDT pourraient très bien être rechargeés correctement à la prochaine boucle, longtemps avant leurs expirations respectives. Mais dans le pire des cas, la charge du DCS pourrait par exemple être continue reproduisant ainsi cet événement systématiquement, rendant le watchdog inefficace pour protéger votre cluster d'un *split brain*.

Si l'on ne souhaite pas courir ce risque, il est possible de réduire le watchdog timeout. En contrepartie, il faut alors soit augmenter ttl, soit diminuer loop\_wait et/ou retry\_timeout. Pour illustrer cela, avec safety\_margin = -1 = ttl / 2 (valeur du WDT dans les anciennes versions de Patroni avant que safety\_margin n'apparaisse), si nous ne modifions par ces autres paramètres, avec les calculs expliqués précédemment nous obtenons alors:

- WDT = ttl / 2 = 30 / 2 = 15 secondes
- le temps disponible au début d'une boucle pour réaliser l'ensemble de toutes ses actions, y compris recharger les TTL et WDT n'est plus que de 5 secondes: WDT - loop\_wait = 15 - 10 = 5 secondes
- en cas de ralentissement du DCS, le watchdog peut se déclencher avant même d'avoir eu la validation du recharge du TTL: WDT - loop\_wait - retry\_timeout = 15 - 10 - 10 = -5 secondes

Avec une telle configuration, le cluster est donc beaucoup plus sensible. De plus :

- augmenter ttl : retarde le déclenchement du failover ;

- diminuer `loop_wait` : augmente la consommation de ressource de Patroni ainsi que le nombre d'accès au DCS. Cela rend donc l'architecture plus sensible à tous types de ralentissements ;
- diminuer `retry_timeout` : rends le système plus sensible aux problèmes réseaux ou charge du DCS.

Quoi qu'il en soit, en cas de gêle du serveur PostgreSQL, de charge importante sur les DCS ou les instances, ou encore d'incident réseau, le bon correctif reste de régler le problème à la racine.

Voici un résumé des paramètres de la section `watchdog` qui permettent de configurer le `watchdog` :

- `mode` : le `watchdog` peut être désactivé (`off`), être activé si c'est possible (`automatic`) ou être obligatoire (`required`). Dans ce dernier mode, si le `watchdog` ne peut pas s'activer, le nœud ne pourra pas devenir `leader` (défaut : `automatic`) ;
- `device` : chemin vers le `watchdog` (défaut : `/dev/watchdog`) ;
- `safety_margin` : marge de sécurité entre le déclenchement du `watchdog` et l'expiration de la `leader key` (défaut: 5).

### 5.6.10.10 Configuration des tags



- tags
  - nofailover
  - clonefrom
  - noloadbalance
  - replicatefrom
  - nosync

Patroni permet de configurer des tags pour adapter le fonctionnement des nœuds dans la section tags du fichier YAML :

- nofailover : permet d'interdire la promotion du nœud (défaut : false) ;
- clonefrom : permet de définir un nœud comme source privilégiée pour l'initialisation des nœuds (pg\_basebackup). Si plusieurs nœuds sont dans ce cas, le nœud sera choisi au hasard (défaut : false) ;
- noloadbalance : lorsque ce tag est à true le nœud renvoie le code HTTP 503 pour l'accès au endpoint GET /replica ce qui l'exclura du load balancing (défaut : false) ;
- replicatefrom : l'adresse IP d'un autre réplica utilisé pour faire de la réPLICATION en cascade ;
- nosync : le nœud ne sera pas sélectionné comme nœud synchrone (défaut : true).

Il est possible d'ajouter des tags spécifiques.

Voici un exemple de configuration :

```
<...>
tags:
  noloadbalance: true
  montag: "mon tag a moi"
```

Les tags sont visibles depuis patronictl list :

```
$ patronictl -c /etc/patroni/demo.yaml list
```

Member	Host	Role	State	TL	Lag in MB	Tags
<i>+ Cluster: patroni-demo (7147602572400925478)</i>						
p1	10.20.89.54	Leader	running	1		montag: mon tag a moi noloadbalance: true
p2	10.20.89.55	Replica	running	1	0	
<i>+-----+</i>						

### 5.6.10.11 Configuration d'un agrégat multi-nodes CITUS



- Permet de simplifier le déploiement d'un cluster Citus
- Paramètres :
- group
- database

Patroni permet de déployer un cluster multi-nodes CITUS<sup>21</sup>. `patronictl` a également été adapté pour afficher les informations sur les groupes de serveurs Citus.

---

<sup>21</sup>[https://docs.citusdata.com/en/stable/installation/multi\\_node.html](https://docs.citusdata.com/en/stable/installation/multi_node.html)

### 5.6.10.12 Variables d'environnement



- Toutes les options précédentes
- PATRONICCTL\_CONFIG\_FILE
- PATRONI\_SCOPE

Il est possible d'utiliser des variables d'environnement<sup>22</sup> pour configurer la plupart des éléments présentés précédemment. On choisit cependant généralement d'utiliser le fichier de configuration YAML pour cela.

Certaines variables sont cependant utiles au quotidien et méritent d'être chargées au démarrage de la session de l'utilisateur destiné à manipuler patronictl.

- PATRONICCTL\_CONFIG\_FILE permet de spécifier l'emplacement du fichier de configuration ce qui évite de spécifier l'option -c à chaque fois.
- PATRONI\_SCOPE permet de spécifier le nom de l'agrégat ce qui évite la plupart du temps d'avoir à saisir le nom de l'agrégat dans les commandes.

---

<sup>22</sup><https://patroni.readthedocs.io/en/latest/ENVIRONMENT.html#environment-configuration-settings>

### 5.6.11 patronictl



- Permet l'interaction avec l'agrégat
- Depuis n'importe quel nœud

**patronictl** permet d'interagir avec l'agrégat pour modifier son comportement ou consulter son état. On peut l'utiliser depuis l'utilisateur système **postgres**.



On peut indiquer l'emplacement du fichier de configuration dans la variable **PATRONICCTL\_CONFIG\_FILE** et ainsi s'affranchir de l'option **-c** (**--config-file**) de la commande **patronictl**:

```
$ export PATRONICCTL_CONFIG_FILE=/etc/patroni/config.yml
$ patronictl topology

+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host           | Role        | State     | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1  | 10.0.3.85:5434 | Leader      | running   | 122 |             |
| + pg-2 | 10.0.3.35:5434 | Sync Standby | running   | 122 | 0           |
| + pg-3 | 10.0.3.70:5434 | Sync Standby | running   | 122 | 0           |
+-----+-----+-----+-----+
```

La liste complète des commandes est disponible par l'option **--help**:

```
$ patronictl --help

Usage: patronictl [OPTIONS] COMMAND [ARGS]...

Options:
  -c, --config-file TEXT  Configuration file
  -d, --dcs TEXT          Use this DCS
  -k, --insecure           Allow connections to SSL sites without certs
  --help                   Show this message and exit.

Commands:
  configure    Create configuration file
  dsn          Generate a dsn for the provided member, defaults to a dsn of...
  edit-config   Edit cluster configuration
  failover     Failover to a replica
  flush         Discard scheduled events
  history       Show the history of failovers/switchovers
  list          List the Patroni members for a given Patroni
  pause         Disable auto failover
  query         Query a Patroni PostgreSQL member
  reinit        Reinitialize cluster member
```

```

reload      Reload cluster member configuration
remove      Remove cluster from DCS
restart     Restart cluster member
resume      Resume auto failover
scaffold    Create a structure for the cluster in DCS
show-config Show cluster configuration
switchover   Switchover to a replica
topology    Prints ASCII topology for given cluster
version     Output version of patronictl command or a running Patroni...

```

### 5.6.11.1 Consultation d'état



- **list** : lister les membres d'un agrégat par ordre alphabétique
- **topology** : afficher la topologie d'un agrégat

```
$ patronictl list

+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host           | Role        | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1  | 10.0.3.85:5434 | Sync Standby | running    | 123 |          0 |
| pg-2  | 10.0.3.35:5434 | Leader       | running    | 123 |          |
| pg-3  | 10.0.3.70:5434 | Sync Standby | running    | 123 |          0 |
+-----+-----+-----+-----+-----+
```

La commande **topology** affiche la liste des nœuds sous la forme d'un arbre débutant par le primaire courant, suivi des nœuds secondaires.

```
$ patronictl topology

+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host           | Role        | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-2  | 10.0.3.35:5434 | Leader       | running    | 123 |          |
| + pg-1 | 10.0.3.85:5434 | Sync Standby | running    | 123 |          0 |
| + pg-3 | 10.0.3.70:5434 | Sync Standby | running    | 123 |          0 |
+-----+-----+-----+-----+-----+
```



- **show-config** : montrer la configuration dynamique (stockée dans le DCS)

La configuration commune des instances peut être affichée depuis n'importe quel nœud de l'agrégat :

```
$ patronictl show-config
```

```

loop_wait: 10
maximum_lag_on_failover: 1048576
postgresql:
  parameters:
    archive_command: pgbackrest --stanza=main archive-push %p
    archive_mode: 'on'
    checkpoint_timeout: 15min
    hot_standby: 'on'
    log_min_duration_statement: -1
    max_replication_slots: 5
    max_wal_senders: 5
    use_pg_rewind: true
    use_slots: true
    wal_keep_segment: 2
    wal_level: replica
  retry_timeout: 10
  synchronous_mode: true
  synchronous_node_count: 2
  ttl: 30

```

### 5.6.11.2 Commandes de modification d'état



- **edit-config** : Éditer la configuration dynamique (stockée dans le DCS) de l'agrégat
  - ni fichier de conf, ni ALTER SYSTEM !
  - Si redémarrage : à demander explicitement

Cette commande lance l'éditeur par défaut du système pour permettre d'appliquer une modification à la configuration de l'agrégat. Elle nécessite la commande less lors de la validation finale et peut être lancée sur n'importe quel nœud de l'agrégat.

Il ne faut pas modifier postgresql.conf directement, mais utiliser edit-config pour adapter le fichier temporaire en YAML indiqué, par exemple celui-ci :

```
$ patronictl -c /etc/patroni/patroni.yml edit-config
```

```

loop_wait: 10
maximum_lag_on_failover: 1048576
postgresql:
  parameters:
    hot_standby: true
    max_replication_slots: 5
    max_wal_senders: 10
    shared_buffers: 64MB
    wal_level: replica
    work_mem: 70MB

```

```
use_pg_rewind: true
use_slots: true
retry_timeout: 10
ttl: 30
```

À l'issue, la configuration sera écrite sur le DCS puis appliquée de manière asynchrone par Patroni, sur chacun des nœuds concernés, si celle-ci ne nécessite pas de redémarrage. Dans le cas contraire, le nœud est marqué comme *pending restart* et devra être redémarré manuellement avec la commande `patronictl restart`.



Utiliser `ALTER SYSTEM` ou modifier manuellement les fichiers du PGDATA peut mener à des nœuds utilisant des configurations différentes !

En pratique, la configuration de PostgreSQL est stockée par Patroni dans le DCS qui fait référence :

```
# ETCDCTL_API=2 etcdctl -u root get /service/cluster-test-01/config
```

puis dans un fichier `postgresql.conf` sur les nœuds.

Le `pg_hba.conf` se modifie là aussi (bien reprendre toute sa configuration la première fois) :

```
$ patronictl -c /etc/patroni/patroni.yml edit-config

+++
@0 -10,5 +10,7 @@
    work_mem: 50MB
    use_pg_rewind: true
    use_slots: true
+ pg_hba:
+ - host user erp 192.168.99.0/24 md5
+ - host all all all scram-sha-256
+ - host replication replicator all scram-sha-256
    retry_timeout: 10
    ttl: 30
```

```
Apply these changes? [y/N]: y
Configuration changed
```

### 5.6.11.3 Commandes de bascule



- **failover** : bascule par défaillance du primaire
- **switchover** : promotion d'un secondaire

La commande `failover` permet de déclencher une bascule en déclarant défaillant le primaire courant. C'est une bonne manière de valider qu'un secondaire est prêt à devenir primaire et que les secondaires sont capables de se raccrocher à lui une fois sa promotion effectuée.

