Formation DBA1

PostgreSQL Administration

24.04

Table des matières

Sur ce document

Formation	Formation DBA1
Titre	PostgreSQL Administration
Révision	24.04
ISBN	978-2-38168-108-5
PDF	https://dali.bo/dba1_pdf
EPUB	https://dali.bo/dba1_epub
HTML	https://dali.bo/dba1_html
Slides	https://dali.bo/dba1_slides

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

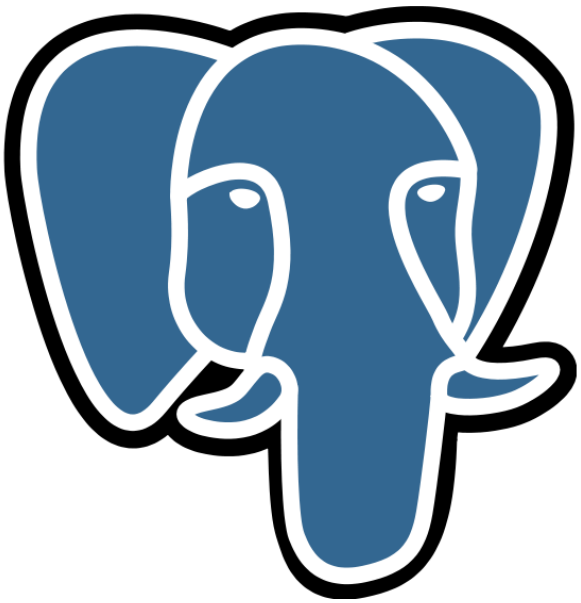
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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

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

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

1/ PostgreSQL : historique & communauté



1.1 PRÉAMBULE

- Quelle histoire !
  - parmi les plus vieux logiciels libres
  - et les plus sophistiqués
- Souvent cité comme exemple
  - qualité du code
  - indépendance des développeurs
  - réactivité de la communauté

L'histoire de PostgreSQL est longue, riche et passionnante. Au côté des projets libres Apache et Linux, PostgreSQL est l'un des plus vieux logiciels libres en activité et fait partie des SGBD les plus sophistiqués à l'heure actuelle.

Au sein des différentes communautés libres, PostgreSQL est souvent cité comme exemple à différents niveaux :

- qualité du code ;
- indépendance des développeurs et gouvernance du projet ;
- réactivité de la communauté ;
- stabilité et puissance du logiciel.

Tous ces atouts font que PostgreSQL est désormais reconnu et adopté par des milliers de grandes sociétés de par le monde.

1.1.1 Au menu

- Origines et historique du projet
- Versions et feuille de route
- Projets satellites
- Sponsors et références
- La communauté

Cette première partie est un tour d'horizon pour découvrir les multiples facettes du système de gestion de base de données libre PostgreSQL.

Les deux premières parties expliquent la genèse du projet et détaillent les différences entre les versions successives du logiciel. PostgreSQL est un des plus vieux logiciels libres ! Comprendre son histoire permet de mieux réaliser le chemin parcouru et les raisons de son succès.

Nous verrons ensuite certains projets satellites et nous listerons plusieurs utilisateurs renommés et cas d'utilisations remarquables.

Enfin, nous terminerons par une découverte de la communauté.

1.2 UN PEU D'HISTOIRE...

- La licence
- L'origine du nom
- Les origines du projet
- Les principes

1.2.1 Licence

- Licence PostgreSQL
  - libre (BSD/MIT)
  - https://www.postgresql.org/about/licence/
- Droit, sans coûts de licence, de :
  - utiliser, copier, modifier, distribuer (et même revendre)
- Reconnue par l'Open Source Initiative
- Utilisée par un grand nombre de projets de l'écosystème

PostgreSQL est distribué sous une licence spécifique, combinant la licence BSD et la licence MIT. Cette licence spécifique est reconnue comme une licence libre par l'Open Source Initiative[1].

Cette licence vous donne le droit de distribuer PostgreSQL, de l'installer, de le modifier… et même de le vendre. Certaines sociétés, comme EnterpriseDB et PostgresPro, produisent leur version propriétaire de PostgreSQL de cette façon.

PostgreSQL n'est pas pour autant complètement gratuit : il peut y avoir des frais et du temps de formation, des projets de migration depuis d'autres bases, ou d'intégration des différents outils périphériques indispensables en production.

Cette licence a ensuite été reprise par de nombreux projets de la communauté : pgAdmin, pgCluu, pgstat, etc.

[1] https://opensource.org/licenses/PostgreSQL

1.2.2 PostgreSQL ?!?!

- 1985 : Michael Stonebraker recode Ingres
- post « ingres » ⇒ post~~in~~gres ⇒ postgres
- postgres ⇒ PostgreSQL

PostgreSQL a une origine universitaire.

L'origine du nom PostgreSQL remonte au système de gestion de base de données Ingres, développé à l'université de Berkeley par Michael Stonebraker. En 1985, il prend la décision de reprendre le développement à partir de zéro et nomme ce nouveau logiciel **Postgres**, comme raccourci de post-Ingres.

En 1995, avec l'ajout du support du langage SQL, Postgres fut renommé **Postgres95** puis **PostgreSQL**.

Aujourd'hui, le nom officiel est « PostgreSQL » (prononcé « post - gresse - Q - L »). Cependant, le nom « Postgres » reste accepté.

Pour aller plus loin :

- Fil de discussion sur les listes de discussion[2] ;
- Article sur le wiki officiel[3].

1.2.3 Principes fondateurs

- Sécurité des données (ACID[4])
- Respect des normes (ISO SQL)
- Portabilité
- Fonctionnalités intéressant le plus grand nombre
- Performances
  - si pas de péril pour les données
- Simplicité du code
- Documentation

Depuis son origine, PostgreSQL a toujours privilégié la stabilité et le respect des standards plutôt que les performances.

La sécurité des données est un point essentiel. En premier lieu, un utilisateur doit être certain qu'à partir du moment où il a exécuté l'ordre `COMMIT` d'une transaction, les données modifiées relatives à cette transaction se trouvent bien sur disque et que même un crash ne pourra pas les faire disparaître. PostgreSQL est très attaché à ce concept et fait son possible pour forcer le système d'exploitation à ne pas conserver les données en cache, mais à les écrire sur disque dès l'arrivée d'un `COMMIT`.

L'intégrité des données, et le respect des contraintes fonctionnelles et techniques qui leur sont imposées, doivent également être garanties par le moteur à tout moment, quoi que fasse l'utilisateur. Par exemple, insérer 1000 caractères dans un champ contraint à 200 caractères maximum doit mener à une erreur explicite et non à l'insertion des 200 premiers caractères en oubliant les autres, comme cela s'est vu ailleurs. De même, un champ avec le type `date` ne contiendra jamais un 31 février, et un champ `NOT NULL` ne sera jamais vide. Tout ceci est formalisé par les propriétés (ACID[5]) que possèdent toute bonne base de données relationnelle.

Le respect des normes est un autre principe au cœur du projet. Les développeurs de PostgreSQL cherchent à coller à la norme SQL[6] le plus possible. PostgreSQL n'est pas compatible à cette norme à 100 %, aucun moteur ne l'est, mais il cherche à s'en approcher. Tout nouvel ajout d'une syntaxe ne sera accepté que si la syntaxe de la norme est ajoutée. Des extensions sont acceptées pour différentes raisons (performances, fonctionnalités en avance sur le comité de la norme, facilité de transition d'un moteur de bases de données à un autre) mais si une fonctionnalité existe dans la norme, une syntaxe différente ne peut être acceptée que si la syntaxe de la norme est elle-aussi présente.

La portabilité est importante : PostgreSQL tourne sur l'essentiel des systèmes d'exploitation : Linux (plate-forme à privilégier), macOS, les Unix propriétaires, Windows… Tout est fait pour que cela soit encore le cas dans le futur.

Ajouter des fonctionnalités est évidemment l'un des buts des développeurs de PostgreSQL. Cependant, comme il s'agit d'un projet libre, rien n'empêche un développeur de proposer une fonctionnalité, de la faire intégrer, puis de disparaître laissant aux autres la responsabilité de la corriger le cas échéant. Comme le nombre de développeurs de PostgreSQL est restreint, il est important que les fonctionnalités ajoutées soient vraiment utiles au plus grand nombre pour justifier le coût potentiel du débogage. Donc ne sont ajoutées dans PostgreSQL que ce qui est vraiment le cœur du moteur de bases de données et que ce qui sera utilisé vraiment par le plus grand nombre. Une fonctionnalité qui ne sert que une à deux personnes aura très peu de chances d'être intégrée. (Le système des extensions offre une élégante solution aux problèmes très spécifiques.)

Les performances ne viennent qu'après tout ça. En effet, rien ne sert d'avoir une modification du code qui permet de gagner énormément en performances si cela met en péril le stockage des données. Cependant, les performances de PostgreSQL sont excellentes et le moteur permet d'opérer des centaines de tables, des milliards de lignes pour plusieurs téraoctets de données, sur une seule instance, pour peu que la configuration matérielle soit correctement dimensionnée.

La simplicité du code est un point important. Le code est relu scrupuleusement par différents contributeurs pour s'assurer qu'il est facile à lire et à comprendre. En effet, cela facilitera le débogage plus tard si cela devient nécessaire.

[5] https://dali.bo/a2_html#ACID
[6] https://fr.wikipedia.org/wiki/Structured_Query_Language

Enfin, la documentation est là-aussi un point essentiel dans l'admission d'une nouvelle fonctionnalité. En effet, sans documentation, peu de personnes pourront connaître cette fonctionnalité. Très peu sauront exactement ce qu'elle est supposée faire, et il serait donc très difficile de déduire si un problème particulier est un manque actuel de cette fonctionnalité ou un bug.

Tous ces points sont vérifiés à chaque relecture d'un patch (nouvelle fonctionnalité ou correction).

1.2.4 Origines

- Années 1970 : Michael Stonebraker développe **Ingres** à Berkeley
- 1985 : **Postgres** succède à Ingres
- 1995 : Ajout du langage SQL
- 1996 : Libération du code : Postgres devient **PostgreSQL**
- 1996 : Création du PostgreSQL Global Development Group

L'histoire de PostgreSQL remonte au système de gestion de base de données Ingres[7], développé dès 1973 à l'Université de Berkeley (Californie) par Michael Stonebraker[8].

Lorsque ce dernier décide en 1985 de recommencer le développement de zéro, il nomme le logiciel Postgres, comme raccourci de post-Ingres. Des versions commencent à être diffusées en 1989, puis commercialisées.

Postgres utilise alors un langage dérivé de QUEL[9], hérité d'Ingres, nommé POSTQUEL[10]. En 1995, lors du remplacement par le langage SQL par Andrew Yu and Jolly Chen, deux étudiants de Berkeley, Postgres est renommé Postgres95.

En 1996, Bruce Momijan et Marc Fournier convainquent l'Université de Berkeley de libérer complètement le code source. Est alors fondé le PGDG (*PostgreSQL Development Group*), entité informelle — encore aujourd'hui — regroupant l'ensemble des contributeurs. Le développement continue donc hors tutelle académique (et sans son fondateur historique Michael Stonebraker) : PostgreSQL 6.0 est publié début 1997.

Plus d'informations :

- Page associée sur le site officiel[11].

[7] https://en.wikipedia.org/wiki/Ingres_(database)
[8] https://en.wikipedia.org/wiki/Michael_Stonebraker
[9] https://en.wikipedia.org/wiki/QUEL_query_languages
[10] La trace se retrouve encore dans le nom de la librairie C pour les clients, la **libpq**.

1.2.5 Apparition de la communauté internationale

- ~ 2000: Communauté japonaise (JPUG)
- 2004 : PostgreSQLFr
- 2006 : SPI
- 2007 : Communauté italienne
- 2008 : PostgreSQL Europe et US
- 2009 : Boom des PGDay
- 2011 : Postgres Community Association of Canada
- 2017 : Community Guidelines
- …et ça continue

Les années 2000 voient l'apparition de communautés locales organisées autour d'association ou de manière informelle. Chaque communauté organise la promotion, la diffusion d'information et l'entraide à son propre niveau.

En 2000 apparaît la communauté japonaise (JPUG). Elle dispose déjà d'un grand groupe, capable de réaliser des conférences chaque année, d'éditer des livres et des magazines. Elle compte, au dernier recensement connu, plus de 3000 membres.

En 2004 naît l'association française (loi 1901) appelée PostgreSQL Fr. Cette association a pour but de fournir un cadre légal pour pouvoir participer à certains événements comme Solutions Linux, les RMLL ou d'en organiser comme le pgDay.fr (qui a déjà eu lieu à Toulouse, Nantes, Lyon, Toulon, Marseille). Elle permet aussi de récolter des fonds pour aider à la promotion de PostgreSQL.

En 2006, le PGDG intègre Software in the Public Interest, Inc.(SPI)[12], une organisation à but non lucratif chargée de collecter et redistribuer des financements. Elle a été créée à l'initiative de Debian et dispose aussi de membres comme LibreOffice.org.

Jusque là, les événements liés à PostgreSQL apparaissaient plutôt en marge de manifestations, congrès, réunions… plus généralistes. En 2008, douze ans après la création du projet, des associations d'utilisateurs apparaissent pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. PostgreSQL UK organise une journée de conférences à Londres, PostgreSQL Fr en organise une à Toulouse. Des « sur-groupes » apparaissent aussi pour aider les groupes locaux : PGUS rassemble les différents groupes américains, plutôt organisés géographiquement, par État ou grande ville. De même, en Europe, est fondée PostgreSQL Europe, association chargée d'aider les utilisateurs de PostgreSQL souhaitant mettre en place des événements. Son principal travail est l'organisation d'un événement majeur en Europe tous les ans : pgconf.eu[13], d'abord à Paris en 2009, puis dans divers pays d'Europe jusque Milan en 2019. Cependant, elle aide aussi les communautés allemande, française et suédoise à monter leur propre événement (respectivement PGConf.DE[14], pgDay Paris[15]

[12] https://fr.wikipedia.org/wiki/Software_in_the_Public_Interest
[13] https://pgconf.eu/
[14] https://pgconf.de/
[15] https://pgday.paris/

et Nordic PGday[16]).

Dès 2010, nous dénombrons plus d'une conférence par mois consacrée uniquement à PostgreSQL dans le monde. Ce mouvement n'est pas prêt de s'arrêter :

- communauté japonaise[17] ;
- communauté francophone[18] ;
- communauté italienne[19] ;
- communauté européenne[20] ;
- communauté américaine (États-Unis)[21].

En 2011, l'association Postgres Community Association of Canada voit le jour[22]. Elle est créée par quelques membres de la *Core Team* pour gérer le nom déposé PostgreSQL, le logo, le nom de domaine sur Internet, etc.

Vu l'émergence de nombreuses communautés internationales, la communauté a décidé d'écrire quelques règles pour ces communautés. Il s'agit des *Community Guidelines*, apparues en 2017, et disponibles sur le site officiel[23].

1.2.6 Progression du code

- 1,6 millions de lignes
  - dont 1/4 de commentaires
  - le reste surtout en C
- Nombres de commit par mois :

[16] https://nordicpgday.org/
[17] https://www.postgresql.jp/
[18] https://www.postgresql.fr/
[19] https://www.itpug.org/
[20] https://www.postgresql.eu/
[21] https://www.postgresql.us/
[22] https://www.postgresql.org/message-id/4DC440BE.5040104%40agliodbs.com%3E
[23] https://www.postgresql.org/community/recognition/

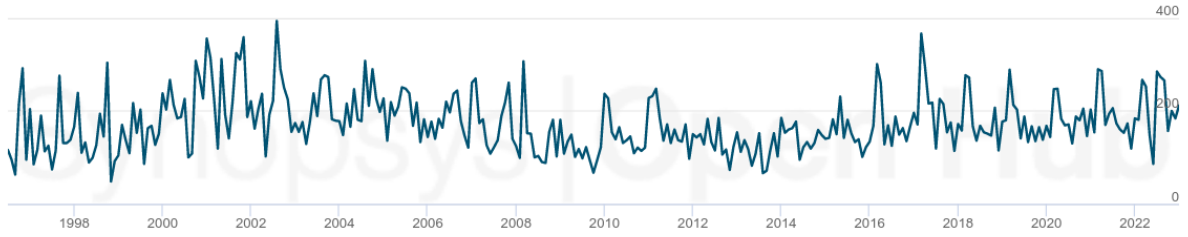


Figure 1/ .1: Évolution du nombre de commit dans le dépôt PostgreSQL

Le dépôt principal de PostgreSQL a été un dépôt CVS, passé depuis à git[24]. Il est en accès public en lecture.

Le graphe ci-dessus (source[25]) représente l'évolution du nombre de commit dans les sources de PostgreSQL. L'activité ne se dément pas. Le plus intéressant est certainement de noter que l'évolution est constante. Il n'y a pas de gros pic, ni dans un sens, ni dans l'autre.

Début 2023, PostgreSQL est composé d'1,6 millions de lignes de code, dont un quart de commentaires. Ce ratio montre que le code est très commenté, très documenté. Ceci fait qu'il est facile à lire, et donc pratique à déboguer. Et le ratio ne change pas au fil des ans. Le code est essentiellement en C, pour environ 200 développeurs actifs, à environ 200 commits par mois ces dernières années.

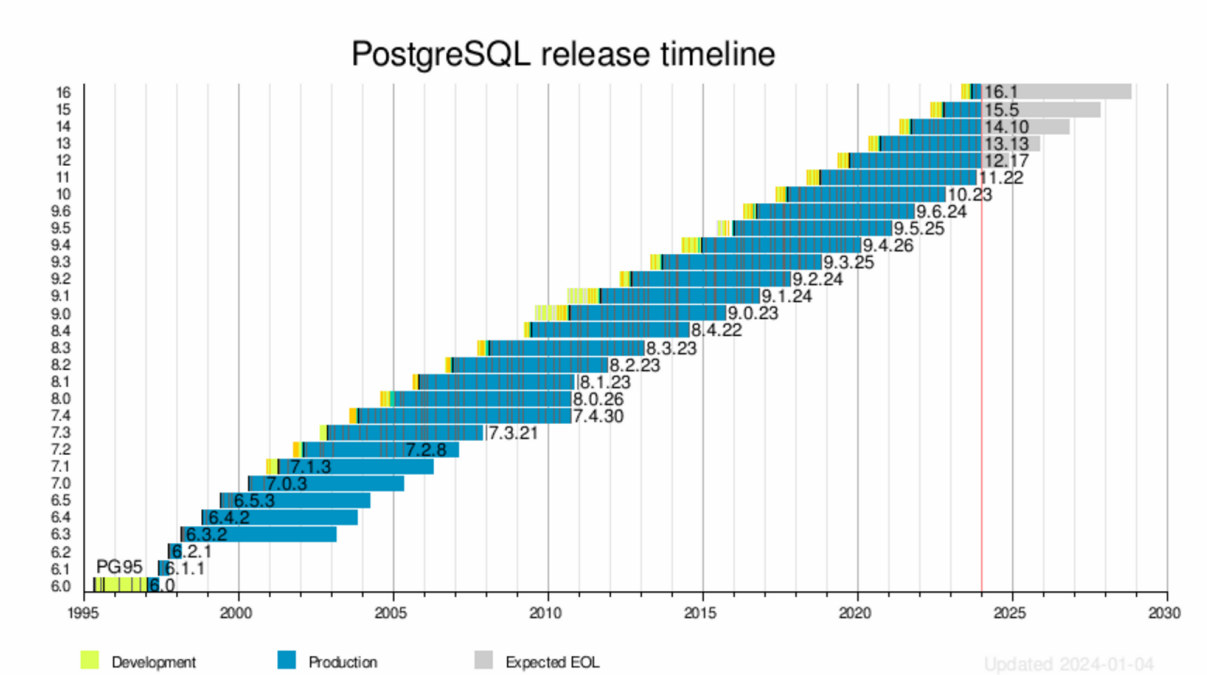
[24] https://git.postgresql.org/
[25] https://www.openhub.net/p/postgres/analyses/latest/

1.3 LES VERSIONS DE POSTGRESQL

Quelle version utiliser ?

- Historique
- Numérotation
- Mises à jour mineures et majeures
- Les versions courantes
- Quelle version en production ?
- Les forks & dérivés

1.3.1 Historique



Sources : page Wikipédia de PostgreSQL[26] et PostgreSQL Versioning Policy[27]

[26] https://en.wikipedia.org/wiki/PostgreSQL
[27] https://www.postgresql.org/support/versioning/

1.3.2 Versions & fonctionnalités

- 1996 : v6.0 -> première version publiée
- 2003 : v7.4 -> première version *réellement* stable
- 2005 : v8.0 -> arrivée sur Windows
- 2008 : v8.3 -> performances et fonctionnalités, organisation (commitfests)
- 2010 : v9.0 -> réplication physique
- 2016 : v9.6 -> parallélisation
- 2017 : v10 -> réplication logique, partitionnement déclaratif
- 2023 : v16 -> performances, fonctionnalités, administration…

La version 7.4 est la première version réellement stable. La gestion des journaux de transactions a été nettement améliorée, et de nombreuses optimisations ont été apportées au moteur.

La version 8.0 marque l'entrée tant attendue de PostgreSQL dans le marché des SGDB de haut niveau, en apportant des fonctionnalités telles que les tablespaces, les routines stockées en Java, le *Point In Time Recovery*, ainsi qu'une version native pour Windows.

La version 8.3 se focalise sur les performances et les nouvelles fonctionnalités. C'est aussi la version qui a causé un changement important dans l'organisation du développement pour encourager les contributions : gestion des commitfests, création de l'outil web associé, etc.

Les versions 9.x sont axées réplication physique. La 9.0 intègre un système de réplication asynchrone asymétrique. La version 9.1 ajoute une réplication synchrone et améliore de nombreux points sur la réplication (notamment pour la partie administration et supervision). La version 9.2 apporte la réplication en cascade. La 9.3 et la 9.4 ajoutent quelques améliorations supplémentaires. La version 9.4 intègre surtout les premières briques pour l'intégration de la réplication logique dans PostgreSQL. La version 9.6 apporte la parallélisation, ce qui était attendu par de nombreux utilisateurs.

La version 10 propose beaucoup de nouveautés, comme une amélioration nette de la parallélisation et du partitionnement (le partitionnement déclaratif complète l'ancien partitionnement par héritage), mais aussi l'ajout de la réplication logique.

Les améliorations des versions 11 à 16 sont plus incrémentales, et portent sur tous les plans. Le partitionnement déclaratif et la réplication logique sont progressivement améliorés, en performances comme en facilité de développement. Les performances s'améliorent encore grâce à la compilation *Just In Time*, la parallélisation de plus en plus d'opérations, les index couvrants, l'affinement des statistiques. La facilité d'administration s'améliore : nouvelles vues système, rôles supplémentaires pour réduire l'utilisation du superutilisateur, outillage de réplication, activation des sommes de contrôle sur une instance existante.

Il est toujours possible de télécharger les sources depuis la version 1.0 jusqu'à la version courante sur postgresql.org[28].

[28] https://www.postgresql.org/ftp/source/

1.3.3 Numérotation

- Version récentes (10+)
  - X : version majeure (10, 11, … 16)
  - X.Y : version mineure (14.8, 15.3)
- Avant la version 10 (toutes périmées !)
  - X.Y : version majeure (8.4, 9.6)
  - X.Y.Z : version mineure (9.6.24)

Une version majeure apporte de nouvelles fonctionnalités, des changements de comportement, etc. Une version majeure sort généralement tous les ans à l'automne. Une migration majeure peut se faire directement depuis n'importe quelle version précédente. Le numéro est incrémenté chaque année (version 12 en 2019, version 16 en 2023).

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Les publications de versions mineures sont plus fréquentes que celles de versions majeures, avec un rythme de sortie trimestriel, sauf bug majeur ou faille de sécurité. Chaque bug est corrigé dans toutes les versions stables actuellement maintenues par le projet. Le numéro d'une version mineure porte deux chiffres. Par exemple, en mai 2023 sont sorties les versions 15.3, 14.8, 13.11, 12.15 et 11.20.

Avant la version 10, les versions majeures annuelles portaient deux chiffres : 9.0 en 2010, 9.6 en 2016. Les mineures avaient un numéro de plus (par exemple 9.6.24). Cela a entraîné quelques confusions, d'où le changement de numérotation. Il va sans dire que ces versions sont totalement périmées et ne sont plus supportées, mais beaucoup continuent de fonctionner.

1.3.4 Mises à jour mineure

De M.m à M.m+n :

- En général chaque trimestre
- Et sans souci
  - *Release notes*
  - tests
  - mise à jour des binaires
  - redémarrage

Une mise à jour mineure consiste à mettre à jour vers une nouvelle version de la même branche majeure, par exemple de 14.8 à 14.9, ou de 16.0 à 16.1 (mais pas d'une version 14.x à une version 16.x). Les mises à jour des versions mineures sont cumulatives : vous pouvez mettre à jour une instance 15.0 en version 15.5 sans passer par les versions 15.1 à 15.4 intermédiaires.

En général, les mises à jour mineures se font sans souci et ne nécessitent que le remplacement des binaires et un redémarrage (et donc une courte interruption). Les fichiers de données conservent le même format. Des opérations supplémentaires sont possibles mais rarissimes. Mais comme pour toute mise à jour, il convient d'être prudent sur d'éventuels effets de bord. En particulier, il faudra lire les *Release Notes* et, si possible, effectuer les tests ailleurs qu'en production.

1.3.5 Versions courantes

- 1 version majeure par an
  - maintenue 5 ans
- Dernières mises à jour mineures[29] (au 9/11/2023) :
  - version 11.22[30] (dernière !)
  - version 12.17[31]
  - version 13.13[32]
  - version 14.10[33]
  - version 15.5[34]
  - version 16.1[35]
- Prochaine sortie de versions mineures prévue : 8 février 2024

La philosophie générale des développeurs de PostgreSQL peut se résumer ainsi :

« Notre politique se base sur la qualité, pas sur les dates de sortie. »

Toutefois, même si cette philosophie reste très présente parmi les développeurs, en pratique une version stable majeure paraît tous les ans, habituellement à l'automne. Pour ne pas sacrifier la qualité des versions, toute fonctionnalité supposée insuffisamment stable est repoussée à la version suivante. Il est déjà arrivé que la sortie de la version majeure soit repoussée à cause de bugs inacceptables.

La tendance actuelle est de garantir un support pour chaque version majeure pendant une durée minimale de 5 ans. Ainsi ne sont plus supportées les versions 10 depuis novembre 2022 et 11 depuis novembre 2023. Il n'y aura pour elles plus aucune mise à jour mineure, donc plus de correction de bug ou de faille de sécurité. Le support de la dernière version majeure, la 16, devrait durer jusqu'en 2028.

Pour plus de détails :

- Politique de versionnement[36] ;
- Dates prévues des futures versions[37].

1.3.6 Versions 9.4 à 11

- `jsonb`
- Row Level Security
- Index BRIN, bloom
- Fonctions OLAP
- Parallélisation
- SQL/MED : accès distants
- Réplication logique
- Partitionnement déclaratif
- Réduction des inconvénients de MVCC
- JIT
- Index couvrants

Ces versions ne sont plus supportées !

La version 9.4 (décembre 2014) a apporté le type `jsonb`, binaire, facilitant la manipulation des objets en JSON.

La 9.5 parue en janvier 2016 apportait notamment les index BRIN et des possibilités OLAP plus avancées que `GROUP BY`. Pour plus de détails :

- Page officielle des nouveautés de la version 9.5[38] ;
- Workshop Dalibo sur la version 9.5[39].

En 9.6, la nouvelle fonctionnalité majeure est certainement la parallélisation de certaines parties de l'exécution d'une requête. Le `VACUUM FREEZE` devient beaucoup moins gênant.

- Page officielle des nouveautés de la version 9.6[40] ;
- