Module I5

Gestion d'un sinistre

DALIBO
L'expertise PostgreSQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Gestion d'un sinistre

Module I5

TITRE : Gestion d'un sinistre
SOUS-TITRE : Module I5

REVISION: 22.09
DATE: 02 septembre 2022

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, 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, 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.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé à :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale : Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

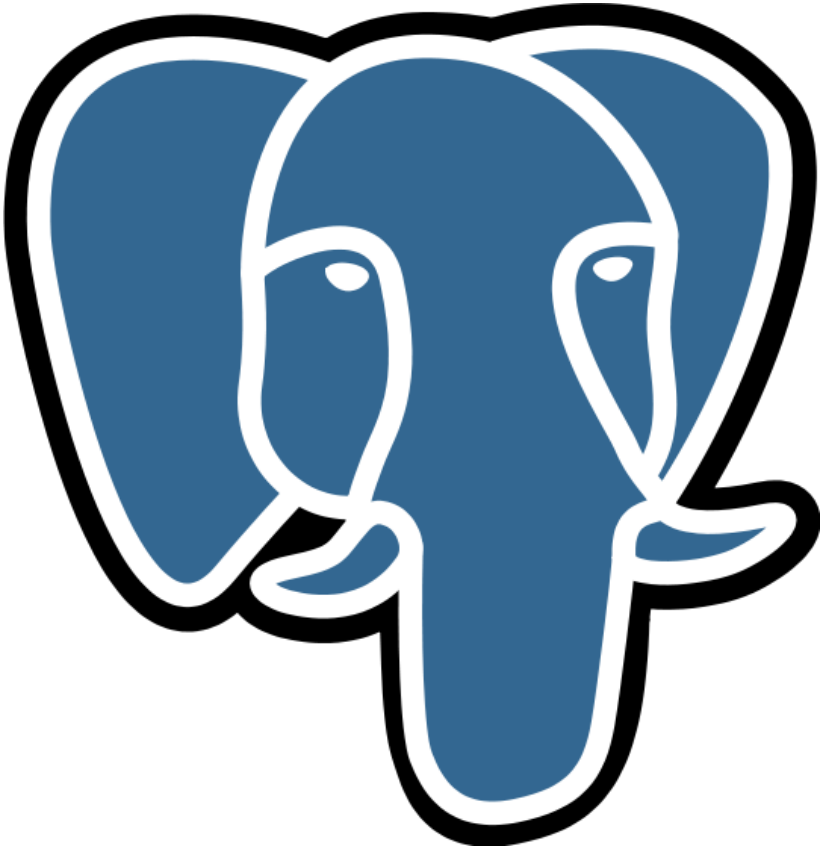
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

Table des Matières

1 POSTGRESQL : GESTION D'UN SINISTRE



1.1 INTRODUCTION

- Une bonne politique de sauvegardes est cruciale
  - mais elle n'empêche pas les incidents
- Il faut être prêt à y faire face

Ce module se propose de faire une description des bonnes et mauvaises pratiques en cas de coup dur :

- crash de l'instance ;
- suppression / corruption de fichiers ;
- problèmes matériels ;
- sauvegardes corrompues…

Seront également présentées les situations classiques de désastres, ainsi que certaines méthodes et outils dangereux et déconseillés.

L'objectif est d'aider à convaincre de l'intérêt qu'il y a à anticiper les problèmes, à mettre en place une politique de sauvegarde pérenne, et à ne pas tenter de manipulation dangereuse sans comprendre précisément à quoi l'on s'expose.

Ce module est en grande partie inspiré de *The Worst Day of Your Life*, une présentation de Christophe Pettus au FOSDEM 2014[2]

1.1.1 AU MENU

- Anticiper les désastres
- Réagir aux désastres
- Rechercher l'origine du problème
- Outils utiles
- Cas type de désastres

1.2 ANTICIPER LES DÉSASTRES

- Un désastre peut toujours survenir
- Il faut savoir le détecter le plus tôt possible
  - et s'être préparé à y répondre

Il est impossible de parer à tous les cas de désastres imaginables.

Le matériel peut subir des pannes, une faille logicielle non connue peut être exploitée, une modification d'infrastructure ou de configuration peut avoir des conséquences imprévues à long terme, une erreur humaine est toujours possible.

Les principes de base de la haute disponibilité (redondance, surveillance…) permettent de mitiger le problème, mais jamais de l'éliminer complètement.

[2] http://thebuild.com/presentations/worst-day-fosdem-2014.pdf

Il est donc extrêmement important de se préparer au mieux, de procéder à des simulations, de remettre en question chaque brique de l'infrastructure pour être capable de détecter une défaillance et d'y réagir rapidement.

1.2.1 DOCUMENTATION

- Documentation complète et à jour
  - emplacement et fréquence des sauvegardes
  - emplacement des traces
  - procédures et scripts d'exploitation
- Sauvegarder et versionner la documentation

Par nature, les désastres arrivent de façon inattendue.

Il faut donc se préparer à devoir agir en urgence, sans préparation, dans un environnement perturbé et stressant — par exemple, en pleine nuit, la veille d'un jour particulièrement critique pour l'activité de la production.

Un des premiers points d'importance est donc de s'assurer de la présence d'une documentation claire, précise et à jour, afin de minimiser le risque d'erreurs humaines.

Cette documentation devrait détailler l'architecture dans son ensemble, et particulièrement la politique de sauvegarde choisie, l'emplacement de celles-ci, les procédures de restauration et éventuellement de bascule vers un environnement de secours.

Les procédures d'exploitation doivent y être expliquées, de façon détaillée mais claire, afin qu'il n'y ait pas de doute sur les actions à effectuer une fois la cause du problème identifié.

La méthode d'accès aux informations utiles (traces de l'instance, du système, supervision…) devrait également être soigneusement documentée afin que le diagnostic du problème soit aussi simple que possible.

Toutes ces informations doivent être organisées de façon claire, afin qu'elles soient immédiatement accessibles et exploitables aux intervenants lors d'un problème.

Il est évidemment tout aussi important de penser à versionner et sauvegarder cette documentation, afin que celle-ci soit toujours accessible même en cas de désastre majeur (perte d'un site).

1.2.2 PROCÉDURES ET SCRIPTS

- Procédures détaillées de restauration / PRA
  - préparer des scripts / utiliser des outils
  - minimiser le nombre d'actions manuelles
- Tester les procédures régulièrement
  - bases de test, développement...
  - s'assurer que chacun les maîtrise
- Sauvegarder et versionner les scripts

La gestion d'un désastre est une situation particulièrement stressante, le risque d'erreur humaine est donc accru.

Un DBA devant restaurer d'urgence l'instance de production en pleine nuit courera plus de risques de faire une fausse manipulation s'il doit taper une vingtaine de commandes en suivant une procédure dans une autre fenêtre (voire un autre poste) que s'il n'a qu'un script à exécuter.

En conséquence, il est important de minimiser le nombre d'actions manuelles à effectuer dans les procédures, en privilégiant l'usage de scripts d'exploitation ou d'outils dédiés (comme pitrery, pgBackRest ou barman pour restaurer une instance PostgreSQL).

Néanmoins, même cette pratique ne suffit pas à exclure tout risque.

L'utilisation de ces scripts ou de ces outils doit également être comprise, correctement documentée, et les procédures régulièrement testées. Le test idéal consiste à remonter fréquemment des environnements de développement et de test ; vos développeurs vous en seront d'ailleurs reconnaissants.

Dans le cas contraire, l'utilisation d'un script ou d'un outil peut aggraver le problème, parfois de façon dramatique — par exemple, l'écrasement d'un environnement sain lors d'une restauration parce que la procédure ne mentionne pas que le script doit être lancé depuis un serveur particulier.

L'aspect le plus important est de s'assurer par des tests réguliers **et manuels** que les procédures sont à jour, n'ont pas de comportement inattendu, et sont maîtrisées par toute l'équipe d'exploitation.

Tout comme pour la documentation, les scripts d'exploitation doivent également être sauvegardés et versionnés.

1.2.3 SUPERVISION ET HISTORISATION

- Tout doit être supervisé
  - réseau, matériel, système, logiciels…
  - les niveaux d'alerte doivent être significatifs
- Les métriques importantes doivent être historisées
  - cela permet de retrouver le moment où le problème est apparu
  - quand cela a un sens, faire des graphes

La supervision est un sujet vaste, qui touche plus au domaine de la haute disponibilité.

Un désastre sera d'autant plus difficile à gérer qu'il est détecté tard. La supervision en place doit donc être pensée pour détecter tout type de défaillance (penser également à superviser la supervision !).

Attention à bien calibrer les niveaux d'alerte, la présence de trop de messages augmente le risque que l'un d'eux passe inaperçu, et donc que l'incident ne soit détecté que tardivement.

Pour aider la phase de diagnostic de l'origine du problème, il faut prévoir d'historiser un maximum d'informations.

La présentation de celles-ci est également importante : il est plus facile de distinguer un pic brutal du nombre de connexions sur un graphique que dans un fichier de traces de plusieurs Go !

1.2.4 AUTOMATISATION

- Des outils existent
  - PAF (Pacemaker), patroni, repmgr…
- Automatiser une bascule est complexe
  - cela peut mener à davantage d'incidents
  - voire à des désastres (*split brain*)

Si on poursuit jusqu'au bout le raisonnement précédent sur le risque à faire effectuer de nombreuses opérations manuelles lors d'un incident, la conclusion logique est que la solution idéale serait de les éliminer complètement, et d'automatiser complètement le déclenchement et l'exécution de la procédure.

Un problème est que toute solution visant à automatiser une tâche se base sur un nombre limité de paramètres et sur une vision restreinte de l'architecture.

De plus, il est difficile à un outil de bascule automatique de diagnostiquer correctement certains types d'incident, par exemple une partition réseau. L'outil peut donc détecter à tort à un incident, surtout s'il est réglé de façon à être assez sensible, et ainsi provoquer lui-même une coupure de service inutile.

Dans le pire des cas, l'outil peut être amené à prendre une mauvaise décision amenant à une situation de désastre, comme un *split brain* (deux instances PostgreSQL se retrouvent ouvertes en écriture en même temps sur les mêmes données).

Il est donc fortement préférable de laisser un administrateur prendre les décisions potentiellement dangereuses, comme une bascule ou une restauration.

1.3 RÉAGIR AUX DÉSASTRES

- Savoir identifier un problème majeur
- Bons réflexes
- Mauvais réflexes

En dépit de toutes les précautions que l'on peut être amené à prendre, rien ne peut garantir qu'aucun problème ne surviendra.

Il faut donc être capable d'identifier le problème lorsqu'il survient, et être prêt à y répondre.

1.3.1 SYMPTÔMES D'UN DÉSASTRE

- Crash de l'instance
- Résultats de requêtes erronnés
- Messages d'erreurs dans les traces
- Dégradation importante des temps d'exécution
- Processus manquants
  - ou en court d'exécution depuis trop longtemps

De très nombreux éléments peuvent aider à identifier que l'on est en situation d'incident grave.

Le plus flagrant est évidemment le crash complet de l'instance PostgreSQL, ou du serveur l'hébergeant, et l'impossibilité pour PostgreSQL de redémarrer.

Les désastres les plus importants ne sont toutefois pas toujours aussi simples à détecter.

Les crash peuvent se produire uniquement de façon ponctuelle, et il existe des cas où l'instance redémarre immédiatement après (typiquement suite au `kill -9` d'un processus backend PostgreSQL).

Cas encore plus délicat, il peut également arriver que les résultats de requêtes soient erronés (par exemple en cas de corruption de fichiers d'index) sans qu'aucune erreur n'apparaisse.

Les symptômes classiques permettant de détecter un problème majeur sont :

- la présence de messages d'erreurs dans les traces de PostgreSQL (notamment des messages `PANIC` ou `FATAL`, mais les messages `ERROR` et `WARNING` sont également très significatifs, particulièrement s'ils apparaissent soudainement en très grand nombre) ;
- la présence de messages d'erreurs dans les traces du système d'exploitation (notamment concernant la mémoire ou le système de stockage) ;
- le constat d'une dégradation importante des temps d'exécution des requêtes sur l'instance ;
- l'absence de certains processus critiques de PostgreSQL ;
- la présence de processus présents depuis une durée inhabituelle (plusieurs semaines, mois…).

1.3.2 BONS RÉFLEXES 1

- Garder la tête froide
- Répartir les tâches clairement
- Minimiser les canaux de communication
- Garder des notes de chaque action entreprise

Une fois que l'incident est repéré, il est important de ne pas foncer tête baissée dans des manipulations.

Il faut bien sûr prendre en considération la criticité du problème, notamment pour définir la priorité des actions (par exemple, en cas de perte totale d'un site, quelles sont les applications à basculer en priorité ?), mais quelle que soit la criticité ou l'impact, il ne faut jamais effectuer une action sans en avoir parfaitement saisi l'impact et s'être assuré qu'elle répondait bien au problème rencontré.

Si le travail s'effectue en équipe, il faut bien faire attention à répartir les tâches clairement, afin d'éviter des manipulations concurrentes ou des oublis qui pourraient aggraver la situation.

Il faut également éviter de multiplier les canaux de communication, cela risque de favoriser la perte d'information, ce qui est critique dans une situation de crise.

Surtout, une règle majeure est de prendre le temps de noter systématiquement toutes les actions entreprises.

Les commandes passées, les options utilisées, l'heure d'exécution, toutes ces informations sont très importantes, déjà pour pouvoir agir efficacement en cas de fausse manipulation, mais également pour documenter la gestion de l'incident après coup, et ainsi en conserver une trace qui sera précieuse si celui-ci venait à se reproduire.

1.3.3 BONS RÉFLEXES 2

- Se prémunir contre une aggravation du problème
  - couper les accès applicatifs
- Si une corruption est suspectée
  - arrêter immédiatement l'instance
  - faire une sauvegarde immédiate des fichiers
  - travailler sur une copie

S'il y a suspicion de potentielle corruption de données, il est primordial de s'assurer au plus vite de couper tous les accès applicatifs vers l'instance afin de ne pas aggraver la situation.

Il est généralement préférable d'avoir une coupure de service plutôt qu'un grand volume de données irrécupérables.

Ensuite, il faut impérativement faire une sauvegarde complète de l'instance avant de procéder à toute manipulation. En fonction de la nature du problème rencontré, le type de sauvegarde pouvant être effectué peut varier (un export de données ne sera possible que si l'instance est démarrée et que les fichiers sont lisibles par exemple). En cas de doute, la sauvegarde la plus fiable qu'il est possible d'effectuer est une copie des fichiers à froid (instance arrêtée) - toute autre action (y compris un export de données) pourrait avoir des conséquences indésirables.

Si des manipulations doivent être tentées pour tenter de récupérer des données, il faut impérativement travailler sur une copie de l'instance, restaurée à partir de cette sauvegarde. Ne jamais travailler directement sur une instance de production corrompue, la moindre action (même en lecture) pourrait aggraver le problème !

Pour plus d'information, voir sur le wiki PostgreSQL[3] .

[3] https://wiki.postgresql.org/wiki/Corruption

1.3.4 BONS RÉFLEXES 3

- Déterminer le moment de démarrage du désastre
- Adopter une vision générale plutôt que focalisée sur un détail
- Remettre en cause chaque élément de l'architecture
  - aussi stable (et/ou coûteux/complexe) soit-il
- Éliminer en priorité les causes possibles côté hardware, système
- Isoler le comportement précis du problème
  - identifier les requêtes / tables / index impliqués

La première chose à identifier est l'instant précis où le problème a commencé à se manifester. Cette information est en effet déterminante pour identifier la cause du problème, et le résoudre — notamment pour savoir à quel instant il faut restaurer l'instance si cela est nécessaire.

Il convient pour cela d'utiliser les outils de supervision et de traces (système, applicatif et PostgreSQL) pour remonter au moment d'apparition des premiers symptômes. Attention toutefois à ne pas confondre les symptômes avec le problème lui-même ! Les symptômes les plus visibles ne sont pas forcément apparus les premiers. Par exemple, la charge sur la machine est un symptôme, mais n'est jamais la cause du problème. Elle est liée à d'autres phénomènes, comme des problèmes avec les disques ou un grand nombre de connexions, qui peuvent avoir commencé à se manifester bien avant que la charge ne commence réellement à augmenter.

Si la nature du problème n'est pas évidente à ce stade, il faut examiner l'ensemble de l'architecture en cause, sans en exclure d'office certains composants (baie de stockage, progiciel…), quels que soient leur complexité / coût / stabilité supposés. Si le comportement observé côté PostgreSQL est difficile à expliquer (crashs plus ou moins aléatoires, nombreux messages d'erreur sans lien apparent…), il est préférable de commencer par s'assurer qu'il n'y a pas un problème de plus grande ampleur (système de stockage, virtualisation, réseau, système d'exploitation).

Un bon indicateur consiste à regarder si d'autres instances / applications / processus rencontrent des problèmes similaires.

Ensuite, une fois que l'ampleur du problème a été cernée, il faut procéder méthodiquement pour en déterminer la cause et les éléments affectés.

Pour cela, les informations les plus utiles se trouvent dans les traces, généralement de PostgreSQL ou du système, qui vont permettre d'identifier précisément les éventuels fichiers ou relations corrompus.

1.3.5 BONS RÉFLEXES 4

- En cas de défaillance matérielle
  - s'assurer de corriger sur du hardware sain et non affecté !
  - baies partagées...

Cette recommandation peut paraître aller de soi, mais si les problèmes sont provoqués par une défaillance matérielle, il est impératif de s'assurer que le travail de correction soit effectué sur un environnement non affecté.

Cela peut s'avérer problématique dans le cadre d'architecture mutualisant les ressources, comme des environnements virtualisés ou utilisant une baie de stockage.

Prendre également la précaution de vérifier que l'intégrité des sauvegardes n'est pas affectée par le problème.

1.3.6 BONS RÉFLEXES 5

- Communiquer, ne pas rester isolé
- Demander de l'aide si le problème est trop complexe
  - autres équipes
  - support
  - forums
  - listes

La communication est très importante dans la gestion d'un désastre.

Il est préférable de minimiser le nombre de canaux de communication plutôt que de les multiplier (téléphone, e-mail, chat, ticket...), ce qui pourrait amener à une perte d'informations et à des délais indésirables.

Il est primordial de rapidement cerner l'ampleur du problème, et pour cela il est généralement nécessaire de demander l'expertise d'autres administrateurs / équipes (applicatif, système, réseau, virtualisation, SAN...). Il ne faut pas rester isolé et risquer que la vision étroite que l'on a des symptômes (notamment en terme de supervision / accès aux traces) empêche l'identification de la nature réelle du problème.

Si la situation semble échapper à tout contrôle, et dépasser les compétences de l'équipe en cours d'intervention, il faut chercher de l'aide auprès de personnes compétentes, par exemple auprès d'autres équipes, du support.

En aucun cas, il ne faut se mettre à suivre des recommandations glanées sur Internet, qui ne se rapporteraient que très approximativement au problème rencontré, voire pas du tout. Si nécessaire, on trouve en ligne des forums et des listes de discussions spécialisées sur lesquels il est également possible d'obtenir des conseils — il est néanmoins indispensable de prendre en compte que les personnes intervenant sur ces médias le font de manière bénévole. Il est déraisonnable de s'attendre à une réaction immédiate, aussi urgent le problème soit-il, et les suggestions effectuées le sont sans aucune garantie.

1.3.7 BONS RÉFLEXES 6

- Dérouler les procédures comme prévu
- En cas de situation non prévue, s'arrêter pour faire le point
  - ne pas hésiter à remettre en cause l'analyse
  - ou la procédure elle-même

Dans l'idéal, des procédures détaillant les actions à effectuer ont été écrites pour le cas de figure rencontré. Dans ce cas, une fois que l'on s'est assuré d'avoir identifié la procédure appropriée, il faut la dérouler méthodiquement, point par point, et valider à chaque étape que tout se déroule comme prévu.

Si une étape de la procédure ne se passe pas comme prévu, il ne faut pas tenter de poursuivre tout de même son exécution sans avoir compris ce qui s'est passé et les conséquences. Cela pourrait être dangereux.

Il faut au contraire prendre le temps de comprendre le problème en procédant comme décrit précédemment, quitte à remettre en cause toute l'analyse menée auparavant, et la procédure ou les scripts utilisés.

C'est également pour parer à ce type de cas de figure qu'il est important de travailler sur une copie et non sur l'environnement de production directement.

1.3.8 BONS RÉFLEXES 7

- En cas de bug avéré
  - tenter de le cerner et de le reproduire au mieux
  - le signaler à la communauté de préférence (configuration, comment reproduire)

Ce n'est heureusement pas fréquent, mais il est possible que l'origine du problème soit liée à un bug de PostgreSQL lui-même.

Dans ce cas, la méthodologie appropriée consiste à essayer de reproduire le problème le plus fidèlement possible et de façon systématique, pour le cerner au mieux.

Il est ensuite très important de le signaler au plus vite à la communauté, généralement sur la liste pgsql-bugs@postgresql.org (cela nécessite une inscription préalable), en respectant les règles définies dans la documentation[4] .

Notamment (liste non exhaustive) :

- indiquer la version précise de PostgreSQL installée, et la méthode d'installation utilisée ;
- préciser la plate-forme utilisée, notamment la version du système d'exploitation utilisé et la configuration des ressources du serveur ;
- signaler uniquement les faits observés, éviter les spéculations sur l'origine du problème ;
- joindre le détail des messages d'erreurs observés (augmenter la verbosité des erreurs avec le paramètre `log_error_verbosity`) ;
- joindre un cas complet permettant de reproduire le problème de façon aussi simple que possible.

Pour les problèmes relevant du domaine de la sécurité (découverte d'une faille), la liste adéquate est security@postgresql.org.

---

[4] https://www.postgresql.org/docs/current/static/bug-reporting.html

1.3.9 BONS RÉFLEXES 8

- Après correction
- Tester complètement l'intégrité des données
  - pour détecter tous les problèmes
  - pour valider après restauration / correction
  - `pg_dumpall > /dev/null` ou `pg_basebackup`
- Reconstruction dans une autre instance
  - `pg_dumpall | psql -h autre serveur`

Une fois les actions correctives réalisées (restauration, recréation d'objets, mise à jour des données...), il faut tester intensivement pour s'assurer que le problème est bien complètement résolu.

Il est donc extrêmement important d'avoir préparé des cas de tests permettant de reproduire le problème de façon certaine, afin de valider la solution appliquée.

En cas de suspicion de corruption de données, il est également important de tenter de procéder à la lecture de la totalité des données depuis PostgreSQL.

Un premier outil pour cela est une sauvegarde avec `pg_basebackup` (voir plus loin).

Alternativement, la commande suivante, exécutée avec l'utilisateur système propriétaire de l'instance (généralement `postgres`) effectue une lecture complète de toutes les tables (mais sans les index ni les vues matérialisées), sans nécessiter de place sur disque supplémentaire :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell, la commande est :

```
PS C:\ pg_dumpall > $null
```

Cette commande ne devrait renvoyer aucune erreur. En cas de problème, notamment une somme de contrôle qui échoue, une erreur apparaîtra :

```
pg_dump: WARNING:  page verification failed, calculated checksum 20565 but expected 17796
pg_dump: erreur : Sauvegarde du contenu de la table « corrompue » échouée :
                  échec de PQgetResult().
pg_dump: erreur : Message d'erreur du serveur :
         ERROR:  invalid page in block 0 of relation base/104818/104828
pg_dump: erreur : La commande était : COPY public.corrompue (i) TO stdout;
pg_dumpall: erreur : échec de pg_dump sur la base de données « corruption », quitte
```

Même si la lecture des données par `pg_dumpall` ou `pg_dump` ne renvoie aucune erreur, il est toujours possible que des problèmes subsistent, par exemple des corruptions silencieuses,

des index incohérents avec les données…

Dans les situations les plus extrêmes (problème de stockage, fichiers corrompus), il est important de tester la validité des données dans une nouvelle instance en effectuant un export/import complet des données.

Par exemple, initialiser une nouvelle instance avec `initdb`, sur un autre système de stockage, voire sur un autre serveur, puis lancer la commande suivante (l'application doit être coupée, ce qui est normalement le cas depuis la détection de l'incident si les conseils précédents ont été suivis) pour exporter et importer à la volée :

```
$ pg_dumpall -h <serveur_corrompu> -U postgres | psql -h <nouveau_serveur> \
                                                    -U postgres postgres
$ vacuumdb --analyze -h <nouveau_serveur> -U postgres postgres
```

D'éventuels problèmes peuvent être détectés lors de l'import des données, par exemple si des corruptions entraînent l'échec de la reconstruction de clés étrangères. Il faut alors procéder au cas par cas.

Enfin, même si cette étape s'est déroulée sans erreur, tout risque n'est pas écarté, il reste la possibilité de corruption de données silencieuses. Sauf si la fonctionnalité de checksum de PostgreSQL a été activée sur l'instance (ce n'est pas activé par défaut !), le seul moyen de détecter ce type de problème est de valider les données fonctionnellement.

Dans tous les cas, en cas de suspicion de corruption de données en profondeur, il est fortement préférable d'accepter une perte de données et de restaurer une sauvegarde d'avant le début de l'incident, plutôt que de continuer à travailler avec des données dont l'intégrité n'est pas assurée.

1.3.10 MAUVAIS RÉFLEXES 1

- Paniquer
- Prendre une décision hâtive
  - exemple, supprimer des fichiers du répertoire `pg_wal`
- Lancer une commande sans la comprendre
  - exemple, `pg_resetwal` ou celles de l'extension `pg_surgery`

Quelle que soit la criticité du problème rencontré, la panique peut en faire quelque chose de pire.

Il faut impérativement garder son calme, et résister au mieux au stress et aux pressions qu'une situation de désastre ne manque pas de provoquer.

Il est également préférable d'éviter de sauter immédiatement à la conclusion la plus évidente. Il ne faut pas hésiter à retirer les mains du clavier pour prendre de la distance par rapport aux conséquences du problème, réfléchir aux causes possibles, prendre le temps d'aller chercher de l'information pour réévaluer l'ampleur réelle du problème.

La plus mauvaise décision que l'on peut être amenée à prendre lors de la gestion d'un incident est celle que l'on prend dans la précipitation, sans avoir bien réfléchi et mesuré son impact. Cela peut provoquer des dégâts irrécupérables, et transformer une situation d'incident en situation de crise majeure.

Un exemple classique de ce type de comportement est le cas où PostgreSQL est arrêté suite au remplissage du système de fichiers contenant les fichiers WAL, `pg_wal`.

Le réflexe immédiat d'un administrateur non averti pourrait être de supprimer les plus vieux fichiers dans ce répertoire, ce qui répond bien aux symptômes observés mais reste une erreur dramatique qui va rendre le démarrage de l'instance impossible.

Quoi qu'il arrive, ne jamais exécuter une commande sans être certain qu'elle correspond bien à la situation rencontrée, et sans en maîtriser complètement les impacts. Même si cette commande provient d'un document mentionnant les mêmes messages d'erreur que ceux rencontrés (et tout particulièrement si le document a été trouvé via une recherche hâtive sur Internet) !

Là encore, nous disposons comme exemple d'une erreur malheureusement fréquente, l'exécution de la commande `pg_resetwal` sur une instance rencontrant un problème. Comme l'indique la documentation, « *[cette commande] ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption* » et « *il ne faut pas perdre de vue que la base de données peut contenir des données incohérentes du fait de transactions partiellement validées* » (documentation[5]). Nous reviendrons ultérieurement sur les (rares) cas d'usage réels de cette commande, mais dans l'immense majorité des cas, l'utiliser va aggraver le problème, en ajoutant des problématiques de corruption logique des données !

Il convient donc de bien s'assurer de comprendre les conséquences de l'exécution de chaque action effectuée.

---

[5] https://docs.postgresql.fr/current/app-pgresetwal.html

1.3.11 MAUVAIS RÉFLEXES 2

- Arrêter le diagnostic quand les symptômes disparaissent
- Ne pas pousser l'analyse jusqu'au bout

Il est important de pousser la réflexion jusqu'à avoir complètement compris l'origine du problème et ses conséquences.

En premier lieu, même si les symptômes semblent avoir disparus, il est tout à fait possible que le problème soit toujours sous-jacent, ou qu'il ait eu des conséquences moins visibles mais tout aussi graves (par exemple, une corruption logique de données).

Ensuite, même si le problème est effectivement corrigé, prendre le temps de comprendre et de documenter l'origine du problème (rapport « post-mortem ») a une valeur inestimable pour prendre les mesures afin d'éviter que le problème ne se reproduise, et retrouver rapidement les informations utiles s'il venait à se reproduire malgré tout.

1.3.12 MAUVAIS RÉFLEXES 3

- Ne pas documenter
  - le résultat de l'investigation
  - les actions effectuées

Après s'être assuré d'avoir bien compris le problème rencontré, il est tout aussi important de le documenter soigneusement, avec les actions de diagnostic et de correction effectuées.

Ne pas le faire, c'est perdre une excellente occasion de gagner un temps précieux si le problème venait à se reproduire.

C'est également un risque supplémentaire dans le cas où les actions correctives menées n'auraient pas suffi à complètement corriger le problème ou auraient eu un effet de bord inattendu.

Dans ce cas, avoir pris le temps de noter le détail des actions effectuées fera là encore gagner un temps précieux.

1.4 RECHERCHER L'ORIGINE DU PROBLÈME

- Quelques pistes de recherche pour cerner le problème
- Liste non exhaustive

Les problèmes pouvant survenir sont trop nombreux pour pouvoir tous les lister, chaque élément matériel ou logiciel d'une architecture pouvant subir de nombreux types de défaillances.

Cette section liste quelques pistes classiques d'investigation à ne pas négliger pour s'efforcer de cerner au mieux l'étendue du problème, et en déterminer les conséquences.

1.4.1 PRÉREQUIS

- Avant de commencer à creuser
  - référencer les symptômes
  - identifier au mieux l'instant de démarrage du problème

La première étape est de déterminer aussi précisément que possible les symptômes observés, sans en négliger, et à partir de quel moment ils sont apparus.

Cela donne des informations précieuses sur l'étendue du problème, et permet d'éviter de se focaliser sur un symptôme particulier, parce que plus visible (par exemple l'arrêt brutal de l'instance), alors que la cause réelle est plus ancienne (par exemple des erreurs IO dans les traces système, ou une montée progressive de la charge sur le serveur).

1.4.2 RECHERCHE D'HISTORIQUE

- Ces symptômes ont-ils déjà été rencontrés dans le passé ?
- Ces symptômes ont-ils déjà été rencontrés par d'autres ?
- Attention à ne pas prendre les informations trouvées pour argent comptant !

Une fois les principaux symptômes identifiés, il est utile de prendre un moment pour déterminer si ce problème est déjà connu.

Notamment, identifier dans la base de connaissances si ces symptômes ont déjà été rencontrés dans le passé (d'où l'importance de bien documenter les problèmes).

Au-delà de la documentation interne, il est également possible de rechercher si ces symptômes ont déjà été rencontrés par d'autres.

Pour ce type de recherche, il est préférable de privilégier les sources fiables (documentation officielle, listes de discussion, plate-forme de support...) plutôt qu'un quelconque document d'un auteur non identifié.

Dans tous les cas, il faut faire très attention à ne pas prendre les informations trouvées pour argent comptant, et ce même si elles proviennent de la documentation interne ou d'une source fiable !

Il est toujours possible que les symptômes soient similaires mais que la cause soit différente. Il s'agit donc ici de mettre en place une base de travail, qui doit être complétée par une observation directe et une analyse.

1.4.3 MATÉRIEL

- Vérifier le système disque (SAN, carte RAID, disques)
- Un `fsync` est-il bien honoré de l'OS au disque ? (batteries !)
- Rechercher toute erreur matérielle
- Firmwares pas à jour
  - ou récemment mis à jour
- Matériel récemment changé

Les défaillances du matériel, et notamment du système de stockage, sont de celles qui peuvent avoir les impacts les plus importants et les plus étendus sur une instance et sur les données qu'elle contient.

Ce type de problème peut également être difficile à diagnostiquer en se contentant d'observer les symptômes les plus visibles. Il est facile de sous-estimer l'ampleur des dégâts.

Parmi les bonnes pratiques, il convient de vérifier la configuration et l'état du système disque (SAN, carte RAID, disques).

Quelques éléments étant une source habituelle de problèmes :

- le système disque n'honore pas les ordres `fsync` ? (SAN ? virtualisation ?) ;
- quel est l'état de la batterie du cache en écriture ?

Il faut évidemment rechercher la présence de toute erreur matérielle, au niveau des disques, de la mémoire, des CPU...

Vérifier également la version des firmwares installés. Il est possible qu'une nouvelle version corrige le problème rencontré, ou à l'inverse que le déploiement d'une nouvelle version soit à l'origine du problème.