Contrairement à la commande `switchover`, la commande `failover` ne nécessite pas de spécifier la localisation de l'instance primaire. Cela permet de promouvoir une instance lorsque toutes les instances disponibles sont au statut `replica`. Ce genre de cas peut se présenter lors de certaines opérations de restauration.

```
$ patronictl failover

Candidate ['pg-1', 'pg-3'] []: pg-1
Current cluster topology
+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1 | 10.0.3.85:5434 | Sync Standby | running | 123 | 0 |
| pg-2 | 10.0.3.35:5434 | Leader | running | 123 | |
| pg-3 | 10.0.3.70:5434 | Sync Standby | running | 123 | 0 |
+-----+-----+-----+-----+
Are you sure you want to failover cluster my-ha-cluster, demoting current leader
↪ pg-2?
[y/N]: y

2021-05-28 13:46:47.41163 Successfully failed over to "pg-1"
+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1 | 10.0.3.85:5434 | Leader | running | 123 | |
| pg-2 | 10.0.3.35:5434 | Replica | stopped | | unknown |
| pg-3 | 10.0.3.70:5434 | Replica | running | 123 | 16 |
+-----+-----+-----+-----+
```

Exemple avec `switchover`:

```
$ patronictl switchover

Primary [pg-1]:
Candidate ['pg-2', 'pg-3'] []: pg-2
When should the switchover take place (e.g. 2021-05-28T14:48 ) [now]:
Current cluster topology
+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1 | 10.0.3.85:5434 | Leader | running | 124 | |
| pg-2 | 10.0.3.35:5434 | Sync Standby | running | 124 | 0 |
| pg-3 | 10.0.3.70:5434 | Sync Standby | running | 124 | 0 |
+-----+-----+-----+-----+
Are you sure you want to switchover cluster my-ha-cluster, demoting current leader
↪ pg-1?
[y/N]: y

2021-05-28 13:48:45.22331 Successfully switched over to "pg-2"
+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+
```

pg-1   10.0.3.85:5434   Replica   stopped   124   unknown
pg-2   10.0.3.35:5434   Leader   running   124   16
+-----+-----+-----+-----+-----+-----+

On peut également forcer une bascule de manière non interactive en spécifiant le primaire courant et le nœud secondaire cible :

```
$ patronictl switchover --leader pg-2 --candidate pg-1 --force
```

```
Current cluster topology
+ Cluster: my-ha-cluster (6876375338380834518) -----+-----+
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| pg-1 | 10.0.3.85:5434 | Sync Standby | running | 125 | 0 |
| pg-2 | 10.0.3.35:5434 | Leader | running | 125 | |
| pg-3 | 10.0.3.70:5434 | Sync Standby | running | 125 | 0 |
+-----+-----+-----+-----+-----+-----+
2021-05-28 14:03:28.68678 Successfully switched over to "pg-1"
+ Cluster: my-ha-cluster (6876375338380834518) -----+-----+
| Member | Host | Role | State | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| pg-1 | 10.0.3.85:5434 | Leader | running | 125 | |
| pg-2 | 10.0.3.35:5434 | Replica | stopped | | unknown |
| pg-3 | 10.0.3.70:5434 | Replica | running | 125 | 16 |
+-----+-----+-----+-----+-----+-----+
```

L'option `--force` permet de désactiver la confirmation avant l'exécution de l'opération.



Au passage, on remarque qu'après un *switchover* ou un *failover*, des secondaires configurés pour être synchrones, ne le sont plus pendant un laps de temps.

#### 5.6.11.4 Contrôle de la bascule automatique



- **pause / resume**
- **history**
- **reinit**

##### Maintenance :

La commande `patronictl pause` met le cluster en mode maintenance. Cela a pour effet de « détacher Patroni de l'instance qu'il manage. Cela a plusieurs effets sur le comportement du système :

- le mécanisme de promotion automatique est désactivé ;
- le redémarrage du service Patroni ne provoquera plus le redémarrage de l'instance associée ;

- il n'est pas possible de faire un *switchover* sans préciser de cible.

On l'utilise généralement lorsqu'une anomalie conduit à une avalanche de bascules non désirées ou lorsque l'on doit exécuter des opérations de maintenance sur le nœud qui entreraient en conflit avec Patroni.

L'option `--wait` permet de s'assurer que la commande a été prise en compte par tous les nœuds. On peut observer son effet avec la commande `patronictl list` qui affiche que le mode maintenance est activé :

La commande `patronictl resume` permet de désactiver le mode maintenance lorsque celui-ci a été désactivé auparavant. L'option `--wait` permet de s'assurer que la commande a été prise en compte par tous les nœuds.

#### **Historique :**

L'historique des bascules est disponible via la commande `patronictl history` :

```
$ patronictl history
```

TL	LSN	Reason	Timestamp
1	25577936	no recovery target specified	
2	83886528	no recovery target specified	
3	83887160	no recovery target specified	
...			
122	4445962400	no recovery target specified	2021-05-28T13:41:57.231514+00:00
123	4462739616	no recovery target specified	2021-05-28T13:46:47.366787+00:00
124	4479516832	no recovery target specified	2021-05-28T13:48:44.616172+00:00

#### **Réinitialisation :**

`patronictl reinit` réinitialise un nœud.



Toutes les données du nœud sont détruites, écrasées par celles du primaire !

```
postgres@pg-1:~$ patronictl reinit my-ha-cluster pg-2
```

```
+ Cluster: my-ha-cluster (6876375338380834518) +-----+
| Member | Host      | Role   | State  | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| pg-1   | 10.0.3.201 | Leader | running | 8  |           |
| pg-2   | 10.0.3.202 | Replica | running | 8  |          0 |
+-----+-----+-----+-----+-----+
```

```
Are you sure you want to reinitialize members pg-2? [y/N]: y
Success: reinitialize for member pg-2
```



Il est impossible de lancer une réinitialisation du nœud primaire puisqu'elle est faite à partir du primaire courant.

```
postgres@pg-1:~$ patronictl reinit my-ha-cluster pg-1
+ Cluster: 13-main (6876375338380834518) -----
| Member | Host      | Role     | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1  | 10.0.3.201 | Leader   | running | 8  |           |
| pg-2  | 10.0.3.202 | Replica  | running | 8  |          0 |
+-----+-----+-----+-----+
Which member do you want to reinitialize [pg-1, pg-2]? []: pg-1
Are you sure you want to reinitialize members pg-1? [y/N]: y
Failed: reinitialize for member pg-1, status code=503, (I am the leader,
can not reinitialize)
```

### 5.6.11.5 Changements de configuration



- **reload**
  - attention aux *pending restart*
- **restart**

#### Changement de configuration :

`patronictl reload` recharge la configuration de tous les nœuds de l'agrégat ou d'un nœud s'il est spécifié.

```
$ patronictl reload my-ha-cluster pg-1
+ Cluster: my-ha-cluster (6876375338380834518) -----
| Member | Host      | Role     | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1  | 10.0.3.85:5434 | Leader   | running | 122 |           |
| pg-2  | 10.0.3.35:5434 | Sync Standby | running | 122 |          0 |
| pg-3  | 10.0.3.70:5434 | Sync Standby | running | 122 |          0 |
+-----+-----+-----+-----+
Are you sure you want to reload members pg-1? [y/N]: y
Reload request received for member pg-1 and will be processed within 10 seconds
```

Les traces montrent l'opération :

```
août 04 15:51:40 pg-2 patroni@my-ha-cluster[21335]: 2021-08-04 15:51:40,473
INFO: Reloading PostgreSQL configuration.
août 04 15:51:40 pg-2 patroni@my-ha-cluster[21335]: envoi d'un signal au serveur
août 04 15:51:41 pg-2 patroni@my-ha-cluster[21335]: 2021-08-04 15:51:41,762
INFO: Lock owner: pg-1; I am pg-2
```

Si les modifications nécessitent un redémarrage de PostgreSQL, les serveurs dont la configuration a été modifiée seront marqué avec la mention `pending restart` dans `patronictl list`. Il faudra les redémarrer pour que le changement de configuration soit bien pris en compte.

**Redémarrage :**

`patronictl restart` redémarre le nœud spécifié ou l'agrégat entier en commençant par le primaire, suivi de ses secondaires.

Elle ne provoque pas de changement de rôle si l'opération se passe bien. Il est possible de redémarrer uniquement les serveurs en mode *pending restart* grâce à l'option `--pending`.

### 5.6.12 endpoints de l'API REST



- Contrôle du rôle du serveur
- Informations sur l'état d'un nœud ou du cluster
- Manipulation du cluster

L'API REST de Patroni permet de confirmer le rôle d'un serveur grâce à une requête http. Différents *endpoint* existent :

- `/primary`, `/leader` et `/standby-leader`
- `/replica`
- `/asynchronous`
- `/synchronous`
- `/read-only` et `/read-only-sync`

Il est possible d'enrichir la requête API en filtrant sur le lag de réPLICATION. On peut aussi vérifier la présence ou la valeur d'un tag dans la configuration d'un nœud. Cette option n'est pas possible pour les *endpoints* : `/primary`, `/leader` et `/standby-leader`.

Exemples d'utilisations :

```
$ curl -I -s http://10.20.61.103:8008/replica?tag_is_candidate=true
HTTP/1.0 200 OK
```

```
$ curl -I -s http://10.20.61.103:8008/replica?tag_is_here=true
HTTP/1.0 503 Service Unavailable
```

```
$ curl -I -s http://10.20.61.103:8008/replica?tag_doesnt_exist=false
HTTP/1.0 503 Service Unavailable
```

```
$ curl -I -s http://10.20.61.103:8008/replica?lag=16MB
HTTP/1.0 200 OK
```

L'API permet aussi d'accéder à différentes informations :

- état d'un nœud : `/patroni,/`;
- état du cluster : `cluster` ;
- disponibilité du service ;

- /health : code HTTP 200 si PostgreSQL fonctionne ;
- /liveness : code HTTP 200 si Patroni fonctionne et que ça boucle de contrôle de la haute disponibilité fonctionne correctement ;
- /readiness : code HTTP 200 si Patroni fonctionne en tant que leader ;
- historique des changements de timeline : /history ;
- configuration dynamique du cluster : /config.

Comme démontré précédemment, la configuration peut également être modifiée avec une requête PATCH ou remplacée avec une requête PUT.

Pour finir, il est possible d'interagir avec le cluster pour réaliser certaines actions de maintenance :

- /switchover, /failover
- /restart, reload, /reinitialize

```
$ curl -s http://10.20.61.103:8008/switchover -XPOST -d \  
> '{"leader":"p2", "candidate":"p1"}'
```

Successfully switched over to "p1"

Les actions switchover et restart peuvent être planifiées :

```
$ curl -s http://10.20.61.103:8008/switchover -XPOST -d \  
> '{"leader":"p1", "candidate":"p3", "scheduled_at":"2023-03-08T11:30+00"}'  
Switchover scheduled  
  
$ curl -s http://10.20.61.103:8009/restart -XPOST -d \  
> '{"schedule":"2023-03-08T11:45+00"}'  
Restart scheduled
```

Seule une opération de chaque type peut être planifiée. Elles peuvent être dé-planifiées avec une requête DELETE.

```
$ curl -s http://10.20.61.103:8008/switchover -XDELETE  
scheduled switchover deleted  
  
$ curl -s http://10.20.61.103:8009/restart -XDELETE  
scheduled restart deleted
```

### 5.6.13 Proxy, VIP et Poolers de connexions



- Connexions aux répliques
- HAProxy
- Keepalived

### 5.6.13.1 Connexions aux répliques



- RéPLICATION asynchrone, synchrone et *remote apply*.
- API REST de Patroni

Les répliques d'un cluster Patroni peuvent être utilisés pour faire de la répartition de charge en lecture (*query off-loading*). Il faut cependant être conscient des limitations associées à cette utilisation des instances secondaires.

