Module R50

Généralités sur la Haute Disponibilité

24.04

Table des matières

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

1/ Généralités sur la haute disponibilité **5**
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 des données . . . 8
1.3 Sauvegardes . . . 10
1.3.1 Sauvegarde 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 *fencing* . . . 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
1.7 Questions . . . 24

Les formations Dalibo **25**
Cursus des formations . . . 25
Les livres blancs . . . 26
Téléchargement gratuit . . . 26

Sur ce document

Formation	Module R50
Titre	Généralités sur la Haute Disponibilité
Révision	24.04
PDF	https://dali.bo/r50_pdf
EPUB	https://dali.bo/r50_epub
HTML	https://dali.bo/r50_html
Slides	https://dali.bo/r50_slides

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**[2]. 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

[2]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 et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

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

Versions de PostgreSQL couvertes

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

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

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

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

1/ Généralités sur la haute disponibilité

1.1 INTRODUCTION

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

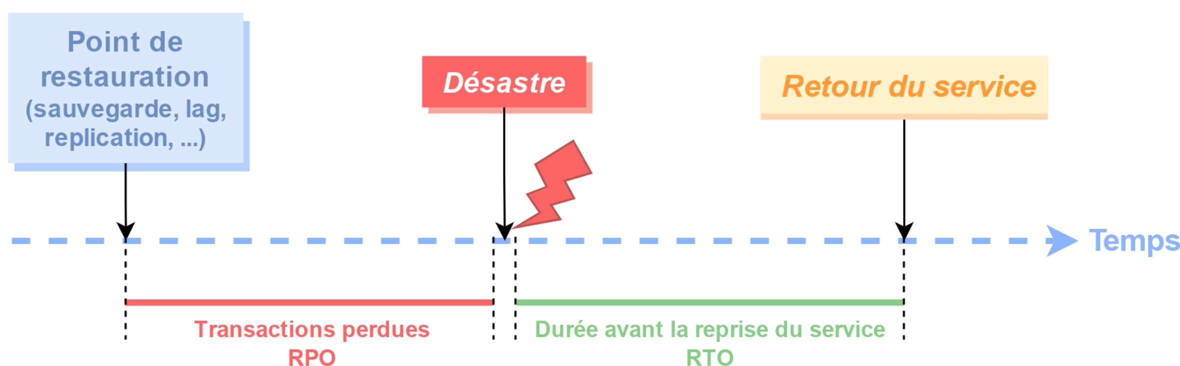
La haute disponibilité est un sujet complexe. Plusieurs outils libres coexistent au sein de l'écosystème PostgreSQL, chacun abordant le sujet d'une façon différente.

Ce document clarifie la définition de « haute disponibilité », des méthodes existantes et des contraintes à considérer. L'objectif est d'aider à 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 objectives*) 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…
  - réplication des données
- 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 redonder 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 des données

- Perte faible ou nulle de données après incident
  - redonder les données
  - garantir les écritures à plusieurs endroits
- Contradictoire avec la haute disponibilité de service
  - arbitrage possible avec une complexité et un budget plus importants

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
- Et ne pas oublier de tester

Les différentes méthodes de sauvegardes de PostgreSQL (logique avec `pg_dump`, ou PITR avec pgBackRest ou d'autres outils) sont souvent sous-estimées. Elles sont pourtant un élément essentiel de toute architecture qui est souvent déjà présent. On n'oubliera d'ailleurs pas de tester régulièrement ses sauvegardes[1].

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 Sauvegarde PITR

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

La sauvegarde PITR (*Point In Time Recovery*) 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.

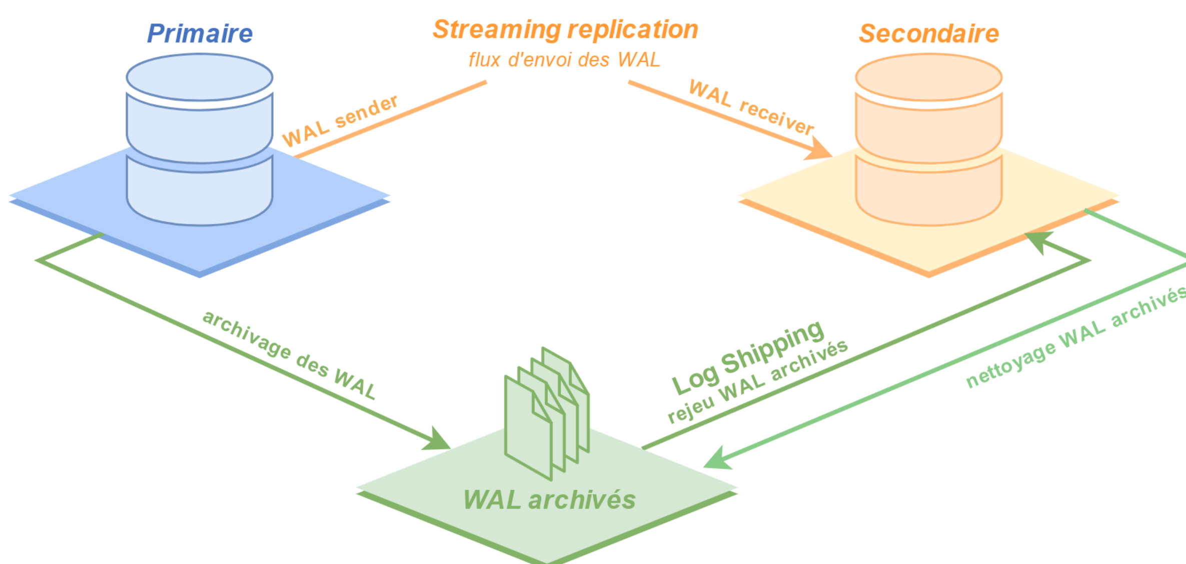
Avec le bon matériel, les bonnes pratiques et une politique de sauvegarde adaptée, il est possible d'atteindre un RTO de quelques minutes, dans la plupart des cas.

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

[1] https://blog.dalibo.com/2024/03/29/world_backup_day_restauration.html

la réplication en *streaming*, soit vers une instance secondaire, soit avec l'outil `pg_receivewal` (pour la sauvegarde uniquement). Nous abordons ce sujet dans un futur chapitre.

1.3.2 PITR et redondance par réplication physique



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[2].

[2] 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 et un RTO plus faibles que celui d'une simple sauvegarde PITR, au prix d'un investissement plus important (matériel redondant), d'une prix d'investissement plus important (redondance du matériel), d'une complexification de l'architecture et de sa maintenance. Le RTO 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 zéro. 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`) ou lors d'écritures en masse de données.

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 et d'analyse 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, ce qui est facilement automatisé.

1.4.3 Bilan sur la réplication

- $0 \leq$ RPO $<$ PITR
- RTO = 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 par les 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 lui 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[3], ou le PDU[4] ; soit éteignent la machine au niveau matériel via l'IPMI[5] ; soit éteignent la machine virtuelle virtuelle brusquement 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'identifiant, 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, elle pourrait ne pas se raccrocher correctement au nouveau primaire, voire corrompre ses propres fichiers de données. En effet, il est impossible de déterminer quelles écritures ont eu lieu sur cet ancien primaire entre sa déconnexion du reste du cluster et 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.

[3] https://fr.wikipedia.org/wiki/Alimentation_sans_interruption
[4] https://fr.wikipedia.org/wiki/Unit%C3%A9_de_distribution_d%27%C3%A9nergie
[5] https://fr.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

Patroni repose sur un DCS[6] extérieur, par exemple etcd[7], 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
- À ré-armer par un composant applicatif du serveur, **périodiquement**
- Permet de déclencher du *self-fencing* rapide et fiable
  - meilleure réactivité de l'agrégat
- « Fencing du pauvre », complémentaire du quorum
- 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[8] entre plus en détails.

[6] *Distributed Control System*
[7] https://etcd.io/
[8] 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[9].

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. Attention aussi au stockage partagé 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

[9] https://documentation.suse.com/sle-ha/15-SP4/html/SLE-HA-all/cha-ha-storage-protect.html

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 (par exemple `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
  - et tout doit passer à présent par lui
- Complexification de l'administration
  - Formation + Tests + Documentation + Communication
  - sinon : erreurs humaines plus fréquentes
  - au final : est-ce plus fiable ?
- Opérations post-bascule toujours à faire

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 (en reconstruisant un nœud, par exemple).

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.

1.7 QUESTIONS

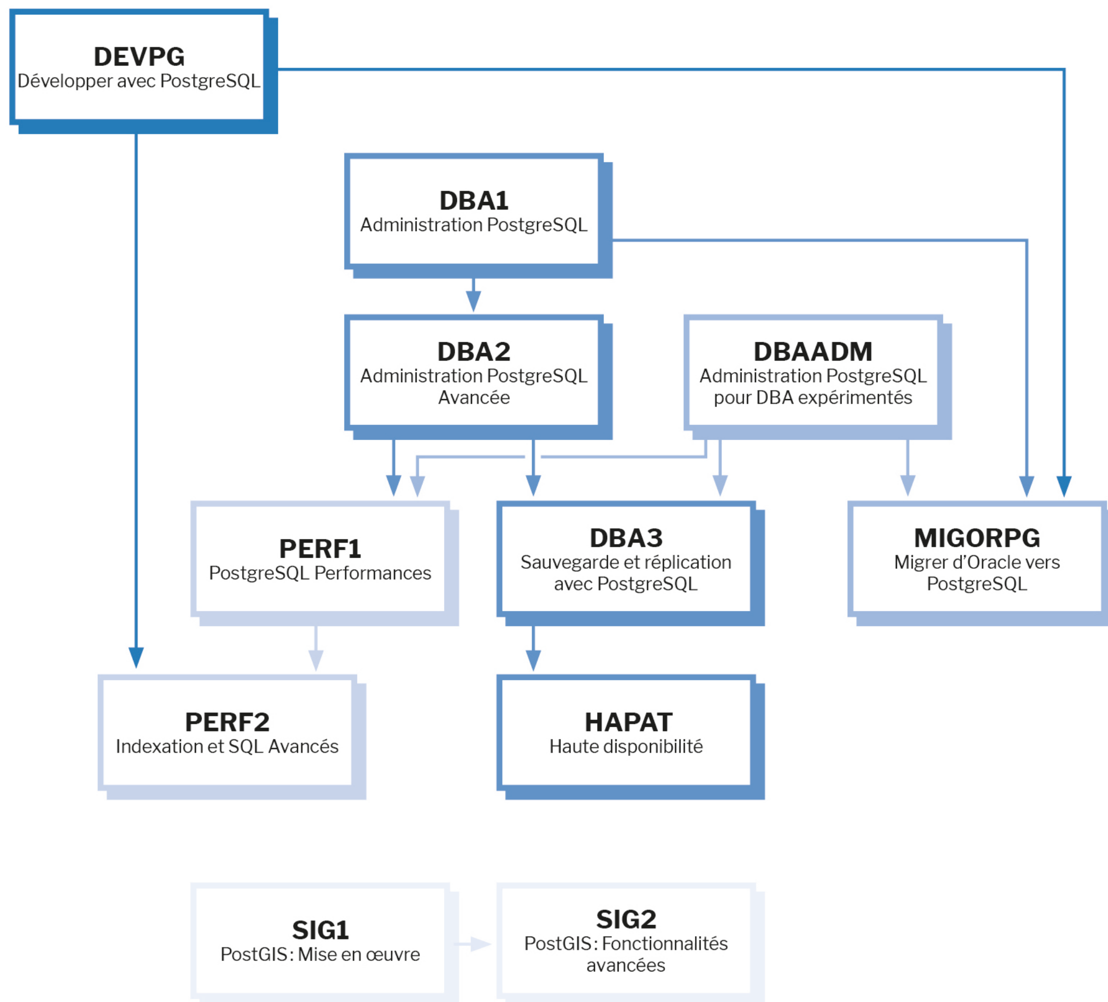
N'hésitez pas, c'est le moment !

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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