Dans le même esprit, il faut vérifier si du matériel a récemment été changé. Il arrive que de nouveaux éléments soient défaillants.

Il convient de noter que l'investigation à ce niveau peut être grandement complexifiée par l'utilisation de certaines technologies (virtualisation, baies de stockage), du fait de la mutualisation des ressources, et de la séparation des compétences et des informations de supervision entre différentes équipes.

1.4.4 VIRTUALISATION

- Mutualisation excessive
- Configuration du stockage virtualisé
- Rechercher les erreurs aussi niveau superviseur
- Mises à jour non appliquées
  - ou appliquées récemment
- Modifications de configuration récentes

Tout comme pour les problèmes au niveau du matériel, les problèmes au niveau du système de virtualisation peuvent être complexes à détecter et à diagnostiquer correctement.

Le principal facteur de problème avec la virtualisation est lié à une mutualisation excessive des ressources.

Il est ainsi possible d'avoir un total de ressources allouées aux VM supérieur à celles disponibles sur l'hyperviseur, ce qui amène à des comportements de fort ralentissement, voire de blocage des systèmes virtualisés.

Si ce type d'architecture est couplé à un système de gestion de bascule automatique (Pacemaker, repmgr…), il est possible d'avoir des situations de bascules impromptues, voire des situations de *split brain*, qui peuvent provoquer des pertes de données importantes. Il est donc important de prêter une attention particulière à l'utilisation des ressources de l'hyperviseur, et d'éviter à tout prix la sur-allocation.

Par ailleurs, lorsque l'architecture inclut une brique de virtualisation, il est important de prendre en compte que certains problèmes ne peuvent être observés qu'à partir de l'hyperviseur, et pas à partir du système virtualisé. Par exemple, les erreurs matérielles ou système risquent d'être invisibles depuis une VM, il convient donc d'être vigilant, et de rechercher toute erreur sur l'hôte.

Il faut également vérifier si des modifications ont été effectuées peu avant l'incident, comme des modifications de configuration ou l'application de mises à jour.

Comme indiqué dans la partie traitant du matériel, l'investigation peut être grandement freinée par la séparation des compétences et des informations de supervision entre différentes équipes. Une bonne communication est alors la clé de la résolution rapide du problème.

1.4.5 SYSTÈME D'EXPLOITATION 1

- Erreurs dans les traces
- Mises à jour système non appliquées
- Modifications de configuration récentes

Après avoir vérifié les couches matérielles et la virtualisation, il faut ensuite s'assurer de l'intégrité du système d'exploitation.

La première des vérifications à effectuer est de consulter les traces du système pour en extraire les éventuels messages d'erreur :

- sous Linux, on trouvera ce type d'informations en sortie de la commande `dmesg`, et dans les fichiers traces du système, généralement situés sous `/var/log` ;
- sous Windows, on consultera à cet effet le journal des événements (les `event logs`).

Tout comme pour les autres briques, il faut également voir s'il existe des mises à jour des paquets qui n'auraient pas été appliquées, ou à l'inverse si des mises à jour, installations ou modifications de configuration ont été effectuées récemment.

1.4.6 SYSTÈME D'EXPLOITATION 2

- Opération d'IO impossible
  - FS plein ?
  - FS monté en lecture seule ?
- Tester l'écriture sur PGDATA
- Tester la lecture sur PGDATA

Parmi les problèmes fréquemment rencontrés se trouve l'impossibilité pour PostgreSQL d'accéder en lecture ou en écriture à un ou plusieurs fichiers.

La première chose à vérifier est de déterminer si le système de fichiers sous-jacent ne serait pas rempli à 100% (commande `df` sous Linux) ou monté en lecture seule (commande `mount` sous Linux).

On peut aussi tester les opérations d'écriture et de lecture sur le système de fichiers pour déterminer si le comportement y est global :

- pour tester une écriture dans le répertoire `PGDATA`, sous Linux :

```
$ touch $PGDATA/test_write
```

- pour tester une lecture dans le répertoire `PGDATA`, sous Linux :

```
$ cat $PGDATA/PGVERSION
```

Pour identifier précisément les fichiers présentant des problèmes, il est possible de tester la lecture complète des fichiers dans le point de montage :

```
$ tar cvf /dev/null $PGDATA
```

1.4.7 SYSTÈME D'EXPLOITATION 3

- Consommation excessive des ressources
  - OOM killer (overcommit !)
- Après un crash, vérifier les processus actifs
  - ne pas tenter de redémarrer si des processus persistent
- Outils : `sar`, `atop`...

Sous Linux, l'installation d'outils d'aide au diagnostic sur les serveurs est très important pour mener une analyse efficace, particulièrement le paquet `systat` qui permet d'utiliser la commande `sar`.

La lecture des traces système et des traces PostgreSQL permettent également d'avancer dans le diagnostic.

Un problème de consommation excessive des ressources peut généralement être anticipée grâce à une supervision sur l'utilisation des ressources et des seuils d'alerte appropriés. Il arrive néanmoins parfois que la consommation soit très rapide et qu'il ne soit pas possible de réagir suffisamment rapidement.

Dans le cas d'une consommation mémoire d'un serveur Linux qui menacerait de dépasser la quantité totale de mémoire allouable, le comportement par défaut de Linux est d'autoriser par défaut la tentative d'allocation.

Si l'allocation dépasse effectivement la mémoire disponible, alors le système va déclencher un processus *Out Of Memory Killer* (OOM Killer) qui va se charger de tuer les processus les plus consommateurs.

Dans le cas d'un serveur dédié à une instance PostgreSQL, il y a de grandes chances que le processus en question appartienne à l'instance.

S'il s'agit d'un *OOM Killer* effectuant un arrêt brutal (`kill -9`) sur un backend, l'instance PostgreSQL va arrêter immédiatement tous les processus afin de prévenir une corruption de la mémoire et les redémarrer.

S'il s'agit du processus principal de l'instance (*postmaster*), les conséquences peuvent être bien plus dramatiques, surtout si une tentative est faite de redémarrer l'instance sans vérifier si des processus actifs existent encore.

Pour un serveur dédié à PostgreSQL, la recommendation est habituellement de désactiver la sur-allocation de la mémoire, empêchant ainsi le déclenchement de ce phénomène.

Voir pour cela les paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio` (référence : https://kb.dalibo.com/overcommit_memory).

1.4.8 POSTGRESQL

- Relever les erreurs dans les traces
  - ou messages inhabituels
- Vérifier les mises à jour mineures

Tout comme pour l'analyse autour du système d'exploitation, la première chose à faire est rechercher toute erreur ou message inhabituel dans les traces de l'instance. Ces messages sont habituellement assez informatifs, et permettent de cerner la nature du problème. Par exemple, si PostgreSQL ne parvient pas à écrire dans un fichier, il indiquera précisément de quel fichier il s'agit.

Si l'instance est arrêtée suite à un crash, et que les tentatives de redémarrage échouent avant qu'un message puisse être écrit dans les traces, il est possible de tenter de démarrer l'instance en exécutant directement le binaire `postgres` afin que les premiers messages soient envoyés vers la sortie standard.

Il convient également de vérifier si des mises à jour qui n'auraient pas été appliquées ne corrigeraient pas un problème similaire à celui rencontré.

Identifier les mises à jours appliquées récemment et les modifications de configuration peut également aider à comprendre la nature du problème.

1.4.9 PARAMÉTRAGE DE POSTGRESQL 1

- La désactivation de certains paramètres est dangereuse
  - `fsync`
  - `full_page_write`

Si des corruptions de données sont relevées suite à un crash de l'instance, il convient particulièrement de vérifier la valeur du paramètre `fsync`.

En effet, si celui-ci est désactivé, les écritures dans les journaux de transactions ne sont pas effectuées de façon synchrone, ce qui implique que l'ordre des écritures ne sera pas conservé en cas de crash. Le processus de *recovery* de PostgreSQL risque alors de provoquer des corruptions si l'instance est malgré tout redémarrée.

Ce paramètre ne devrait jamais être positionné à une autre valeur que `on`, sauf dans des cas extrêmement particuliers (en bref, si l'on peut se permettre de restaurer intégralement les données en cas de crash, par exemple dans un chargement de données initial).

Le paramètre `full_page_write` indique à PostgreSQL d'effectuer une écriture complète d'une page chaque fois qu'elle reçoit une nouvelle écriture après un checkpoint, pour éviter un éventuel mélange entre des anciennes et nouvelles données en cas d'écriture partielle.

La désactivation de ce paramètre peut avoir le même type de conséquences que la désactivation de `fsync`.

1.4.10 PARAMÉTRAGE DE POSTGRESQL 2

- Activez les checksums !
  - `initdb --data-checksums`
  - ou `pg_checksums --enable` (v12)
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Vérification lors de `pg_basebackup` (v11)

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

Les sommes de contrôles (*checksums*) permettent de se prémunir contre des corruptions silencieuses de données. Leur mise en place est fortement recommandée sur une nouvelle instance. Malheureusement, jusqu'en version 11 comprise, on ne peut le faire qu'à

l'initialisation de l'instance. La version 12 permet de les mettre en place, *base arrêtée*, avec l'utilitaire `pg_checksums`[6] .

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb -D /tmp/sans_checksums/
$ initdb -D /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacun des deux clusters :

```
CREATE TABLE test (name text);
INSERT INTO test (name) VALUES ('toto');
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

```
SELECT pg_relation_filepath('test');

 pg_relation_filepath
----------------------
 base/12036/16317
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/12036/16317
$ hexedit /tmp/avec_checksums/base/12036/16399
```

Enfin, on peut ensuite exécuter des requêtes sur ces deux clusters.

Sans *checksums* :

```
TABLE test;

 name
------
 qoto
```

Avec *checksums* :

```
TABLE test;

WARNING:  page verification failed, calculated checksum 16321
          but expected 21348
ERROR:  invalid page in block 0 of relation base/12036/16387
```

Depuis la version 11, les sommes de contrôles, si elles sont là, sont vérifiées par défaut lors d'un `pg_basebackup`. En cas de corruption des données, l'opération sera interrompue.

[6] https://docs.postgresql.fr/current/app-pgchecksums.html

Il est possible de désactiver cette vérification avec l'option `--no-verify-checksums` pour obtenir une copie, aussi corrompue que l'original, mais pouvant servir de base de travail.

En pratique, si vous utilisez PostgreSQL 9.5 au moins et si votre processeur supporte les instructions SSE 4.2 (voir dans `/proc/cpuinfo`), il n'y aura pas d'impact notable en performances. Par contre vous générerez un peu plus de journaux.

1.4.11 ERREUR DE MANIPULATION

- Traces système, traces PostgreSQL
- Revue des dernières manipulations effectuées
- Historique des commandes
- Danger : `kill -9`, `rm -rf`, `rsync`, `find … -exec` ...

L'erreur humaine fait également partie des principales causes de désastre.

Une commande de suppression tapée trop rapidement, un oubli de clause `WHERE` dans une requête de mise à jour, nombreuses sont les opérations qui peuvent provoquer des pertes de données ou un crash de l'instance.

Il convient donc de revoir les dernières opérations effectuées sur le serveur, en commençant par les interventions planifiées, et si possible récupérer l'historique des commandes passées.

Des exemples de commandes particulièrement dangereuses :

- `kill -9`
- `rm -rf`
- `rsync`
- `find` (souvent couplé avec des commandes destructices comme `rm`, `mv`, `gzip`...)

1.5 OUTILS

- Quelques outils peuvent aider
  - à diagnostiquer la nature du problème
  - à valider la correction apportée
  - à appliquer un contournement
- ATTENTION
  - certains de ces outils peuvent corrompre les données !

1.5.1 OUTILS - PG_CONTROLDATA

- Fournit des informations de contrôle sur l'instance
- Ne nécessite pas que l'instance soit démarrée

L'outil `pg_controldata` lit les informations du fichier de contrôle d'une instance PostgreSQL.

Cet outil ne se connecte pas à l'instance, il a juste besoin d'avoir un accès en lecture sur le répertoire `PGDATA` de l'instance.

Les informations qu'il récupère ne sont donc pas du temps réel, il s'agit d'une vision de l'instance telle qu'elle était la dernière fois que le fichier de contrôle a été mis à jour. L'avantage est qu'elle peut être utilisée même si l'instance est arrêtée.

pg_controldata affiche notamment les informations initialisées lors d'initdb, telles que la version du catalogue, ou la taille des blocs, qui peuvent être cruciales si l'on veut restaurer une instance sur un nouveau serveur à partir d'une copie des fichiers.

Il affiche également de nombreuses informations utiles sur le traitement des journaux de transactions et des checkpoints, par exemple :

- positions de l'avant-dernier checkpoint et du dernier checkpoint dans les WAL ;
- nom du WAL correspondant au dernier WAL ;
- timeline sur laquelle se situe le dernier checkpoint ;
- instant précis du dernier checkpoint.

Quelques informations de paramétrage sont également renvoyées, comme la configuration du niveau de WAL, ou le nombre maximal de connexions autorisées.

En complément, le dernier état connu de l'instance est également affiché. Les états potentiels sont :

- `in production` : l'instance est démarrée et est ouverte en écriture ;
- `shut down` : l'instance est arrêtée ;
- `in archive recovery` : l'instance est démarrée et est en mode `recovery` (restauration, `Warm` ou `Hot Standby`) ;
- `shut down in recovery` : l'instance s'est arrêtée alors qu'elle était en mode `recovery` ;
- `shutting down` : état transitoire, l'instance est en cours d'arrêt ;
- `in crash recovery` : état transitoire, l'instance est en cours de démarrage suite à un crash ;
- `starting up` : état transitoire, concrètement jamais utilisé.

Bien entendu, comme ces informations ne sont pas mises à jour en temps réel, elles peuvent être erronées.

Cet asynchronisme est intéressant pour diagnostiquer un problème, par exemple si `pg_controldata` renvoie l'état `in production` mais que l'instance est arrêtée, cela signifie que l'arrêt n'a pas été effectué proprement (*crash* de l'instance, qui sera donc suivi d'un `recovery` au démarrage).

Exemple de sortie de la commande :

```
$ /usr/pgsql-10/bin/pg_controldata /var/lib/pgsql/10/data

pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6451139765284827825
Database cluster state:               in production
pg_control last modified:             Mon 28 Aug 2017 03:40:30 PM CEST
Latest checkpoint location:           1/2B04EC0
Prior checkpoint location:            1/2B04DE8
Latest checkpoint's REDO location:    1/2B04E88
Latest checkpoint's REDO WAL file:    000000010000000100000002
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:1023
Latest checkpoint's NextOID:          41064
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        548
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1022
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Mon 28 Aug 2017 03:40:30 PM CEST
Fake LSN counter for unlogged rels:   0/1
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
```

```
max_prepared_xacts setting:           0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:            7fb23aca2465c69b2c0f54ccf03e0ece
                                      3c0933c5f0e5f2c096516099c9688173
```

1.5.2 OUTILS - EXPORT/IMPORT DE DONNÉES

- `pg_dump`
- `pg_dumpall`
- `COPY`
- `psql` / `pg_restore`
  - `--section=pre-data` / `data` / `post-data`

Les outils `pg_dump` et `pg_dumpall` permettent d'exporter des données à partir d'une instance démarrée.

Dans le cadre d'un incident grave, il est possible de les utiliser pour :

- extraire le contenu de l'instance ;
- extraire le contenu des bases de données ;
- tester si les données sont lisibles dans un format compréhensible par PostgreSQL.

Par exemple, un moyen rapide de s'assurer que tous les fichiers des tables de l'instance sont lisibles est de forcer leur lecture complète, notamment grâce à la commande suivante :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell :

```
pg_dumpall > $null
```

Attention, les fichiers associés aux index ne sont pas parcourus pendant cette opéra-

tion. Par ailleurs, ne pas avoir d'erreur ne garantit en aucun cas pas l'intégrité fonctionnelle des données : les corruptions peuvent très bien être silencieuses ou concerner les index. Une vérification exhaustive implique d'autres outils comme `pg_checksums` ou `pg_basebackup` (voir plus loin).

Si `pg_dumpall` ou `pg_dump` renvoient des messages d'erreur et ne parviennent pas à exporter certaines tables, il est possible de contourner le problème à l'aide de la commande `COPY`, en sélectionnant exclusivement les données lisibles autour du bloc corrompu.

Il convient ensuite d'utiliser `psql` ou `pg_restore` pour importer les données dans une nouvelle instance, probablement sur un nouveau serveur, dans un environnement non affecté par le problème. Pour parer au cas où le réimport échoue à cause de contraintes non respectées, il est souvent préférable de faire le réimport par étapes :

```
$ pg_restore -1 --section=pre-data --verbose -d cible base.dump
$ pg_restore -1 --section=data --verbose -d cible base.dump
$ pg_restore -1 --section=post-data --exit-on-error --verbose -d cible base.dump
```

En cas de problème, on verra les contraintes posant problème.

Il peut être utile de générer les scripts en pur SQL avant de les appliquer, éventuellement par étape :

```
$ pg_restore --section=post-data -f postdata.sql base.dump
```

Pour rappel, même après un export / import de données réalisé avec succès, des corruptions logiques peuvent encore être présentes. Il faut donc être particulièrement vigilant et prendre le temps de valider l'intégrité fonctionnelle des données.

1.5.3 OUTILS - PAGEINSPECT

- Extension
- Vision du contenu d'un bloc
- Sans le dictionnaire, donc sans décodage des données
- Affichage brut
- Utilisé surtout en debug, ou dans les cas de corruption
- Fonctions de décodage pour les tables, les index (B-tree, hash, GIN, GiST), FSM
- Nécessite de connaître le code de PostgreSQL

Voici quelques exemples.

Contenu d'une page d'une table :

```
SELECT * FROM heap_page_items(get_raw_page('dspam_token_data',0)) LIMIT 2;
```

```
-[ RECORD 1 ]------
lp          | 1
lp_off      | 8152
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,7)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0100000001000000010000000 1000000
-[ RECORD 2 ]------
lp          | 2
lp_off      | 8112
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,8)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0200000001000000010000000 2000000
```

Et son entête :

```
SELECT * FROM page_header(get_raw_page('dspam_token_data',0));

-[ RECORD 1 ]--------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 0
lower     | 56
upper     | 7872
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 839
```

Méta-données d'un index (contenu dans la première page) :

```
SELECT * FROM bt_metap('dspam_token_data_uid_key');

-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 243
level     | 2
fastroot  | 243
fastlevel | 2
```

La page racine est la 243. Allons la voir :

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',243) LIMIT 10;

 offset |   ctid    | len |nulls|vars|                data
--------+-----------+-----+-----+----+-----------------------------------
      1 | (3,1)     |   8 | f   | f  |
      2 | (44565,1) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (242,1)   |  20 | f   | f  | 77 c6 0d 6f a6 92 db 81 28 00 00 00
      4 | (43569,1) |  20 | f   | f  | 47 a6 aa be 29 e3 13 83 18 00 00 00
      5 | (481,1)   |  20 | f   | f  | 30 17 dd 8e d9 72 7d 84 0a 00 00 00
      6 | (43077,1) |  20 | f   | f  | 5c 3c 7b c5 5b 7a 4e 85 0a 00 00 00
      7 | (719,1)   |  20 | f   | f  | 0d 91 d5 78 a9 72 88 86 26 00 00 00
      8 | (41209,1) |  20 | f   | f  | a7 8a da 17 95 17 cd 87 0a 00 00 00
      9 | (957,1)   |  20 | f   | f  | 78 e9 64 e9 64 a9 52 89 26 00 00 00
     10 | (40849,1) |  20 | f   | f  | 53 11 e9 64 e9 1b c3 8a 26 00 00 00
```

La première entrée de la page 243, correspondant à la donnée `f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00` est stockée dans la page 3 de notre index :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',3);

-[ RECORD 1 ]-+-----
blkno         | 3
type          | i
live_items    | 202
dead_items    | 0
avg_item_size | 19
page_size     | 8192
free_size     | 3312
btpo_prev     | 0
btpo_next     | 44565
btpo          | 1
btpo_flags    | 0

SELECT * FROM bt_page_items('dspam_token_data_uid_key',3) LIMIT 10;

 offset |   ctid    | len |nulls|vars|                data
--------+-----------+-----+-----+----+-----------------------------------
```

```
     1 | (38065,1) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
     2 | (1,1)     |   8 | f   | f  |
     3 | (37361,1) |  20 | f   | f  | 30 fd 30 b8 70 c9 01 80 26 00 00 00
     4 | (2,1)     |  20 | f   | f  | 18 2c 37 36 27 03 03 80 27 00 00 00
     5 | (4,1)     |  20 | f   | f  | 36 61 f3 b6 c5 1b 03 80 0f 00 00 00
     6 | (43997,1) |  20 | f   | f  | 30 4a 32 58 c8 44 03 80 27 00 00 00
     7 | (5,1)     |  20 | f   | f  | 88 fe 97 6f 7e 5a 03 80 27 00 00 00
     8 | (51136,1) |  20 | f   | f  | 74 a8 5a 9b 15 5d 03 80 28 00 00 00
     9 | (6,1)     |  20 | f   | f  | 44 41 3c ee c8 fe 03 80 0a 00 00 00
    10 | (45317,1) |  20 | f   | f  | d4 b0 7c fd 5d 8d 05 80 26 00 00 00
```

Le type de la page est `i`, c'est-à-dire «internal», donc une page interne de l'arbre. Continuons notre descente, allons voir la page 38065 :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',38065);
```

```
-[ RECORD 1 ]-+-----
blkno         | 38065
type          | l
live_items    | 169
dead_items    | 21
avg_item_size | 20
page_size     | 8192
free_size     | 3588
btpo_prev     | 118
btpo_next     | 119
btpo          | 0
btpo_flags    | 65
```

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',38065) LIMIT 10;
```

```
 offset |    ctid     | len |nulls|vars|                data
--------+-------------+-----+-----+----+-----------------------------------
      1 | (11128,118) |  20 | f   | f  | 33 37 89 95 b9 23 cc 80 0a 00 00 00
      2 | (45713,181) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (45424,97)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 26 00 00 00
      4 | (45255,28)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 27 00 00 00
      5 | (15672,172) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 28 00 00 00
      6 | (5456,118)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 0f 00 00 00
      7 | (8356,206)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 28 00 00 00
      8 | (33895,272) |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0a 00 00 00
      9 | (5176,108)  |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0f 00 00 00
     10 | (5466,41)   |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 26 00 00 00
```

Nous avons trouvé une feuille (type `l`). Les ctid pointés sont maintenant les adresses dans la table :

```
SELECT * FROM dspam_token_data WHERE ctid = '(11128,118)';
```

```
 uid |        token         | spam_hits | innocent_hits |  last_hit
-----+----------------------+-----------+---------------+------------
  40 | -6317261189288392210 |         0 |             3 | 2014-11-10
```

1.5.4 OUTILS - PG_RESETWAL

- Efface les WAL courants
- Permet à l'instance de démarrer en cas de corruption d'un WAL
  - comme si elle était dans un état cohérent
  - ...ce qui n'est pas le cas
- **Cet outil est dangereux et mène à des corruptions !!!**
- Pour récupérer ce qu'on peut, et réimporter ailleurs

`pg_resetwal` est un outil fourni avec PostgreSQL. Son objectif est de pouvoir démarrer une instance après un crash si des corruptions de fichiers (typiquement WAL ou fichier de contrôle) empêchent ce démarrage.

Cette action n'est pas une action de réparation ! La réinitialisation des journaux de transactions implique que des transactions qui n'étaient que partiellement validées ne seront pas détectées comme telles, et ne seront donc pas annulées lors du *recovery*.

La conséquence est que les **données de l'instance ne sont plus cohérentes**. Il est fort possible d'y trouver des violations de contraintes diverses (notamment clés étrangères), ou d'autres cas d'incohérences plus difficiles à détecter.

Il s'utilise manuellement, en ligne de commande. Sa fonctionnalité principale est d'effacer les fichiers WAL courants, et il se charge également de réinitialiser les informations correspondantes du fichier de contrôle.

Il est possible de lui spécifier les valeurs à initialiser dans le fichier de contrôle si l'outil ne parvient pas à les déterminer (par exemple, si tous les WAL dans le répertoire `pg_wal` ont été supprimés).

Attention, `pg_resetwal` ne doit **jamais** être utilisé sur une instance démarrée. Avant d'exécuter l'outil, il faut toujours vérifier qu'il ne reste aucun processus de l'instance.

Après la réinitialisation des WAL, une fois que l'instance a démarré, **il ne faut surtout pas ouvrir les accès à l'application** ! Comme indiqué, les données présentent sans aucun doute des incohérences, et toute action en écriture à ce point ne ferait qu'aggraver le problème.

L'étape suivante est donc de faire un export immédiat des données, de les restaurer dans une nouvelle instance initialisée à cet effet (de préférence sur un nouveau serveur, surtout

si l'origine de la corruption n'a pas été clairement identifiée), et ensuite de procéder à une validation méthodique des données.

Il est probable que certaines données incohérentes puissent être identifiées à l'import, lors de la phase de recréation des contraintes : celles-ci échoueront si les données ne les respectent, ce qui permettra de les identifier.

En ce qui concerne les incohérences qui passeront au travers de ces tests, il faudra les trouver et les corriger manuellement, en procédant à une validation fonctionnelle des données.

Il faut donc bien retenir les points suivants :

- `pg_resetwal` n'est pas magique ;
- `pg_resetwal` rend les données incohérentes (ce qui est souvent pire qu'une simple perte d'une partie des données, comme on aurait en restaurant une sauvegarde) ;
- n'utiliser `pg_resetwal` que s'il n'y a aucun autre moyen de faire autrement pour récupérer les données ;
- ne pas l'utiliser sur l'instance ayant subi le problème, mais sur une copie complète effectuée à froid ;
- après usage, exporter toutes les données et les importer dans une nouvelle instance ;
- valider soigneusement les données de la nouvelle instance.

1.5.5 OUTILS - EXTENSION PG_SURGERY

- Extension apparue en v14
- Collection de fonctions permettant de modifier le statut des tuples d'une relation
- Extrèmement dangereuse

Cette extension regroupe des fonctions qui permettent de modifier le satut d'un tuple dans une relation. Il est par exemple possible de rendre une ligne morte ou de rendre visible des tuples qui sont invisibles à cause des informations de visibilité.

Ces fonctions sont dangereuses et peuvent provoquer ou aggraver des corruptions. Elles peuvent pas exemple rendre une table incohérente par rapport à ses index, ou provoquer une violation de contrainte d'unicité ou de clé étrangère. Il ne faut donc les utiliser qu'en dernier recours, sur une copie de votre instance.

1.5.6 OUTILS - VÉRIFICATION D'INTÉGRITÉ

- À froid : `pg_checksums` (à froid, v11)
- Lors d'une sauvegarde : `pg_basebackup` (v11)
- `amcheck` : pure vérification
  - v10 : 2 fonctions pour l'intégrité des index
  - v11 : vérification de la cohérence avec la table (probabiliste)
  - v14 : ajout d'un outil `pg_amcheck`

Depuis la version 11, `pg_checksums` permet de vérifier les sommes de contrôles existantes sur les bases de données **à froid** : l'instance doit être arrêtée proprement auparavant. (En version 11 l'outil s'appelait `pg_verify_checksums`.)

Par exemple, suite à une modification de deux blocs dans une table avec l'outil `hexedit`, on peut rencontrer ceci :

```
$ /usr/pgsql-12/bin/pg_checksums -D /var/lib/pgsql/12/data -c

pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 0:
  calculated checksum 5BF9 but block contains C55D
pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 4438:
  calculated checksum A3 but block contains B8AE
Checksum operation completed
Files scanned:  1282
Blocks scanned: 28484
Bad checksums:  2
Data checksum version: 1
```

À partir de PostgreSQL 12, l'outil `pg_checksums` peut aussi ajouter ou supprimer les sommes de contrôle sur une instance existante arrêtée (donc après le `initdb`), ce qui n'était pas possible dans les versions antérieures.

Une alternative, toujours à partir de la version 11, est d'effectuer une sauvegarde physique avec `pg_basebackup`, ce qui est plus lourd, mais n'oblige pas à arrêter la base.

Le module `amcheck` était apparu en version 10 pour vérifier la cohérence des index et de leur structure interne, et ainsi détecter des bugs, des corruptions dues au système de fichier voire à la mémoire. Il définit deux fonctions :

- `bt_index_check` est destinée aux vérifications de routine, et ne pose qu'un verrou *AccessShareLock* peu gênant ;
- `bt_index_parent_check` est plus minutieuse, mais son exécution gêne les modifications dans la table (verrou *ShareLock* sur la table et l'index) et elle ne peut pas être

exécutée sur un serveur secondaire.

En v11 apparaît le nouveau paramètre `heapallindex`. S'il vaut `true`, chaque fonction effectue une vérification supplémentaire en recréant temporairement une structure d'index et en la comparant avec l'index original. `bt_index_check` vérifiera que chaque entrée de la table possède une entrée dans l'index. `bt_index_parent_check` vérifiera en plus qu'à chaque entrée de l'index correspond une entrée dans la table.

Les verrous posés par les fonctions ne changent pas. Néanmoins, l'utilisation de ce mode a un impact sur la durée d'exécution des vérifications. Pour limiter l'impact, l'opération n'a lieu qu'en mémoire, et dans la limite du paramètre `maintenance_work_mem` (soit entre 256 Mo et 1 Go, parfois plus, sur les serveurs récents). C'est cette restriction mémoire qui implique que la détection de problèmes est probabiliste pour les plus grosses tables (selon la documentation, la probabilité de rater une incohérence est de 2 % si l'on peut consacrer 2 octets de mémoire à chaque ligne). Mais rien n'empêche de relancer les vérifications régulièrement, diminuant ainsi les chances de rater une erreur.

`amcheck` ne fournit aucun moyen de corriger une erreur, puisqu'il détecte des choses qui ne devraient jamais arriver. `REINDEX` sera souvent la solution la plus simple et facile, mais tout dépend de la cause du problème.

Soit `unetable_pkey`, un index de 10 Go sur un entier :

```
CREATE EXTENSION amcheck ;

SELECT bt_index_check('unetable_pkey');
Durée : 63753,257 ms (01:03,753)

SELECT bt_index_check('unetable_pkey', true);
Durée : 234200,678 ms (03:54,201)
```

Ici, la vérification exhaustive multiplie le temps de vérification par un facteur 4.

En version 14, PostgreSQL dispose d'un nouvel outil appelé `pg_amcheck`. Ce dernier facilite l'utilisation de l'extension `amcheck`.

1.6 CAS TYPE DE DÉSASTRES

- Les cas suivants sont assez rares
- Ils nécessitent généralement une restauration
- Certaines manipulations à haut risque sont possibles
  - mais complètement déconseillées !

Cette section décrit quelques-unes des pires situations de corruptions que l'on peut être amené à observer.

Dans la quasi-totalité des cas, la seule bonne réponse est la restauration de l'instance à partir d'une sauvegarde fiable.

1.6.1 AVERTISSEMENT

- Privilégier une solution fiable (restauration, bascule)
- Les actions listées ici sont parfois destructrices
- La plupart peuvent (et vont) provoquer des incohérences
- Travailler sur une copie

La plupart des manipulations mentionnées dans cette partie sont destructives, et peuvent (et vont) provoquer des incohérences dans les données.

Tous les experts s'accordent pour dire que l'utilisation de telles méthodes pour récupérer une instance tend à aggraver le problème existant ou à en provoquer de nouveaux, plus graves. S'il est possible de l'éviter, ne pas les tenter (*ie* : préférer la restauration d'une sauvegarde) !

S'il n'est pas possible de faire autrement (ie : pas de sauvegarde utilisable, données vitales à extraire...), alors TRAVAILLER SUR UNE COPIE.

Il ne faut pas non plus oublier que chaque situation est unique, il faut prendre le temps de bien cerner l'origine du problème, documenter chaque action prise, s'assurer qu'un retour arrière est toujours possible.

1.6.2 CORRUPTION DE BLOCS DANS DES INDEX

- Messages d'erreur lors des accès par l'index ; requêtes incohérentes
- Données différentes entre un indexscan et un seqscan
- Supprimer et recréer l'index : `REINDEX`

Les index sont des objets de structure complexe, ils sont donc particulièrement vulnérables aux corruptions.

Lorsqu'un index est corrompu, on aura généralement des messages d'erreur de ce type :

```
ERROR: invalid page header in block 5869177 of relation base/17291/17420
```

Il peut arriver qu'un bloc corrompu ne renvoie pas de message d'erreur à l'accès, mais que les données elles-mêmes soient altérées, ou que des filtres ne renvoient pas les données attendues.

Ce cas est néanmoins très rare dans un bloc d'index.

Dans la plupart des cas, si les données de la table sous-jacente ne sont pas affectées, il est possible de réparer l'index en le reconstruisant intégralement grâce à la commande `REINDEX`.

1.6.3 CORRUPTION DE BLOCS DANS DES TABLES 1

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

- Cas plus problématique
- Restauration probablement nécessaire

Les corruptions de blocs vont généralement déclencher des erreurs du type suivant :

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

Si la relation concernée est une table, tout ou partie des données contenues dans ces blocs est perdu.

L'apparition de ce type d'erreur est un signal fort qu'une restauration est certainement nécessaire.

1.6.4 CORRUPTION DE BLOCS DANS DES TABLES 2

- `SET zero_damaged_pages = true ;` + `VACUUM FULL`
- Des données vont certainement être perdues

Néanmoins, s'il est nécessaire de lire le maximum de données possibles de la table, il est possible d'utiliser l'option de PostgreSQL `zero_damaged_pages` pour demander au moteur de réinitialiser les blocs invalides à zéro lorsqu'ils sont lus au lieu de tomber en erreur. Il s'agit d'un des très rares paramètres absents de `postgresql.conf`.

Par exemple :

```
> SET zero_damaged_pages = true ;
SET
> VACUUM FULL mycorruptedtable ;
WARNING:  invalid page header in block 32570 of relation base/16390/2663;
          zeroing out page
VACUUM
```

Si cela se termine sans erreur, les blocs invalides ont été réinitialisés.

Les données qu'ils contenaient sont évidemment perdues, mais la table peut désormais être accédée dans son intégralité en lecture, permettant ainsi par exemple de réaliser un export des données pour récupérer ce qui peut l'être.

Attention, du fait des données perdues, le résultat peut être incohérent (contraintes non respectées…).

Par ailleurs, par défaut PostgreSQL ne détecte pas les corruptions logiques, c'est-à-dire n'affectant pas la structure des données mais uniquement le contenu.

Il ne faut donc pas penser que la procédure d'export complet de données suivie d'un import sans erreur garantit l'absence de corruption.

1.6.5 CORRUPTION DE BLOCS DANS DES TABLES 3

- Si la corruption est importante, l'accès au bloc peut faire crasher l'instance
- Il est tout de même possible de réinitialiser le bloc
  - identifier le fichier à l'aide de `pg_relation_filepath()`
  - trouver le bloc avec `ctid` / `pageinspect`
  - réinitialiser le bloc avec `dd`
  - il faut vraiment ne pas avoir d'autre choix

Dans certains cas, il arrive que la corruption soit suffisamment importante pour que le simple accès au bloc fasse crasher l'instance.

Dans ce cas, le seul moyen de réinitialiser le bloc est de le faire manuellement au niveau du fichier, instance arrêtée, par exemple avec la commande `dd`.

Pour identifier le fichier associé à la table corrompue, il est possible d'utiliser la fonction `pg_relation_filepath()` :

```
> SELECT pg_relation_filepath('test_corruptindex') ;

 pg_relation_filepath
----------------------
 base/16390/40995
```

Le résultat donne le chemin vers le fichier principal de la table, relatif au `PGDATA` de l'instance.

Attention, une table peut contenir plusieurs fichiers. Par défaut une instance PostgreSQL sépare les fichiers en segments de 1 Go. Une table dépassant cette taille aura donc des fichiers supplémentaires (`base/16390/40995.1`, `base/16390/40995.2`...).

Pour trouver le fichier contenant le bloc corrompu, il faudra donc prendre en compte le numéro du bloc trouvé dans le champ `ctid`, multiplier ce numéro par la taille du bloc (paramètre `block_size`, 8 ko par défaut), et diviser le tout par la taille du segment.

Cette manipulation est évidemment extrêmement risquée, la moindre erreur pouvant rendre irrécupérables de grandes portions de données.

Il est donc fortement déconseillé de se lancer dans ce genre de manipulations à moins d'être absolument certain que c'est indispensable.

Encore une fois, ne pas oublier de travailler sur une copie, et pas directement sur l'instance de production.

1.6.6 CORRUPTION DES WAL 1

- Situés dans le répertoire `pg_wal`
- Les WAL sont nécessaires au *recovery*
- Démarrage impossible s'ils sont corrompus ou manquants
- Si les fichiers WAL ont été archivés, les récupérer
- Sinon, la restauration est la seule solution viable

Les fichiers WAL sont les journaux de transactions de PostgreSQL.

Leur fonction est d'assurer que les transactions qui ont été effectuées depuis le dernier checkpoint ne seront pas perdues en cas de crash de l'instance.

Si certains sont corrompus ou manquants (rappel : il ne faut JAMAIS supprimer les fichiers WAL, même si le système de fichiers est plein !), alors PostgreSQL ne pourra pas redémarrer.

Si l'archivage était activé et que les fichiers WAL affectés ont bien été archivés, alors il est possible de les restaurer avant de tenter un nouveau démarrage.

Si ce n'est pas possible ou des fichiers WAL archivés ont également été corrompus ou supprimés, l'instance ne pourra pas redémarrer.

Dans cette situation, comme dans la plupart des autres évoquées ici, la seule solution permettant de s'assurer que les données ne seront pas corrompues est de procéder à une restauration de l'instance.

1.6.7 CORRUPTION DES WAL 2

- `pg_resetwal` permet de forcer le démarrage
- ATTENTION !!!
  - cela va provoquer des pertes de données
  - des corruptions de données sont également probables
  - ce n'est pas une action corrective !

L'utilitaire `pg_resetwal` a comme fonction principale de supprimer les fichiers WAL courants et d'en créer un nouveau, avant de mettre à jour le fichier de contrôle pour permettre le redémarrage.

Au minimum, cette action va provoquer la perte de toutes les transactions validées effectuées depuis le dernier checkpoint.

Il est également probable que des incohérences vont apparaître, certaines relativement

simples à détecter via un export/import (incohérences dans les clés étrangères par exemple), certaines complètement invisibles.

L'utilisation de cet utilitaire est extrêmement dangereuse, n'est pas recommandée, et ne peut jamais être considérée comme une action corrective. Il faut toujours privilégier la restauration d'une sauvegarde plutôt que son exécution.

Si l'utilisation de `pg_resetwal` est néanmoins nécessaire (par exemple pour récupérer des données absentes de la sauvegarde), alors il faut travailler sur une copie des fichiers de l'instance, récupérer ce qui peut l'être à l'aide d'un export de données, et les importer dans une autre instance.

Les données récupérées de cette manière devraient également être soigneusement validées avant d'être importée de façon à s'assurer qu'il n'y a pas de corruption silencieuse.

Il ne faut en aucun cas remettre une instance en production après une réinitialisation des WAL.

1.6.8 CORRUPTION DU FICHIER DE CONTRÔLE

- Fichier `global/pg_control`
- Contient les informations liées au dernier checkpoint
- Sans lui, l'instance ne peut pas démarrer
- Recréation avec `pg_resetwal`… parfois
- Restauration nécessaire

Le fichier de contrôle de l'instance contient de nombreuses informations liées à l'activité et au statut de l'instance, notamment l'instant du dernier checkpoint, la position correspondante dans les WAL, le numéro de transaction courant et le prochain à venir…

Ce fichier est le premier lu par l'instance. S'il est corrompu ou supprimé, l'instance ne pourra pas démarrer.

Il est possible de forcer la réinitialisation de ce fichier à l'aide de la commande `pg_resetwal`, qui va se baser par défaut sur les informations contenues dans les fichiers WAL présents pour tenter de « deviner » le contenu du fichier de contrôle.

Ces informations seront très certainement erronées, potentiellement à tel point que même l'accès aux bases de données par leur nom ne sera pas possible :

```
$ pg_isready
/var/run/postgresql:5432 - accepting connections
```

```
$ psql postgres
psql: FATAL:  database "postgres" does not exist
```

Encore une fois, utiliser `pg_resetwal` n'est en aucun cas une solution, mais doit uniquement être considéré comme un contournement temporaire à une situation désastreuse.

Une instance altérée par cet outil ne doit pas être considérée comme saine.

1.6.9 CORRUPTION DU CLOG

- Fichiers dans `pg_xact`
- Statut des différentes transactions
- Son altération risque de causer des incohérences

Le fichier CLOG (*Commit Log*) dans `PGDATA/pg_xact/` contient le statut des différentes transactions, notamment si celles-ci sont en cours, validées ou annulées.

S'il est altéré ou supprimé, il est possible que des transactions qui avaient été marquées comme annulées soient désormais considérées comme valides, et donc que les modifications de données correspondantes deviennent visibles aux autres transactions.

C'est évidemment un problème d'incohérence majeur, tout problème avec ce fichier devrait donc être soigneusement analysé.

Il est préférable dans le doute de procéder à une restauration et d'accepter une perte de données plutôt que de risquer de maintenir des données incohérentes dans la base.

1.6.10 CORRUPTION DU CATALOGUE SYSTÈME

- Le catalogue contient la définition du schéma
- Sans lui, les données sont inaccessibles
- Situation très délicate…

Le catalogue système contient la définition de toutes les relations, les méthodes d'accès, la correspondance entre un objet et un fichier sur disque, les types de données existantes…

S'il est incomplet, corrompu ou inaccessible, l'accès aux données en SQL risque de ne pas être possible du tout.

Cette situation est très délicate, et appelle là encore une restauration.

Si le catalogue était complètement inaccessible, sans sauvegarde la seule solution restante serait de tenter d'extraire les données directement des fichiers data de l'instance, en oubliant toute notion de cohérence, de type de données, de relation...

Personne ne veut faire ça.

1.7 CONCLUSION

- Les désastres peuvent arriver
- Il faut s'y être préparé
- Faites des sauvegardes !
  - et testez-les

1.8 QUIZ

https://dali.bo/i5_quiz

1.9 TRAVAUX PRATIQUES

1.9.1 CORRUPTION D'UN BLOC DE DONNÉES

Nous allons corrompre un bloc et voir certains impacts possibles.

> Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

> Créer une base **pgbench** avec un facteur d'échelle 10 **avec** les clés étrangères entre tables (option `--foreign-keys`).

> Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

> Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

> Arrêter PostgreSQL.

> Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

> Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

> Tenter un `pgdumpall > /dev/null`.

> Arrêter PostgreSQL.
> Voir ce que donne `pg_checksums` (en v12) ou `pg_verify_checksums` (en v11).

> Faire une copie de travail à froid du PGDATA.
> Protéger en écriture le PGDATA original.
> Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

> Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause). Démarrer le cluster sur la copie avec `pg_ctl`.

> Que renvoie `SELECT * FROM pgbench_accounts LIMIT 100` ?

> Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

1.9.2 CORRUPTION D'UN BLOC DE DONNÉES ET INCOHÉRENCES

Nous allons à présent corrompre une table portant une clé étrangère. Nous continuons sur la copie de la base de travail, où les sommes de contrôle ont été désactivées.

> Consulter le format et le contenu de la table `pgbench_branches`.

> Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

> Arrêter PostgreSQL.
> Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
> En toute fin de fichier, remplacer le dernier `01` par un `FF`.
> Redémarrer PostgreSQL.

> Compter le nombre de lignes dans `pgbench_branches`.
> Recompter après `SET enable_seqscan TO off ;`.
> Quelle est la bonne réponse ? Vérifier le contenu de la table.

> Qu'affiche `pageinspect` pour cette table ?

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

> Exporter `pgbench_accounts`, définition des index comprise. Supprimer la table (il faudra supprimer `pgbench_history` aussi). Tenter de la réimporter.

1.10 TRAVAUX PRATIQUES (SOLUTION)

1.10.1 CORRUPTION D'UN BLOC DE DONNÉES

> Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

```
# SHOW data_checksums ;

 data_checksums
----------------
 on
```

Si la réponse est `off`, on peut (à partir de la v12) mettre les checksums en place :

```
$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data.BACKUP/  --enable --progress
58/58 MB (100%) computed
Checksum operation completed
Files scanned:  964
Blocks scanned: 7524
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
```

> Créer une base **pgbench** avec un facteur d'échelle 10 **avec** les clés étrangères entre tables (option `--foreign-keys`).

```
$ dropdb --if-exists pgbench ;
$ createdb pgbench ;

$ /usr/pgsql-14/bin/pgbench -i -s 10 -d pgbench --foreign-keys
```

```
...
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 0.15 s, remaining 1.31 s)
200000 of 1000000 tuples (20%) done (elapsed 0.35 s, remaining 1.39 s)
..
1000000 of 1000000 tuples (100%) done (elapsed 2.16 s, remaining 0.00 s)
vacuuming...
creating primary keys...
creating foreign keys...
done.
```

> Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

La table fait 128 Mo selon un `\d+`.

La clé `aid` va de 1 à 100000 :

```
# SELECT min(aid), max(aid) FROM pgbench_accounts ;

 min |   max
-----+---------
   1 | 1000000
```

Un `SELECT` montre que les valeurs sont triées mais c'est dû à l'initialisation.

> Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

```
 SELECT pg_relation_filepath('pgbench_accounts') ;

 pg_relation_filepath
----------------------
 base/16454/16489
```

> Arrêter PostgreSQL.

```
 # systemctl restart postgresql-14
```

Cela permet d'être sûr qu'il ne va pas écraser nos modifications lors d'un checkpoint.

> Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

```
 # dnf install hexedit
```

```
postgres$ hexedit /var/lib/pgsql/14/data/base/16454/16489
```

Aller par exemple sur la 2è ligne, modifier `80 9F` en `FF FF`. Sortir avec Ctrl-X, confirmer la sauvegarde.

> Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

```
# SELECT * FROM pgbench_accounts ;
```

```
WARNING:  page verification failed, calculated checksum 62947 but expected 57715
ERROR:  invalid page in block 0 of relation base/16454/16489
```

> Tenter un `pgdumpall > /dev/null`.

```
$ pg_dumpall > /dev/null
pg_dump: WARNING:  page verification failed, calculated checksum 62947 but expected 57715
pg_dump: error: Dumping the contents of table "pgbench_accounts" failed: PQgetResult() failed.
pg_dump: error: Error message from server:
         ERROR:  invalid page in block 0 of relation base/16454/16489
pg_dump: error: The command was:
         COPY public.pgbench_accounts (aid, bid, abalance, filler) TO stdout;
pg_dumpall: error: pg_dump failed on database "pgbench", exiting
```

> Arrêter PostgreSQL.
> Voir ce que donne `pg_checksums` (en v12) ou `pg_verify_checksums` (en v11).

```
 # systemctl stop postgresql-14
```

```
$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data/  --check --progress
pg_checksums: error: checksum verification failed in file
"/var/lib/pgsql/14/data//base/16454/16489", block 0:
                              calculated checksum F5E3 but block contains E173
216/216 MB (100%) computed
Checksum operation completed
Files scanned:  1280
Blocks scanned: 27699
Bad checksums:  1
Data checksum version: 1
```

> Faire une copie de travail à froid du PGDATA.

> Protéger en écriture le PGDATA original.
> Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

Dans l'idéal, la copie devrait se faire vers un autre support, une corruption rend celui-ci suspect. Dans le cadre du TP, ceci suffira :

```
$ cp -upR /var/lib/pgsql/14/data/  /var/lib/pgsql/14/data.BACKUP/
$ chmod -R -w /var/lib/pgsql/14/data/
```

Dans `/var/lib/pgsql/14/data.BACKUP/pg_hba.conf` ne doit plus subsister que :

```
local   all             all                                     trust
```

> Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause).

```
$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data.BACKUP/  --disable
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums disabled in cluster
```

> Démarrer le cluster sur la copie avec `pg_ctl`.
> Que renvoie `SELECT * FROM pgbench_accounts LIMIT 100` ?

```
/usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ start

# SELECT *  FROM pgbench_accounts LIMIT 10;

ERROR:  out of memory
DÉTAIL : Failed on request of size 536888061 in memory context "printtup".
```

Ce ne sera pas forcément cette erreur, plus rien n'est sûr en cas de corruption. L'avantage des sommes de contrôle est justement d'avoir une erreur moins grave et plus ciblée.

Un `pg_dumpall` renverra le même message.

> Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

```
pgbench=# SET zero_damaged_pages TO on ;
SET
pgbench=# VACUUM FULL pgbench_accounts ;
VACUUM

pgbench=# SELECT *  FROM pgbench_accounts LIMIT 100;
```

```
 aid | bid | abalance |                                        filler
-----+-----+----------+------------------------------------------------------------------------------------------
   2 |   1 |        0 |
   3 |   1 |        0 |
   4 |   1 |        0 |
[...]

pgbench=# SELECT min(aid), max(aid), count(aid) FROM pgbench_accounts ;

 min |   max   | count
-----+---------+--------
   2 | 1000000 | 999999
```

Apparemment une ligne a disparu, celle portant la valeur 1 pour la clé. Il est rare que la perte soit aussi évidente !

1.10.2 CORRUPTION D'UN BLOC DE DONNÉES ET INCOHÉRENCES

> Consulter le format et le contenu de la table `pgbench_branches`.

Cette petite table ne contient que 10 valeurs :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
   3 |        0 |
   4 |        0 |
   5 |        0 |
   6 |        0 |
   7 |        0 |
   8 |        0 |
   9 |        0 |
  10 |        0 |
(10 lignes)
```

> Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

```
# SELECT pg_relation_filepath('pgbench_branches') ;

 pg_relation_filepath
----------------------
 base/16454/16490
```

> Arrêter PostgreSQL.
> Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
> En toute fin de fichier, remplacer le dernier `01` par un `FF`.
> Redémarrer PostgreSQL.

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ stop

$ hexedit /var/lib/pgsql/14/data.BACKUP/base/16454/16490

$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ start
```

> Compter le nombre de lignes dans `pgbench_branches`.
> Recompter après `SET enable_seqscan TO off ;`.
> Quelle est la bonne réponse ? Vérifier le contenu de la table.

Les deux décomptes sont contradictoires :

```
pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
     9

pgbench=# SET enable_seqscan TO off ;
SET

pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
    10
```

En effet, le premier lit la (petite) table directement, le second passe par l'index, comme un EXPLAIN le montrerait. Les deux objets diffèrent.

Et le contenu de la table est devenu :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
   3 |        0 |
   4 |        0 |
```

```
   5 |        0 |
   6 |        0 |
   7 |        0 |
   8 |        0 |
   9 |        0 |
(9 lignes)
```

Le 1 est devenu 255 (c'est notre première modification) mais la ligne 10 a disparu !

Les requêtes peuvent renvoyer un résultat incohérent avec leur critère :

```
pgbench=# SET enable_seqscan TO off;
SET
pgbench=# SELECT * FROM pgbench_branches
          WHERE    bid = 1 ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
```

> Qu'affiche `pageinspect` pour cette table ?

```
pgbench=# CREATE EXTENSION pageinspect ;

pgbench=# SELECT t_ctid, lp_off, lp_len, t_xmin, t_xmax, t_data
          FROM heap_page_items(get_raw_page('pgbench_branches',0));

 t_ctid | lp_off | lp_len | t_xmin | t_xmax |       t_data
--------+--------+--------+--------+--------+--------------------
 (0,1)  |   8160 |     32 |  63726 |      0 | \xff00000000000000
 (0,2)  |   8128 |     32 |  63726 |      0 | \x0200000000000000
 (0,3)  |   8096 |     32 |  63726 |      0 | \x0300000000000000
 (0,4)  |   8064 |     32 |  63726 |      0 | \x0400000000000000
 (0,5)  |   8032 |     32 |  63726 |      0 | \x0500000000000000
 (0,6)  |   8000 |     32 |  63726 |      0 | \x0600000000000000
 (0,7)  |   7968 |     32 |  63726 |      0 | \x0700000000000000
 (0,8)  |   7936 |     32 |  63726 |      0 | \x0800000000000000
 (0,9)  |   7904 |     32 |  63726 |      0 | \x0900000000000000
        |  32767 |    127 |        |        |
(10 lignes)
```

La première ligne indique bien que le 1 est devenu un 255.

La dernière ligne porte sur la première modification, qui a détruit les informations sur le `ctid`. Celle-ci est à présent inaccessible.

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

La documentation est sur https://docs.postgresql.fr/current/amcheck.html.

Une vérification complète se fait ainsi :

```
pgbench=# CREATE EXTENSION amcheck ;

pgbench=# SELECT bt_index_check (index => 'pgbench_branches_pkey',
                                 heapallindexed => true);
 bt_index_check
----------------


(1 ligne)

pgbench=# SELECT bt_index_parent_check (index => 'pgbench_branches_pkey',
                                        heapallindexed => true, rootdescend => true);
ERROR:  heap tuple (0,1) from table "pgbench_branches"
        lacks matching index tuple within index "pgbench_branches_pkey"
```

Un seul des problèmes a été repéré.

Un `REINDEX` serait ici une mauvaise idée : c'est la table qui est corrompue ! Les sommes de contrôle, là encore, auraient permis de cibler le problème très tôt.

> Exporter `pgbench_accounts`, définition des index comprise.
> Supprimer la table (il faudra supprimer `pgbench_history` aussi).
> Tenter de la réimporter.

```
$ pg_dump -d pgbench  -t pgbench_accounts -f /tmp/pgbench_accounts.dmp

$ psql pgbench -c 'DROP TABLE pgbench_accounts CASCADE'
NOTICE:  drop cascades to constraint pgbench_history_aid_fkey on table pgbench_history
DROP TABLE

$ psql pgbench < /tmp/pgbench_accounts.dmp
SET
SET
SET
SET
SET
 set_config
------------
```

```
(1 ligne)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 999999
ALTER TABLE
CREATE INDEX
ERROR:  insert or update on table "pgbench_accounts"
        violates foreign key constraint "pgbench_accounts_bid_fkey"
DÉTAIL : Key (bid)=(1) is not present in table "pgbench_branches".
```

La contrainte de clé étrangère entre les deux tables ne peut être respectée : `bid` est à 1 sur de nombreuses lignes de `pgbench_accounts` mais n'existe plus dans la table `pgbench_branches` ! Ce genre d'incohérence doit être recherchée très tôt pour ne pas surgir bien plus tard, quand on doit restaurer pour d'autres raisons.

NOTES

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **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

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.