Le mode de réPLICATION est contrôlé par les paramètres `synchronous_commit` et `synchronous_standby_names`.

Si la réPLICATION est asynchrone, l'instance secondaire n'est pas nécessairement à jour. La mise en place d'une réPLICATION synchrone ne règle pas ce problème en effet, avec cette configuration, la garantie porte sur l'écriture des données dans les WAL de l'instance secondaire et la demande de synchronisation sur disque. Il est possible d'utiliser le *remote apply* qui garanti que les données sont visibles sur la secondaire avant que la primaire ne rende la main au client. Cependant dans ce cas Les données sont potentiellement visibles sur la secondaire avant la primaire.

Un autre point à prendre en compte est la cohérence des données entre les instances secondaires. Sur ce plan, il n'y a aucune garantie.

L'API REST de Patroni permet de choisir à quel serveur on se connecte. Pour cela, il faut utiliser les `endpoints` `/replica`, `/read-only`, `/asynchronous` ou `/synchronous`. Une requête http renverra le code 200, si le serveur correspond au filtre du `endpoint`.

On peut également éliminer des serveurs en fonction de la valeur d'un tag sur la base du retard de réPLICATION pour les `endpoint` : `/replica`, `/read-only`, `/asynchronous`.

### 5.6.13.2 HAProxy



- Répartiteur de charge (*round-robin*)
- Check http sur l'API REST de Patroni
  - \primary
  - \replica
- Page Web pour les statistiques
- Il faut le mettre en Haute Disponibilité

HAProxy est un répartiteur de charge.

Il peut être installé sur un serveur indépendant dans ce cas il faut penser à sa mise en haute disponibilité pour éviter de créer un *SPOF* dans l'architecture.

HAProxy consomme généralement peu de ressources. Une autre solution consiste à placer HAProxy sur le serveur d'application ou de base de données et coupler la haute disponibilité du répartiteur de charge avec celle de l'application qu'on lui associe.

La documentation de Patroni propose d'utiliser la configuration suivante pour HAProxy.

```
global
    maxconn 100

defaults
    log      global
    mode     tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen stats
    mode http
    bind *:7000
    stats enable
    stats uri /

listen primary
    bind *:5000
    option httpchk HEAD /primary
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 10.20.61.103:5432 maxconn 100 check port 8008
    server node2 10.20.61.104:5432 maxconn 100 check port 8008
    server node3 10.20.61.105:5432 maxconn 100 check port 8008

listen replicas
    bind *:5001
    option httpchk HEAD /replica
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 10.20.61.103:5432 maxconn 100 check port 8008
    server node2 10.20.61.104:5432 maxconn 100 check port 8008
    server node3 10.20.61.105:5432 maxconn 100 check port 8008
```

Elle permet de mettre en place :

- un *endpoint* pour la consultation des statistiques sur le port 7000 ;
- un *endpoint* pour la connexion à l'instance primaire sur le port 5000 ;
- un *endpoint* pour la connexion aux instances secondaires sur le port 5001.

La section `global` définit le nombre de connexion maximal à 100.

La section `default` permet de définir où tracer (paramètre `log`), le type de connexion ici `tcp` (paramètre `mode`), ainsi que les timeouts pour les connexions à HAProxy et à PostgreSQL (paramètres `timeout*`).

La section `listen stats` permet de définir un *endpoint* pour la consultation des statistiques d'utilisation de HAProxy sur le port 7000. Elle est consultable en se connectant avec un navigateur web (paramètre `mode`).

La section `listen primary` permet de renvoyer les connexions sur l'instance primaire lorsque l'on se connecte au port 5000 de HAProxy (paramètre `bind`). Une vérification est réalisée sur le `endpoint /primary` de l'API REST de Patroni pour établir quelle instance est la primaire (paramètres `http-check` et option `httpch1`). Pour se faire HAProxy test pour chaque serveur déclaré (paramètres `server`), si l'API REST qui écoute sur le port 8008 (paramètre `check port` du serveur) répond par un code retour http 200.

La section `listen replicas` permet de renvoyer les connexions sur les instances secondaires lorsque l'on se connecte au port 5001 de HAProxy. Une vérification est réalisée sur le `endpoint /replica` de l'API REST de Patroni pour vérifier quelles instances sont des secondaires. Pour se faire HAProxy test pour chaque serveur déclaré, si l'API REST qui écoute sur le port 8008 répond par un code retour http 200.

Pour chacune des sections permettant de se connecter à PostgreSQL, la connexion est vérifiée toutes les 3 secondes (paramètre `inter`). Au bout de trois échecs consécutifs, le serveur est considéré comme hors service (paramètre `fall`) et les sessions en cours sont stoppées (paramètre `on-marked-down shutdown-sessions`). Si un serveur est marqué comme indisponible, il faut 2 tests réussis consécutivement avant que le serveur soit considéré comme disponible (paramètre `rise`).

Il faut faire attention à l'utilisation du paramètre `lag=valeur` sur le `endpoint` de l'API REST de Patroni. Une valeur trop faible peut entraîner des changements de statut fréquents et donc des déconnexions fréquentes.

Par défaut, la répartition de charge se fait avec un algorithme de *round-robin*. On peut changer l'algorithme en ajoutant un paramètre `balance` dans la définition d'un `endpoint` et/ou spécifier des poids par serveur pour influencer l'algorithme (paramètre `weight` d'un serveur). Il y a beaucoup de possibilités d'algorithme, par exemple :

- `static-rr`: comme *round-robin* mais sans prendre en compte les poids ;
- `leastconn`: serveur avec le moins de connexions ;
- `first`: premier serveur (tri par id) avec une connexion disponible, si aucun id n'est donné pour un serveur, il se voit allouer un id automatiquement.

Le nom fourni pour la déclaration des serveurs sera visible dans la page de statistiques, ici `node1`, `node2`, `node3`.

### 5.6.13.3 Keepalived



- Monte la VIP sur le serveur de l'instance primaire
- Utilisation de l'API REST de Patroni

Une VIP ou *Virtual IP address* est une adresse IP qui peut être partagée par plusieurs serveurs. Elle n'est active que sur un serveur à la fois. Cela permet d'avoir un point d'accès unique qui change en fonction de la disponibilité d'un service sur plusieurs serveurs.

Keepalived permet de gérer une VIP et de s'assurer qu'elle ne soit montée que sur un seul serveur. Il permet d'utiliser des scripts de vérification, de surveiller l'état d'un processus, la disponibilité d'un serveur ou la présence d'un fichier de déclenchement afin de conditionner le montage de la VIP.

Keepalived est un outil très versatile qui permet aussi de faire la répartition de charge.

Voici un exemple de configuration pour maintenir une VIP sur le serveur de l'instance primaire d'un cluster Patroni. Elle doit être mise en place sur tous les serveurs PostgreSQL du cluster.

```
global_defs {
    enable_script_security
    script_user root
}

vrrp_script keepalived_check_patroni {
    script "/usr/local/bin/keepalived_check_patroni.sh"
    interval 3          # interval between checks
    timeout 5           # how long to wait for the script return
    rise 1              # How many time the script must return ok, for the
                        # host to be considered healthy (avoid flapping)
    fall 1              # How many time the script must return Ko; for the
                        # host to be considered unhealthy (avoid flapping)
}

vrrp_instance VI_1 {
    state MASTER
    interface eth1
    virtual_router_id 51
    priority 244
    advert_int 1
    virtual_ipaddress {
        10.20.30.50/32
    }
    track_script {
        keepalived_check_patroni
    }
}
```

La section `vrrp_instance` permet de gérer la configuration de la vip (spécifiée dans le paramètre `virtual_ipaddress`). Ici l'état initial de cette instance sera `MASTER` (paramètre `state`). La

VIP sera montée sur l’interface `eth1` (paramètre `interface`). Il est important de choisir un `virtual_router_id` inutilisé pour la configuration de la VIP. Il est possible de mettre un poids sur les serveurs (paramètre `priority`), par convention un serveur primaire devrait avoir la priorité 255. Le cas présent est un peu différent, puisqu’on utilise un script pour déterminer l’état de l’instance (paramètre `track_script`).

La section `keepalived_check_patroni` permet de gérer le script chargé de vérifier l’état de l’instance dans Patroni. Le script utilisé doit être capable d’interroger l’API REST de Patroni pour connaître l’état d’une instance en particulier (paramètre `script`). On peut définir la fréquence de passage du script (paramètre `interval`), ainsi qu’un timeout pour le script (paramètre `timeout`). Il est possible de configurer le nombre d’échecs successifs du script avant qu’une ressource ne soit considérée comme indisponible (paramètre `fall`). De même, on peut spécifier le nombre succès du script pour que la ressource soit considérée comme à nouveau disponible.

Voici un exemple de script de vérification :

```
#!/bin/bash

/usr/bin/curl \
-X GET -I --fail \
# --cacert ca.pem --cert p1.pem --key p1-key.pem \
https://127.0.0.1:8008/primary &>>/var/log/patroni/keepalived_vip.log
```

Il faut prévoir un *logrotate* sur le fichier de log.

### 5.6.14 Questions



- C'est le moment !

## 5.7 INSTALLATION DE PATRONI DEPUIS LES PAQUETS COMMUNAUTAIRES



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

Elle ne dure que quelques minutes. Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Sur chacun des nœuds, nous allons installer PostgreSQL 14. Ce qui suit vise au plus rapide, les méthodes recommandées figurent dans notre module sur l'installation<sup>23</sup>.

### 5.7.1 Sur Red Hat 7 / CentOS 7

```
# yum install -y
↳ https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/\
pgdg-redhat-repo-latest.noarch.rpm

# yum install -y postgresql14-server
```

Sur chacun des nœuds, installer Patroni :

```
# yum -y install epel-release
# yum -y install patroni-etcd
```

Sur chacun des nœuds, créer un répertoire /etc/patroni pour la configuration :

```
# mkdir /etc/patroni
```

Le fichier de configuration que nous utiliserons par la suite sera /etc/patroni/patroni.yml.

### 5.7.2 Sur Rocky Linux 8

Sur chacun des nœuds, installer PostgreSQL 14 :

```
# 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
# dnf install -y postgresql14-server
```

Sur chacun des nœuds, installer Patroni :

```
# dnf install -y epel-release
# dnf install -y patroni-etcd
```

Sur chacun des nœuds, créer un répertoire /etc/patroni pour la configuration :

---

<sup>23</sup>[https://dali.bo/b\\_html#installation-de-postgresql-depuis-les-paquets-communautaires](https://dali.bo/b_html#installation-de-postgresql-depuis-les-paquets-communautaires)

```
# mkdir /etc/patroni
```

Le fichier de configuration que nous utiliserons par la suite sera `/etc/patroni/patroni.yml`.

### 5.7.3 Sur Debian / Ubuntu

Sur chacun des nœuds, installer PostgreSQL 14 :

```
# echo "deb http://apt.postgresql.org/pub/repos/apt buster-pgdg main" > \
/etc/apt/sources.list.d/pgdg.list
# curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
# apt update && apt upgrade -y && apt install -y postgresql-14
```

Sur chacun des nœuds, supprimer l'instance installée via le paquet `postgresql-14` :

```
$ pg_dropcluster 14 main --stop
```

Sur chacun des nœuds, installer Patroni :

```
# apt install -y patroni
```

NB : sur Debian, le fichier de configuration attendu par le service est `/etc/patroni/config.yml`.

### 5.7.4 Configuration de Patroni

Liste des variables utilisées dans la procédure à ajuster :

Variable	Description
PATRONI_CLUSTER_NAME	Nom du cluster Patroni. Ex: cluster-test-01
PATRONI1_NAME	Nom du nœud 1 Patroni. Ex: node-pg1
PATRONI2_NAME	Nom du nœud 2 Patroni. Ex: node-pg2
PATRONI3_NAME	Nom du nœud 3 Patroni. Ex: node-pg3
PATRONI1_IP	Adresse IPv4 du nœud 1 Patroni
PATRONI2_IP	Adresse IPv4 du nœud 2 Patroni
PATRONI3_IP	Adresse IPv4 du nœud 3 Patroni
ETCD1_IP	Adresse IPv4 du nœud 1 etcd
ETCD2_IP	Adresse IPv4 du nœud 2 etcd
ETCD3_IP	Adresse IPv4 du nœud 3 etcd
ETCD_ROOT_PASSWORD	Mot de passe du super-utilisateur <code>root</code> d'etcd
PGDATA	Répertoire des données de l'instance PostgreSQL
PGPORT	Numéro de port d'écoute de l'instance PostgreSQL
PGBIN	Répertoire contenant les binaires PostgreSQL

Le chemin du répertoire des données (PGDATA) que nous emploierons sur Red Hat ou CentOS sera `/var/lib/pgsql/14/${PATRONI_CLUSTER_NAME}`. Sur Debian ou Ubuntu, ce sera `/var/lib/postgresql/14/${PATRONI_CLUSTER_NAME}`.

Rappel : sous Red Hat/CentOS, les binaires PostgreSQL sont localisés dans `/usr/pgsql-14/bin`, tandis que sous Debian/Ubuntu, ils seront dans `/usr/lib/postgresql/14/bin`.

Sur le nœud Patroni n°1, en plus des variables mentionnées ci-dessus, définir :

```
PATRONI_NAME=${PATRONI1_NAME}  
PATRONI_IP=${PATRONI1_IP}
```

Sur le nœud n°2, en complément des variables dans le tableau ci-dessus, définir :

```
PATRONI_NAME=${PATRONI2_NAME}  
PATRONI_IP=${PATRONI2_IP}
```

Sur le nœud n°3, en complément des variables dans le tableau ci-dessus, définir :

```
PATRONI_NAME=${PATRONI3_NAME}  
PATRONI_IP=${PATRONI3_IP}
```

Puis, sur chacun des nœuds, dans les mêmes terminaux que ceux précédemment utilisés pour définir les variables utiles pour créer la configuration Patroni, créer le fichier `/etc/patroni/patroni.yml` comme suit :

```
# cat >/etc/patroni/patroni.yml <<_CONFIGURATION_PATRONI_  
scope: ${PATRONI_CLUSTER_NAME}  
namespace: /service/ # valeur par défaut  
name: ${PATRONI_NAME}  
  
restapi:  
  listen: ${PATRONI_IP}:8008  
  connect_address: ${PATRONI_IP}:8008  
log:  
  level: INFO  
  dir: /var/log/patroni/${PATRONI_CLUSTER_NAME}  
etcd:  
  hosts:  
    - ${ETCD1_IP}:2379  
    - ${ETCD2_IP}:2379  
    - ${ETCD3_IP}:2379  
  username: root  
  password: ${ETCD_ROOT_PASSWORD}  
  protocol: http  
bootstrap:  
  dcs:  
    ttl: 30  
    loop_wait: 10  
    retry_timeout: 10  
    maximum_lag_on_failover: 1048576  
  postgresql:  
    use_pg_rewind: true  
    use_slots: true  
    parameters:  
      wal_level: replica
```

```
hot_standby: on
  max_wal_senders: 10
  max_replication_slots: 5
initdb:
- encoding: UTF8
- data-checksums
pg_hba:
- host all all all md5
- host replication replicator all md5
users:
  dba:
    password: dba_password
    options:
      - createrole
      - createdb
postgresql:
  listen: "*:${PGPORT}"
  connect_address: ${PATRONI_IP}:${PGPORT}
  data_dir: ${PGDATA}
  bin_dir: ${PGBIN}
  authentication:
    replication:
      username: replicator
      password: replicator_password
    superuser:
      username: postgres
      password: postgres_password
    rewind:
      username: rewinder
      password: rewinder_password
  parameters:
    unix_socket_directories: '.'
basebackup:
  max-rate: "100M"
  checkpoint: "fast"
watchdog:
  mode: automatic
  device: /dev/watchdog
  safety_margin: 5
tags:
  nofailover: false
  noloadbalance: false
  clonefrom: false
  nosync: false
_CONFIGURATION_PATRONI_
```

Noter que ce fichier contient :

- des utilisateurs et mots de passe par défaut ;
- un début de configuration du pg\_hba.conf ;
- des paramètres de PostgreSQL.

Par exemple, pour créer le fichier /etc/patroni/patroni.yml du 1er nœud Patroni sous Red Hat ou CentOS :

```
PATRONI_CLUSTER_NAME='cluster-test-01'
```

```
PATRONI1_NAME='node-pg1'
PATRONI2_NAME='node-pg2'
PATRONI3_NAME='node-pg3'
PATRONI1_IP='172.16.100.1'
PATRONI2_IP='172.16.100.2'
PATRONI3_IP='172.16.100.3'
ETCD1_IP='172.16.200.1'
ETCD2_IP='172.16.200.2'
ETCD3_IP='172.16.200.3'
ETCD_ROOT_PASSWORD='root_password'
PGDATA="/var/lib/pgsql/14/${PATRONI_CLUSTER_NAME}"
PGPORT='5434'
PGBIN='/usr/pgsql-14/bin'

PATRONI_NAME=${PATRONI1_NAME}
PATRONI_IP=${PATRONI1_IP}

cat >/etc/patroni/patroni.yml <<_CONFIGURATION_PATRONI_
[voir plus haut pour le contenu.]
_CONFIGURATION_PATRONI_
```

Le contenu du fichier sera donc :

```
scope: cluster-test-01
namespace: /service/ # valeur par défaut
name: node-pg1

restapi:
  listen: 172.16.100.1:8008
  connect_address: 172.16.100.1:8008
log:
  level: INFO
  dir: /var/log/patroni/cluster-test-01
etcd:
  hosts:
    - 172.16.200.1:2379
    - 172.16.200.2:2379
    - 172.16.200.3:2379
  username: root
  password: root_password
  protocol: http
bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 10
    maximum_lag_on_failover: 1048576
  postgresql:
    use_pg_rewind: true
    use_slots: true
    parameters:
      wal_level: replica
      hot_standby: on
      max_wal_senders: 10
      max_replication_slots: 5
initdb:
```

```
- encoding: UTF8
- data-checksums
pg_hba:
- host all all all md5
- host replication replicator all md5
users:
dba:
  password: dba_password
  options:
  - createrole
  - createdb
postgresql:
  listen: "*:5434"
  connect_address: 172.16.100.1:5434
  data_dir: /var/lib/pgsql/14/cluster-test-01
  bin_dir: /usr/pgsql-14/bin
  authentication:
    replication:
      username: replicator
      password: replicator_password
    superuser:
      username: postgres
      password: postgres_password
    rewind:
      username: rewinder
      password: rewinder_password
  parameters:
    unix_socket_directories: '.'
basebackup:
  max-rate: "100M"
  checkpoint: "fast"
watchdog:
  mode: automatic
  device: /dev/watchdog
  safety_margin: 5
tags:
  nofailover: false
  noloadbalance: false
  clonefrom: false
  nosync: false
```

Sur chacun des nœuds, rendre l'utilisateur et le groupe `postgres` propriétaire :

```
# chown postgres. /etc/patroni/patroni.yml
```

Sur chacun des nœuds, créer le répertoire qui hébergera les traces de Patroni :

```
# install -o postgres -g postgres -m 0750 -d
↪ /var/log/patroni/${PATRONI_CLUSTER_NAME}
```

#### 5.7.4.1 Gestion du service

Activation et démarrage du service, à exécuter sur tous les nœuds :

```
# systemctl enable --now patroni
```

Le cluster doit démarrer et être créé :

```
root@pg-1:/etc/patroni# systemctl status patroni
● patroni.service - Runners to orchestrate a high-availability PostgreSQL
  Loaded: loaded (/lib/systemd/system/patroni.service; enabled; vendor preset:
    ↳ enabled)
  Active: active (running) since Fri 2022-04-22 16:50:25 UTC; 1min 25s ago
    Main PID: 8364 (patroni)
      Tasks: 13 (limit: 18846)
     Memory: 59.0M
    CGroup: /system.slice/patroni.service
            ├─8364 /usr/bin/python3 /usr/bin/patroni /etc/patroni/config.yml
            ├─8387 /usr/lib/postgresql/14/bin/postgres -D
    ↳ /var/lib/postgresql/14/cluster-test-01
          └─--config-file=/var/lib/postgresql/14/cluster-test-01/postgresql.conf
              └─--listen_addresses=* --port=5432 --cluster_name=cluster-test-01
    ↳ --wal_level=replica
        └─--hot_standby=on --max_connections
            ├─8392 postgres: cluster-test-01: checkpointer
            ├─8393 postgres: cluster-test-01: background writer
            ├─8394 postgres: cluster-test-01: stats collector
            ├─8401 postgres: cluster-test-01: postgres postgres ::1(37886) idle
            ├─8406 postgres: cluster-test-01: walwriter
            ├─8407 postgres: cluster-test-01: autovacuum launcher
            └─8408 postgres: cluster-test-01: logical replication launcher
...
...
```

#### 5.7.4.2 Traces

Le fichier `/var/log/patroni.log` se destiné uniquement à Patroni.

Les traces de l'instance sont redirigées par Patroni et peuvent être exploitées par `systemctl`:

```
# journalctl -u patroni --follow
```

#### 5.7.4.3 Tests du cluster Patroni

En tant qu'utilisateur `postgres` : vérifier l'état du cluster Patroni :

```
$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: ${PATRONI_CLUSTER_NAME} (<dbsysid>) -----+-----+-----+
| Member           | Host             | Role   | State  | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| ${PATRONI1_NAME} | ${PATRONI1_IP}:5434 | Leader | running | 1  | |
| ${PATRONI2_NAME} | ${PATRONI2_IP}:5434 | Replica | running | 1  | 0 |
| ${PATRONI3_NAME} | ${PATRONI3_IP}:5434 | Replica | running | 1  | 0 |
+-----+-----+-----+-----+-----+
```

En reprenant l'exemple du nœud 1 sur Red Hat/CentOS, avec un cluster composé de trois nœuds, la sortie pourrait ressembler à :

+ Cluster: cluster-test-01 (6916914555750681574) +-----+					
Member	Host	Role	State	TL	Lag in MB
node-pg1	172.16.100.1:5434	Leader	running	1	
node-pg2	172.16.100.2:5434	Replica	running	1	0
node-pg3	172.16.100.3:5434	Replica	running	1	0

Depuis l'utilisateur système **postgres**, on peut par exemple ainsi à l'instance (cela dépend bien sûr du pg\_hba.conf) :

```
$ psql -h localhost
```

à noter que pour une connexion en local, la socket Unix est dans le PGDATA :

```
$ psql -p5432 -h /var/lib/postgresql/14/cluster-test-01/
```

#### 5.7.4.4 Tests de bascule

Il est facile de tuer une instance PostgreSQL et le service patroni qui la gère sur un nœud :

```
# killall -u postgres
```

L'instance passe en statut stopped. Un simple redémarrage du service Patroni permet de le redémarrer :

```
# systemctl restart patroni
```

#### 5.7.4.5 Modification de la configuration dynamique

```
$ patronictl -c /etc/patroni/patroni.yml edit-config
```

Ajouter ou modifier le paramétrage dans le fichier YAML qui s'affiche.

Les modifications de paramétrage seront propagées à tous les nœuds par la suite.

## 5.8 QUIZ



[https://dali.bo/w8\\_quiz](https://dali.bo/w8_quiz)

## 5.9 TRAVAUX PRATIQUES



### 5.9.1 Raft

Nous allons utiliser le simulateur Raftscope<sup>24</sup>.

Les 5 nœuds portent tous le dernier numéro de mandat qu'ils ont connu. Le *leader* est cerclé de noir.

Le *time out* de chaque *follower* se décrémente en permanence et est réinitialisé quand un message de *heart beat* du *leader* est reçu.

Le journal de chaque nœud est visible sur la droite pour suivre la propagation des informations de chaque *request*.

Il est possible de mettre en pause, d'accélérer ou ralentir.

Les actions se font par clic droit sur chaque nœud :

- *stop / resume / restart* pour stopper/(re)démarrer un nœud ;
- *time out* sur un nœud provoque une élection.
- *request* est une interrogation client (à faire uniquement sur le *leader*).
  - Observer la première élection et l'échange des *heart beats* par la suite.
  - Mettre en défaut le *leader* (*stop*) et attendre une élection.
  - Lancer plusieurs *requests* sur le *leader*.
  - Remettre en route l'ancien *leader*.
  - Arrêter deux nœuds, dont le *leader*, et observer le comportement du cluster.
  - Arrêter un nœud secondaire (pour un total de 3 nœuds arrêtés).
  - Tester en soumettant des *requests* au primaire.
  - Qu'en est-il de la tolérance de panne ?

---

<sup>24</sup><https://raft.github.io/raftscope/index.html>

- Rallumer un nœud pour revenir à 3 nœuds actifs dont un *leader*.
- Éteindre le *leader*. Tenter de soumettre des *requests*.
- Que se passe-t-il ?

### 5.9.2 etcd : installation



```
$ sudo ansible-playbook -i inventory.yml setup.yml
```

La suite des travaux pratiques repose sur des conteneurs LXC sous Debian (3 conteneurs pour les 3 nœuds etcd, 3 pour les 3 nœuds Patroni). Trois fichiers sont requis :

- *inventory* (déclaration des machines)
- *setup.yml* (configuration)
- *teardown.yml* (suppression)

#### Création :

Ce travail est facilité par l'usage d'un *playbook Ansible* qu'il faut appliquer à l'aide de la commande :

```
$ sudo ansible-playbook -i inventory.yml setup.yml
```

#### Suppression :

La commande suivante permet de supprimer tous les conteneurs créés précédemment :

```
$ sudo ansible-playbook -i inventory.yml teardown.yml
```

- Créer les machines avec le *playbook* fourni.
- Installer et configurer etcd sur les 3 nœuds correspondants (**e1**, **e2**, **e3**) à l'aide du guide d'installation précédent (adapter les IP et noms de machines).

### 5.9.3 etcd : manipulation (optionnel)



**But** : manipuler la base de données distribuée d'Etcd.

- Depuis un nœud, utiliser `etcdctl set` pour écrire une clef `foo` à la valeur `bar` (l'authentification étant activée, il faudra préciser l'utilisateur et le mot de passe de **root**).
- Récupérer cette valeur depuis un autre nœud.
- Modifier la valeur à `baz`.
- Créer un répertoire `food` contenant les clés/valeur `poisson: bar` et `vin: blanc`.
- Récupérer récursivement toutes les clefs du répertoire `food` en exigeant une réponse d'un quorum.

**But :** constater le comportement d'Etcd conforme à l'algorithme Raft.

- Tout en observant les logs de etcd et la santé de l'agrégat, procéder au *fencing* du *leader* avec `lxc-freeze`.
- Geler le nouveau *leader* de la même manière et voir les traces du nœud restant.

#### 5.9.4 Patroni : installation

Sur les machines créées précédemment :

- installer et configurer Patroni sur **2 nœuds** en suivant le guide d'installation
- observer les traces de chaque nœud Patroni
- observer la topologie de l'agrégat avec `patronictl`.
- Ajouter le 3ème nœud à l'agrégat.
- Déterminer le primaire via l'API Patroni en direct avec `curl`.
- Quels sont les slots de réPLICATION sur le *leader* ?

#### 5.9.5 Patroni : utilisation

- Se connecter depuis l'extérieur en tant que **dba** à la base **postgres**, à un nœud.

- Comment garantir une connexion au *leader* pour écrire ?

- Créer une table :

```
CREATE TABLE insertions (id int GENERATED ALWAYS AS IDENTITY,  
                      d timestampz DEFAULT now(),  
                      source text DEFAULT inet_server_addr()  
                    );  
GRANT ALL ON TABLE insertions TO dba ;
```

- Insérer une ligne toutes les secondes, à chaque fois dans une nouvelle connexion au *leader*.
- Dans une autre fenêtre, afficher les 20 dernières lignes de cette table.

- Stopper le nœud *leader* Patroni.
- Que se passe-t-il dans la topologie, et dans les requêtes ci-dessus ?

- Arrêter à nouveau le *leader*. Il doit donc rester un nœud actif. Que se passe-t-il ?

- Arrêter deux nœuds du cluster etcd. Que se passe-t-il ?

- Redémarrer les nœuds etcd.

- Relancer un des nœuds Patroni.

- Sur le troisième nœud (arrêté), détruire le PGDATA. Relancer Patroni.

- Forcer un *failover* vers le nœud **pg-1**.

- Modifier les paramètres shared\_buffers et work\_mem. Si besoin, redémarrer les nœuds.

## 5.10 TRAVAUX PRATIQUES (SOLUTIONS)



### 5.10.1 Raft

- Observer la première élection et l'échange des *heart beats* par la suite.

Les nœuds portent au départ le n° 1, il n'y a pas de *leader*. Il faut attendre que l'un des nœuds ait atteint son *time out* pour qu'il propose une élection (qu'il va gagner), avec le mandat n°2.

Les autres se raccrochent à lui.

- Mettre en défaut le *leader* (*stop*) et attendre une élection.

Le même phénomène se produit et l'on arrive au mandat 3.

- Lancer plusieurs *requests* sur le *leader*.

Elles apparaissent sur la droite dans le journal au fur et à mesure que le *leader* les diffuse.

- Remettre en route l'ancien *leader*.

Celui-ci redémarre à 2 (en tant que *follower*) et bascule sur le mandat 3 au premier *heart beat* reçu. Il rattrape ensuite son journal.

- Arrêter deux nœuds, dont le *leader*, et observer le comportement du cluster.

Comme précédemment, les trois nœuds restants s'accordent sur un primaire.

- Arrêter un nœud secondaire (pour un total de 3 nœuds arrêtés).
- Tester en soumettant des *requests* au primaire.
- Qu'en est-il de la tolérance de panne ?

Le cluster continue de fonctionner : on peut lui soumettre des *requests*. Elles sont bien répliquées mais ne sont pas exécutées (ou commitées) par le *leader*. Pour cela, il faut que les *\_request\_aient* été répliquées vers la majorité des nœuds du cluster. Cela signifie que le client reste en attente de confirmation du traitement de sa demande.

La tolérance de panne est maintenant nulle.

- Rallumer un nœud pour revenir à 3 nœuds actifs dont un *leader*.
- Éteindre le *leader*. Tenter de soumettre des *requests*.
- Que se passe-t-il ?

L'un des deux nœuds survivants lance une élection. L'autre le suit. Mais comme le quorum de 3 (moitié de 5 plus 1) n'est pas obtenu, l'élection échoue. Les *time out* continuent d'expirer et de nouvelles élections sont lancées.

Faute de *leader*, il devient impossible de soumettre des *requests*. Tout client s'appuyant dessus reçoit une erreur. Il faut attendre le retour en ligne d'un nœud et l'élection d'un nouveau *leader*.

### 5.10.2 etcd : installation

- Créer les machines avec le *playbook* fourni.
- Installer et configurer etcd sur les 3 nœuds correspondants (**e1**, **e2**, **e3**) à l'aide du guide d'installation précédent (adapter les IP et noms de machines).

Cette installation de devrait pas poser de problème, sauf erreur de frappe. À titre d'exemple, voici le fichier pour le nœud **e2** :

```
ETCD_NAME="e2"
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://127.0.0.1:2380,http://10.0.3.102:2380"
ETCD_LISTEN_CLIENT_URLS="http://127.0.0.1:2379,http://10.0.3.102:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.3.102:2380"
ETCD_INITIAL_CLUSTER="e1=http://10.0.3.101:2380,e2=http://10.0.3.102:2380,e3=http://10.0.3.103:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.3.101:2379,http://10.0.3.102:2379,http://10.0.3.103:2379"
```

Les trois nœuds doivent répondre à cette commande (deux *followers* et un *leader*) :

```
curl -s http://10.0.3.102:2379/v2/stats/self | jq
{
  "name": "e2",
  "id": "97463691c7858a7b",
  "state": "StateFollower",
  "startTime": "2022-04-25T12:18:21.838513548Z",
  "leaderInfo": {
    "leader": "7ef9d5bb55cefbc",
    "uptime": "3h50m31.057077064s",
    "startTime": "2022-04-25T12:18:23.156685361Z"
  },
  "recvAppendRequestCnt": 66487,
  "recvPkgRate": 4.817513333917741,
  "recvBandwidthRate": 471.92360619058195,
  "sendAppendRequestCnt": 0
}
```

### 5.10.3 etcd : manipulation (optionnel)

- Depuis un nœud, utiliser `etcdctl set` pour écrire une clef `foo` à la valeur `bar` (l'authentification étant activée, il faudra préciser l'utilisateur et le mot de passe de **root**).

```
$ etcdctl -u root set foo bar
```

Il faudra le mot de passe de **root**.

- Récupérer cette valeur depuis un autre nœud.

```
$ etcdctl -u root get foo
```

bar

- Modifier la valeur à `baz`.

```
$ etcdctl -u root update foo baz
```

baz

ou aussi :

```
$ etcdctl -u root set foo baz
```

baz

- Créer un répertoire `food` contenant les clés/valeur `poisson: bar` et `vin: blanc`.

```
$ etcdctl -u root mkdir food
$ etcdctl -u root set food/poisson bar
```

bar

```
$ etcdctl -u root set food/vin blanc
```

blanc

```
$ etcdctl -u root ls --recursive food
```

/food/vin

/food/poisson

- Récupérer récursivement toutes les clefs du répertoire `food` en exigeant une réponse d'un `quorum`.

```
$ etcdctl -u root ls --recursive --quorum food
```

/food/vin

/food/poisson

**But** : constater le comportement d'Etcd conforme à l'algorithme Raft.

- Tout en observant les logs de etcd et la santé de l'agrégat, procéder au *fencing* du leader avec `lxc-freeze`.

Dans une fenêtre sur le nœud **e2**, laisser défiler le journal :

```
# journalctl -u etcd -f
```

Depuis une session dans le serveur hôte, consulter la santé de l'agrégat avec :

```
$ watch -n1 "curl -s -XGET http://e2:2379/health | python -m json.tool"
{
    "health": "true"
}
```

ainsi que chaque nœud :

```
$ watch -n 1 "curl -s http://e1:2379/v2/stats/self|python -m json.tool"
$ watch -n 2 "curl -s http://e2:2379/v2/stats/self|python -m json.tool"
$ watch -n 3 "curl -s http://e3:2379/v2/stats/self|python -m json.tool"
```

On suppose ici que **e1** possède l'état :

```
"state": "StateLeader",
```

Depuis le serveur hôte, cette commande gèle le conteneur LXC de **e1** :

```
# lcx-freeze e1
```

Les traces sur **e2** indiquent alors le déclenchement de l'élection (l'ID indiqué est celui de **e2**) avec un nouveau mandat 498. L'autre nœud accepte, et le quorum (ici 2 sur 3) permet l'élection de **e2** comme nouveau leader.

```
16:57:07 e2 etcd[92]: 97463691c7858a7b is starting a new election at term 497
16:57:07 e2 etcd[92]: 97463691c7858a7b became candidate at term 498
16:57:07 e2 etcd[92]: 97463691c7858a7b received MsgVoteResp from 97463691c7858a7b at
  ↵ term 498
16:57:07 e2 etcd[92]: 97463691c7858a7b [logterm: 497, index: 67990]
  ↵ sent MsgVote request to 736293150f1cffb7 at term 498
16:57:07 e2 etcd[92]: 97463691c7858a7b [logterm: 497, index: 67990]
  ↵ sent MsgVote request to 7ef9d5bb55cefbc at term 498
16:57:07 e2 etcd[92]: raft.node: 97463691c7858a7b lost leader 736293150f1cffb7 at
  ↵ term 498
16:57:07 e2 etcd[92]: 97463691c7858a7b received MsgVoteResp from 7ef9d5bb55cefbc at
  ↵ term 498
16:57:07 e2 etcd[92]: 97463691c7858a7b [quorum:2] has received 2 MsgVoteResp votes
  ↵ and 0 vote rejections
16:57:07 e2 etcd[92]: 97463691c7858a7b became leader at term 498
16:57:07 e2 etcd[92]: raft.node: 97463691c7858a7b elected leader 97463691c7858a7b at
  ↵ term 498
16:57:11 e2 etcd[92]: lost the TCP streaming connection with peer 736293150f1cffb7
  ↵ (stream MsgApp v2 reader)
16:57:11 e2 etcd[92]: failed to read 736293150f1cffb7 on stream MsgApp v2
  ↵ (read tcp 10.0.3.102:56262->10.0.3.101:2380: i/o timeout)
```

```
16:57:11 e2 etcd[92]: peer 736293150f1cffb7 became inactive (message send to peer
↪ failed)
16:57:11 e2 etcd[92]: lost the TCP streaming connection with peer 736293150f1cffb7
↪ (stream Message reader)
```

On a effectivement en interrogeant l'état de **e2** :

```
{
  "name": "e2",
  "id": "97463691c7858a7b",
  "state": "StateLeader",
  ...
}
```

- Geler le nouveau *leader* de la même manière et voir les traces du nœud restant.

L'état de l'agrégat tombe en erreur suite à cette perte du quorum :

```
$ curl -s -XGET http://e3:2379/health |python3 -m json.tool

{
  "health": "false"
}
```

Dans les traces de **e3**, on constate qu'il tente une élection, envoie des messages mais faute de réponse, n'obtient pas l'accord pour une élection. Puis il recommence :

```
17:02:49 e3 etcd[93]: health check for peer 736293150f1cffb7 could not connect:
  read tcp 10.0.3.103:45098->10.0.3.101:2380: i/o timeout ...
17:02:51 e3 etcd[93]: 7ef9d5bb55cefbcc is starting a new election at term 498
17:02:51 e3 etcd[93]: 7ef9d5bb55cefbcc became candidate at term 499
17:02:51 e3 etcd[93]: 7ef9d5bb55cefbcc received MsgVoteResp from 7ef9d5bb55cefbcc at
↪ term 499
17:02:51 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 736293150f1cffb7 at term 499
17:02:51 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 97463691c7858a7b at term 499
17:02:51 e3 etcd[93]: raft.node: 7ef9d5bb55cefbcc lost leader 97463691c7858a7b at
↪ term 499
17:02:53 e3 etcd[93]: 7ef9d5bb55cefbcc is starting a new election at term 499
17:02:53 e3 etcd[93]: 7ef9d5bb55cefbcc became candidate at term 500
17:02:53 e3 etcd[93]: 7ef9d5bb55cefbcc received MsgVoteResp from 7ef9d5bb55cefbcc at
↪ term 500
17:02:53 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 736293150f1cffb7 at term 500
17:02:53 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 97463691c7858a7b at term 500
17:02:54 e3 etcd[93]: health check for peer 736293150f1cffb7 could not connect:
  read tcp 10.0.3.103:45140->10.0.3.101:2380: i/o timeout ...
17:02:54 e3 etcd[93]: 7ef9d5bb55cefbcc is starting a new election at term 500
```

La situation revient à la normale dès qu'un des deux autres revient en ligne (commande `lxc-unfreeze`). D'ailleurs **e3** perd l'élection :

```

17:08:04 e3 etcd[93]: peer 97463691c7858a7b became active
17:08:04 e3 etcd[93]: established a TCP streaming connection with peer
  ↳ 97463691c7858a7b ...
17:08:04 e3 etcd[93]: established a TCP streaming connection with peer
  ↳ 97463691c7858a7b ...
17:08:04 e3 etcd[93]: got unexpected response error (etcdserver: request timed out)
17:08:05 e3 etcd[93]: 7ef9d5bb55cefbcc is starting a new election at term 719
17:08:05 e3 etcd[93]: 7ef9d5bb55cefbcc became candidate at term 720
17:08:05 e3 etcd[93]: 7ef9d5bb55cefbcc received MsgVoteResp from 7ef9d5bb55cefbcc at
  ↳ term 720
17:08:05 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 736293150f1cffb7 at term 720
17:08:05 e3 etcd[93]: 7ef9d5bb55cefbcc [logterm: 498, index: 69146]
  sent MsgVote request to 97463691c7858a7b at term 720
17:08:05 e3 etcd[93]: lost the TCP streaming connection with peer 97463691c7858a7b
  ↳ ...
17:08:05 e3 etcd[93]: established a TCP streaming connection with peer
  ↳ 97463691c7858a7b ...
17:08:06 e3 etcd[93]: got unexpected response error (etcdserver: request timed out)
  [merged 1 repeated lines in 1.63s]
17:08:06 e3 etcd[93]: 7ef9d5bb55cefbcc [term: 720] received a MsgVote message with
  ↳ higher term
    from 97463691c7858a7b [term: 721]
17:08:06 e3 etcd[93]: 7ef9d5bb55cefbcc became follower at term 721

```

En effet, si **e2** se raccroche aussitôt à **e3** (mandat 720), l'expiration de son *time out* lui fait déclencher aussitôt une autre élection ;

```

17:08:04 e2 etcd[92]: failed to send out heartbeat on time (exceeded the 100ms
  ↳ timeout for 5m14.099597803s)
17:08:04 e2 etcd[92]: server is likely overloaded
...
17:08:05 e2 etcd[92]: established a TCP streaming connection with peer
  ↳ 7ef9d5bb55cefbcc ...
17:08:05 e2 etcd[92]: 97463691c7858a7b [term: 498] received a MsgAppResp message
  ↳ with higher term
    from 7ef9d5bb55cefbcc [term: 720]
17:08:05 e2 etcd[92]: 97463691c7858a7b became follower at term 720
17:08:05 e2 etcd[92]: raft.node: 97463691c7858a7b changed leader from
  ↳ 97463691c7858a7b
    to 7ef9d5bb55cefbcc at term 720
17:08:06 e2 etcd[92]: 97463691c7858a7b is starting a new election at term 720
17:08:06 e2 etcd[92]: 97463691c7858a7b became candidate at term 721
17:08:06 e2 etcd[92]: 97463691c7858a7b received MsgVoteResp from 97463691c7858a7b at
  ↳ term 721
17:08:06 e2 etcd[92]: 97463691c7858a7b [logterm: 498, index: 69169]
  sent MsgVote request to 736293150f1cffb7 at term 721
17:08:06 e2 etcd[92]: 97463691c7858a7b [logterm: 498, index: 69169] sent MsgVote
  ↳ request
    to 7ef9d5bb55cefbcc at term 721
17:08:06 e2 etcd[92]: raft.node: 97463691c7858a7b lost leader 7ef9d5bb55cefbcc at
  ↳ term 721
17:08:06 e2 etcd[92]: 97463691c7858a7b received MsgVoteResp from 7ef9d5bb55cefbcc at
  ↳ term 721
17:08:06 e2 etcd[92]: 97463691c7858a7b [quorum:2] has received 2 MsgVoteResp votes
  ↳ and 0 vote rejections

```

```
17:08:06 e2 etcd[92]: 97463691c7858a7b became leader at term 721
17:08:06 e2 etcd[92]: raft.node: 97463691c7858a7b elected leader 97463691c7858a7b at
  ↵ term 721
```

#### 5.10.4 Patroni : installation

Dans ce qui suit, ce n'est pas à faire en tant que **root** est à faire en tant qu'utilisateur système **postgres**.

Sur les machines créées précédemment :

- installer et configurer Patroni sur **2 nœuds** en suivant le guide d'installation
- observer les traces de chaque nœud Patroni
- observer la topologie de l'agrégat avec `patronictl`.

L'installation est un peu fastidieuse ne devrait pas poser de problème.

Les binaires de PostgreSQL doivent être installés sur chaque machine.

Attention au nom du fichier de configuration attendu par le service (`/etc/patroni/config.yml` ou `/etc/patroni/patroni.yml` selon la distribution), à chercher avec ;

```
# systemctl status patroni
```

Attention aux chemins des binaires et du répertoire, qui varient évidemment en fonction de la distribution ou de vos choix.

Dans la configuration du DCS, attention aux IP ou chemins, qui doivent être accessibles depuis le nœud PostgreSQL. Le user et le mot de passe du DCS doivent bien sûr être entrés correctement (ici `root/root_password`).

Sur le nœud **pg-2**, le fichier de configuration final est (distribution Debian) :

```
scope: cluster-test-01
namespace: /service/ # valeur par défaut
name: pg-2

restapi:
  listen: 10.0.3.202:8008
  connect_address: 10.0.3.202:8008
log:
  level: INFO
  dir: /var/log/patroni/cluster-test-01
etcd:
  hosts:
    - 10.0.3.101:2379
    - 10.0.3.102:2379
    - 10.0.3.103:2379
  username: root
  password: root_password
  protocol: http
bootstrap:
  dcs:
```

```
ttl: 30
loop_wait: 10
retry_timeout: 10
maximum_lag_on_failover: 1048576
postgresql:
  use_pg_rewind: true
  use_slots: true
  parameters:
    wal_level: replica
    hot_standby: on
    max_wal_senders: 10
    max_replication_slots: 5
initdb:
- encoding: UTF8
- data-checksums
pg_hba:
- host all all all md5
- host replication replicator all md5
users:
  dba:
    password: dba_password
    options:
      - createrole
      - createdb
postgresql:
  listen: "*:5432"
  connect_address: 10.0.3.202:5432
  data_dir: /var/lib/postgresql/14/cluster-test-01
  bin_dir: /usr/lib/postgresql/14/bin
  authentication:
    replication:
      username: replicator
      password: replicator_password
    superuser:
      username: postgres
      password: postgres_password
    rewind:
      username: rewinder
      password: rewinder_password
  parameters:
    unix_socket_directories: '.'
basebackup:
  max-rate: "100M"
  checkpoint: "fast"
watchdog:
  mode: automatic
  device: /dev/watchdog
  safety_margin: 5
tags:
  nofailover: false
  noloadbalance: false
  clonefrom: false
  nosync: false
```

Une fois la configuration en place, un simple démarrage du service doit suffire :

```
# service start patroni
```

Le fichier `/var/log/patroni/cluster-test-01/patroni.log` montre alors la connexion au DCS, puis la création de zéro d'une nouvelle instance :

```
2022-04-26 08:41:51,360 INFO: Selected new etcd server http://10.0.3.101:2379
2022-04-26 08:41:51,442 INFO: No PostgreSQL configuration items changed, nothing to
  ↵ reload.
2022-04-26 08:41:51,603 INFO: Lock owner: None; I am pg-1
2022-04-26 08:41:51,772 INFO: trying to bootstrap a new cluster
2022-04-26 08:41:55,982 INFO: postmaster pid=853
2022-04-26 08:41:57,111 INFO: establishing a new patroni connection to the postgres
  ↵ cluster
2022-04-26 08:41:57,211 INFO: running post_bootstrap
2022-04-26 08:41:57,370 WARNING: Could not activate Linux watchdog device: "Can't
  ↵ open watchdog device...
2022-04-26 08:41:57,787 INFO: initialized a new cluster
2022-04-26 08:42:07,619 INFO: no action. I am (pg-1), the leader with the lock
2022-04-26 08:42:17,608 INFO: no action. I am (pg-1), the leader with the lock
```

Cette instance est visible dans les processus.

Le premier nœud est actif :

```
$ patronictl -c /etc/patroni/patroni.yml list
```

+ Cluster: cluster-test-01 (7090829996154823495) -----+					
Member	Host	Role	State	TL	Lag in MB
pg-1	10.0.3.201	Leader	running	1	

La création du second nœud se fait de la même manière sur **pg-2**.

Après son lancement, on peut voir qu'elle se raccroche au *leader* :

```
$ patronictl -c /etc/patroni/patroni.yml list
```

+ Cluster: cluster-test-01 (7090829996154823495) -----+					
Member	Host	Role	State	TL	Lag in MB
pg-1	10.0.3.201	Leader	running	1	
pg-2	10.0.3.202	Replica	stopped		unknown

puis très vite :

```
root@pg-1:~# sudo -iu postgres patronictl -c /etc/patroni/patroni.yml list

+ Cluster: cluster-test-01 (7090829996154823495) -----+
| Member | Host      | Role    | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| pg-1   | 10.0.3.201 | Leader  | running | 1  |           |
| pg-2   | 10.0.3.202 | Replica | running | 1  |          0 |
+-----+-----+-----+-----+-----+
```

Dans `/var/log/patroni/cluster-test-01/patroni.log`:

```
2022-04-26 08:54:08,441 INFO: Selected new etcd server http://10.0.3.102:2379
2022-04-26 08:54:08,523 INFO: No PostgreSQL configuration items changed, nothing to
    ↵ reload.
2022-04-26 08:54:08,681 INFO: Lock owner: pg-1; I am pg-2
2022-04-26 08:54:08,761 INFO: trying to bootstrap from leader 'pg-1'
2022-04-26 08:54:11,926 INFO: replica has been created using basebackup
2022-04-26 08:54:11,928 INFO: bootstrapped from leader 'pg-1'
2022-04-26 08:54:12,240 INFO: postmaster pid=8934
2022-04-26 08:54:13,354 INFO: Lock owner: pg-1; I am pg-2
2022-04-26 08:54:13,355 INFO: establishing a new patroni connection to the postgres
    ↵ cluster
2022-04-26 08:54:13,449 INFO: no action. I am (pg-2), a secondary, and following a
    ↵ leader (pg-1)
2022-04-26 08:54:17,639 INFO: no action. I am (pg-2), a secondary, and following a
    ↵ leader (pg-1)
2022-04-26 08:54:27,627 INFO: no action. I am (pg-2), a secondary, and following a
    ↵ leader (pg-1)
```

- Ajouter le 3ème nœud à l'agrégat.

Après l'ajout du troisième nœud :

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role     | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1  | 10.0.3.201 | Leader   | running | 1  |           |
| + pg-2 | 10.0.3.202 | Replica  | running | 1  | 0          |
| + pg-3 | 10.0.3.203 | Replica  | running | 1  | 0          |
+-----+-----+-----+-----+
```

Il est conseillé lors des tests de garder une fenêtre répétant l'ordre régulièrement :

```
$ watch -n1 patronictl -c /etc/patroni/patroni.yml topology
```

- Déterminer le primaire via l'API Patroni en direct avec curl.

La configuration est visible de l'extérieur :

```
curl -s http://pg-1:8008/patroni | jq
```

```
{
  "state": "running",
  "postmaster_start_time": "2022-04-26 08:41:55.995199+00:00",
  "role": "master",
  "server_version": 140002,
  "xlog": {
    "location": 83886176
  },
  "timeline": 1,
  "replication": [
    {
      "username": "replicator",
```

```
"application_name": "pg-2",
"client_addr": "10.0.3.202",
"state": "streaming",
"sync_state": "async",
"sync_priority": 0
},
{
  "username": "replicator",
  "application_name": "pg-3",
  "client_addr": "10.0.3.203",
  "state": "streaming",
  "sync_state": "async",
  "sync_priority": 0
}
],
"dcs_last_seen": 1650963717,
"database_system_identifier": "7090829996154823495",
"patroni": {
  "version": "2.1.3",
  "scope": "cluster-test-01"
}
}
```

De manière plus précise :

```
$ curl -s http://pg-1:8008/cluster | \
jq '.members[] | select (.role == "leader") and (.state == "running") | { name }'

{
  "name": "pg-1"
}
```

– Quels sont les slots de réPLICATION sur le *leader* ?

Par défaut, les slots de réPLICATION portent le nom des nœuds réPLICAS.

```
$ curl -s http://pg-1:8008/cluster | jq '.members[] | select (.role == "replica") | {
  name }'

{
  "name": "pg-2"
}
{
  "name": "pg-3"
}
```

On peut le vérifier en se connectant à PostgreSQL sur le serveur primaire (ici **pg-1**) :

```
psql -h localhost -c "SELECT slot_name FROM pg_replication_slots"

slot_name
-----
pg_2
pg_3
```

### 5.10.5 Patroni : utilisation

- Se connecter depuis l'extérieur en tant que dba à la base postgres, à un nœud.

La connexion à un nœud ne pose pas de problème si le nom de la cible peut être résolu par le client :

```
psql -h pg-1 -U dba -d postgres -p 5432
```

La machine cliente (ce peut être l'hôte des containers LXC) doit bien sûr posséder les outils clients.

Pour plus de facilité, renseigner le mot de passe sur le client dans ~/.pgpass, fichier qui ne doit être lisible que par son propriétaire :

```
pg-1:5432:*:dba:dba_password  
pg-2:5432:*:dba:dba_password  
pg-3:5432:*:dba:dba_password
```

En cas de problème, il faut regarder :

- les règles de *firewall* ;
- les traces du nœud PostgreSQL concerné, plus instructive que le simple message d'erreur du client :

```
journalctl -fu patroni
```

- la configuration du pg\_hba.conf dans le PGDATA de PostgreSQL.

- Comment garantir une connexion au *leader* pour écrire ?

Quand on a besoin d'écrire dans la base, il faut éviter s'adresser au nœud *leader*. La chaîne de connexion permet de préciser plusieurs nœuds, et que l'on exige une connexion en écriture, et cela sans proxy ni d'appel d'API. Il en existe deux formes :

```
psql "host=pg-1,pg-2,pg-3 port=5432 user=dba dbname=postgres  
      ↳ target_session_attrs=primary"
```

```
psql "postgresql://dba@pg-1,pg-2,pg-3:5432/postgres?target_session_attrs=primary"
```

Pour le test, le mot de passe peut être rajouté dans ces chaînes (respectivement : ajouter un paramètre password=..., ou remplacer dba par dba:dba\_password).

```
psql "postgresql://dba@pg-1,pg-2,pg-3:5432/postgres?target_session_attrs=primary"
```

Pour un client en lecture seule :

```
psql "postgresql://dba@pg-1,pg-2,pg-3:5432/postgres?target_session_attrs=prefer-  
      ↳ standby"
```

NB : les valeurs primary et prefer-standby apparaissent en PostgreSQL 14. target\_session\_attrs possède d'autres options<sup>25</sup>, à savoir : any, read-only, read-write, standby.

---

<sup>25</sup><https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-CONNECT-TARGET-SESSION-ATTRS>

- Créer une table :

```
CREATE TABLE insertions (id int GENERATED ALWAYS AS IDENTITY,
                        d timestampz DEFAULT now(),
                        source text DEFAULT inet_server_addr()
                      );
GRANT ALL ON TABLE insertions TO dba ;
```

- Insérer une ligne toutes les secondes, à chaque fois dans une nouvelle connexion au *leader*.
- Dans une autre fenêtre, afficher les 20 dernières lignes de cette table.

```
watch -n1 ' psql -X -d \
"host=pg-1,pg-2,pg-3 port=5432 user=dba dbname=postgres"
  ↵ target_session_attrs=primary" \
-c "INSERT INTO insertions SELECT ;" ' 

watch -n1 ' psql -X -d \
"host=pg-1,pg-2,pg-3 port=5432 user=dba dbname=postgres" \
-c "SELECT * FROM insertions ORDER BY d DESC LIMIT 20" '
```

Bien sûr, on voit toujours le même nœud, ici **pg-1**:

<b>id</b>	<b>d</b>	<b>source</b>
...		
114	2022-04-26 11:57:28.548761+00	10.0.3.201/32
113	2022-04-26 11:57:27.496975+00	10.0.3.201/32
112	2022-04-26 11:57:26.444883+00	10.0.3.201/32
111	2022-04-26 11:57:25.388476+00	10.0.3.201/32
110	2022-04-26 11:57:24.334427+00	10.0.3.201/32
(20 lignes)		

- Stopper le nœud *leader* Patroni.
- Que se passe-t-il dans la topologie, et dans les requêtes ci-dessus ?

Sur **pg-1**:

```
$ systemctl stop patroni
```

**pg-3** prend ici le rôle de *leader*:

```
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role     | State    | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-3   | 10.0.3.203 | Leader   | running  | 4  |           |
| + pg-1 | 10.0.3.201 | Replica  | stopped  |     | unknown    |
| + pg-2 | 10.0.3.202 | Replica  | running  | 4  | 0          |
+-----+-----+-----+-----+
```

Les insertions échouent le temps de la bascule puis continuent depuis l'autre nœud :

<b>id</b>	<b>d</b>	<b>source</b>
...		

```
...
365 | 2022-04-26 12:36:09.411084+00 | 10.0.3.203/32
364 | 2022-04-26 12:36:08.325285+00 | 10.0.3.203/32
363 | 2022-04-26 12:36:07.243333+00 | 10.0.3.203/32
362 | 2022-04-26 12:36:06.165184+00 | 10.0.3.203/32
361 | 2022-04-26 12:36:05.093206+00 | 10.0.3.203/32
360 | 2022-04-26 12:36:04.010203+00 | 10.0.3.203/32
359 | 2022-04-26 12:36:02.931134+00 | 10.0.3.203/32
(20 lignes)
```

- Arrêter à nouveau le *leader*. Il doit donc rester un nœud actif. Que se passe-t-il ?

Le *leader* Patroni a changé à nouveau :

```
# killall -u postgres
$ patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State   | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-2  | 10.0.3.202 | Leader  | running | 5  |           |
+-----+-----+-----+-----+
```

- Arrêter deux nœuds du cluster etcd. Que se passe-t-il ?

Sur **e1** et **e2**:

```
# systemctl stop etcd
```

Il n'y a plus de quorum etcd garantissant une référence. Le cluster Patroni se met donc en lecture seule et les insertions tombent en échec puisqu'elles exigent une connexion ouverte en écriture :

```
psql: erreur : la connexion au serveur sur « pg-1 » (10.0.3.201), port 5432 a échoué
  ↵ : Connexion refusée
      Le serveur est-il actif sur cet hôte et accepte-t-il les connexions ?
la connexion au serveur sur « pg-1 » (10.0.3.201), port 5432 a échoué : Connexion
  ↵ refusée
      Le serveur est-il actif sur cet hôte et accepte-t-il les connexions ?
la connexion au serveur sur « pg-2 » (10.0.3.202), port 5432 a échoué : la session
  ↵ est en lecture seule
la connexion au serveur sur « pg-3 » (10.0.3.203), port 5432 a échoué : Connexion
  ↵ refusée
      Le serveur est-il actif sur cet hôte et accepte-t-il les connexions ?
la connexion au serveur sur « pg-3 » (10.0.3.203), port 5432 a échoué : Connexion
  ↵ refusée
      Le serveur est-il actif sur cet hôte et accepte-t-il les connexions ?
```

- Redémarrer les nœuds etcd.

Les écritures reprennent.

- Relancer un des nœuds Patroni.

**pg-2** est ici le *leader*, et on redémarrer **pg-1**:

```
# systemctl restart patroni
```

Le raccrochage se fait:

```
patronictl -c /etc/patroni/patroni.yml topology
```

+ Cluster: cluster-test-01 (7090829996154823495) -----+					
Member	Host	Role	State	TL	Lag in MB
pg-2	10.0.3.202	Leader	running	6	
+ pg-1	10.0.3.201	Replica	running	6	0

- Sur le troisième nœud (arrêté), détruire le PGDATA. Relancer Patroni.

Pour un chemin sur une machine Debian, la commande est :

```
$ rm -rf /var/lib/postgresql/14/cluster-test-01
```

Le nœud est relancé :

```
# systemctl start patroni
```

et se raccroche :

```
patroni[42738]: 2022-04-26 14:59:11.701 UTC [42765] LOG: démarrage de PostgreSQL
  ↵ 14.2 (Debian 14.2-1...
patroni[42738]: 2022-04-26 14:59:11.702 UTC [42765] LOG: en écoute sur IPv4,
  ↵ adresse « 0.0.0.0 », port 5432
patroni[42738]: 2022-04-26 14:59:11.702 UTC [42765] LOG: en écoute sur IPv6,
  ↵ adresse « :: », port 5432
patroni[42738]: 2022-04-26 14:59:11.718 UTC [42765] LOG: écoute sur la socket Unix
  ↵ « ./s.PGSQL.5432 »
patroni[42738]: 2022-04-26 14:59:11.734 UTC [42768] LOG: le système de bases de
  ↵ données a été interrompu ...
patroni[42738]: 2022-04-26 14:59:11.816 UTC [42768] LOG: entre en mode standby
patroni[42738]: 2022-04-26 14:59:11.845 UTC [42768] LOG: la ré-exécution commence à
  ↵ 0/F000028
patroni[42738]: 2022-04-26 14:59:11.853 UTC [42768] LOG: état de restauration
  ↵ cohérent atteint à 0/F001830
patroni[42738]: 2022-04-26 14:59:11.853 UTC [42765] LOG: le système de bases de
  ↵ données est prêt ...
patroni[42738]: 2022-04-26 14:59:11.869 UTC [42772] LOG: démarré le flux des
  ↵ journaux depuis le principal...
patroni[42738]: localhost:5432 - acceptation des connexions
patroni[42738]: localhost:5432 - acceptation des connexions
```

Le temps de la reconstruction de zéro, on peut voir l'état transitoire :

```
patronictl -c /etc/patroni/patroni.yml topology
```

```
patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-2   | 10.0.3.202 | Leader  | running     | 6  |
| + pg-1 | 10.0.3.201 | Replica | running     | 6  | 0
| + pg-3 | 10.0.3.203 | Replica | creating replica | | unknown
+-----+-----+-----+-----+
```

avant d'arriver à un nouvel état stable :

```
patronictl -c /etc/patroni/patroni.yml topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-2   | 10.0.3.202 | Leader  | running     | 6  |
| + pg-1 | 10.0.3.201 | Replica | running     | 6  | 0
| + pg-3 | 10.0.3.203 | Replica | running     | 6  | 0
+-----+-----+-----+-----+
```

- Forcer un *failover* vers le nœud pg-1.

```
$ patronictl -c /etc/patroni/patroni.yml failover

Candidate ['pg-1', 'pg-3'] []: pg-1
Current cluster topology
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1   | 10.0.3.201 | Replica | running     | 6  | 0
| pg-2   | 10.0.3.202 | Leader  | running     | 6  |
| pg-3   | 10.0.3.203 | Replica | running     | 6  | 0
+-----+-----+-----+-----+
Are you sure you want to failover cluster cluster-test-01, demoting current master
↪ pg-2? [y/N]: y
2022-04-26 14:30:11.43574 Successfully failed over to "pg-1"
+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1   | 10.0.3.201 | Leader  | running     | 6  |
| pg-2   | 10.0.3.202 | Replica | stopped    | | unknown
| pg-3   | 10.0.3.203 | Replica | running     | 6  | 0
+-----+-----+-----+-----+
$ patronictl -c /etc/patroni/patroni.yml failover

+ Cluster: cluster-test-01 (7090829996154823495) -----
| Member | Host      | Role    | State       | TL | Lag in MB |
+-----+-----+-----+-----+-----+
| pg-1   | 10.0.3.201 | Leader  | running     | 7  |
| + pg-2 | 10.0.3.202 | Replica | running     | 7  | 0
| + pg-3 | 10.0.3.203 | Replica | running     | 6  | 0
+-----+-----+-----+-----+
$ patronictl -c /etc/patroni/patroni.yml failover
```

+ Cluster: cluster-test-01 (7090829996154823495) -----+					
Member	Host	Role	State	TL	Lag in MB
pg-1	10.0.3.201	Leader	running	7	
+ pg-2	10.0.3.202	Replica	running	7	0
+ pg-3	10.0.3.203	Replica	running	7	0

- Modifier les paramètres `shared_buffers` et `work_mem`. Si besoin, redémarrer les nœuds.

Ici nous passerons par `patronictl`, plutôt que modifier directement les fichiers de configuration. (Une alternative est de modifier la configuration statique, donc le fichier YAML. Cela facilite leur maintenance, mais impose que le contenu reste identique entre les nœuds, ce qui est généralement le cas dans un déploiement industrialisé.)

`patronictl` appellera l'éditeur par défaut, souvent `vi`.

```
$ patronictl -c /etc/patroni/patroni.yml edit-config  
+++  
@@ -6,6 +6,8 @@  
    max_replication_slots: 5  
    max_wal_senders: 10  
    wal_level: replica  
+    shared_buffers: 300MB  
+    work_mem: 50MB  
    use_pg_rewind: true  
    use_slots: true  
    retry_timeout: 10  
  
Apply these changes? [y/N]: y  
Configuration changed
```

Les nœuds sont à redémarrer à cause de la modification de `shared_buffers`:

```
$ patronictl -c /etc/patroni/patroni.yml topology  
Cluster: cluster-test-01 (7090829996154823495) -----+  
| Member | Host      | Role     | State    | TL | Lag in MB | Pending restart |  
+-----+-----+-----+-----+-----+-----+-----+  
| pg-2  | 10.0.3.202 | Leader   | running  | 8  |           | *             |  
| + pg-1 | 10.0.3.201 | Replica  | running  | 8  |           | 0             |  
| + pg-3 | 10.0.3.203 | Replica  | running  | 8  |           | 0             |  
+-----+-----+-----+-----+-----+-----+-----+  
  
$ patronictl -c /etc/patroni/patroni.yml restart cluster-test-01  
+ Cluster: cluster-test-01 (7090829996154823495) -----+  
| Member | Host      | Role     | State    | TL | Lag in MB | Pending restart |  
+-----+-----+-----+-----+-----+-----+-----+  
| pg-1  | 10.0.3.201 | Replica  | running  | 8  |           | 0             | *  
| pg-2  | 10.0.3.202 | Leader   | running  | 8  |           | *             |  
| pg-3  | 10.0.3.203 | Replica  | running  | 8  |           | 0             | *  
+-----+-----+-----+-----+-----+-----+-----+
```

```
When should the restart take place (e.g. 2022-04-26T16:13) [now]:  
Are you sure you want to restart members pg-2, pg-1, pg-3? [y/N]: y  
Restart if the PostgreSQL version is less than provided (e.g. 9.5.2) []: 14.9  
Success: restart on member pg-2  
Success: restart on member pg-1  
Success: restart on member pg-3  
  
postgres=> SHOW shared_buffers ;  
  
shared_buffers  
-----  
300MB
```

Noter que le contenu des modifications est tracé dans un fichier `patroni.dynamic.json` dans le PGDATA.

## 5.11 INTRODUCTION À PGBENCH

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 2017<sup>26</sup>.

### 5.11.1 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   |
```

### 5.11.2 Générer de l'activité

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

<sup>26</sup>[https://youtu.be/aTwh\\_CgRaE0](https://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.

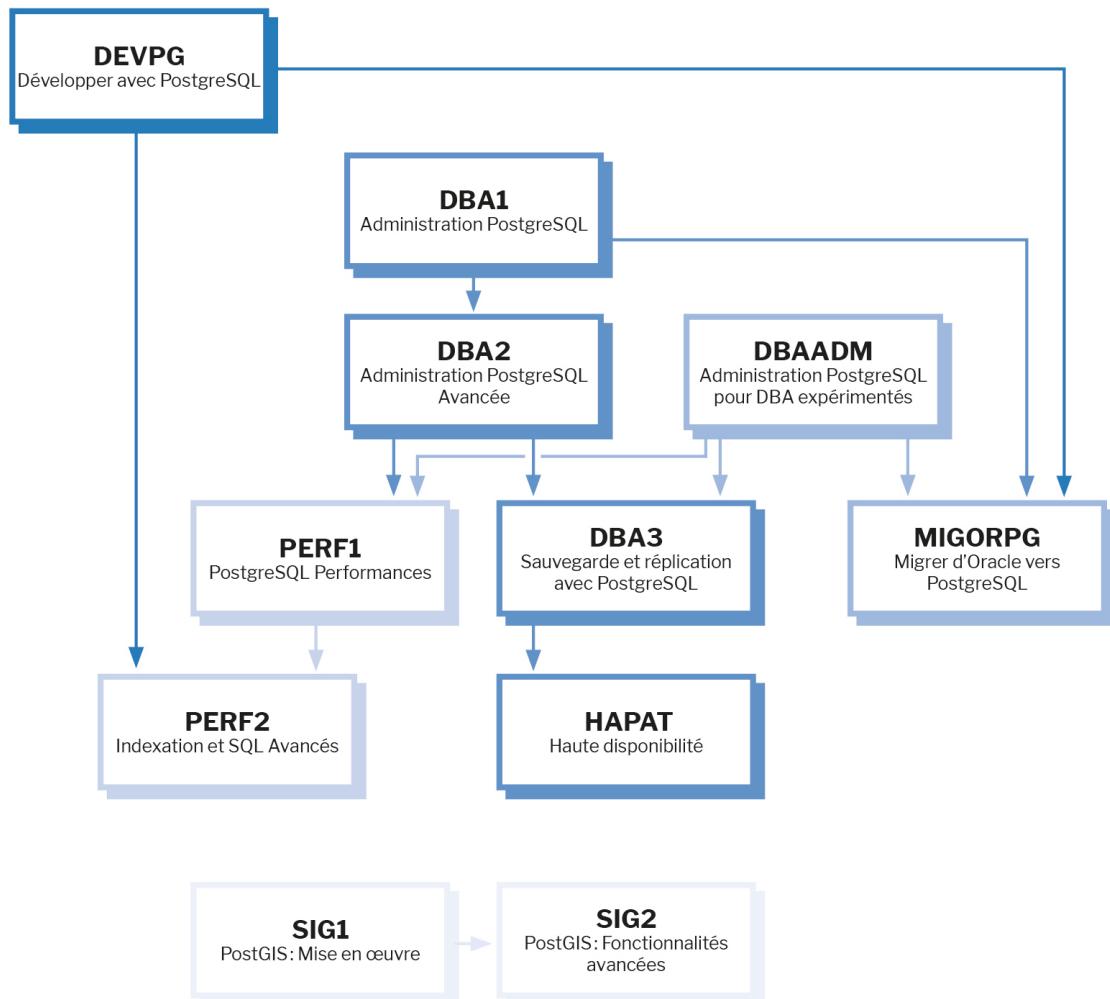


# Les formations Dalibo

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.



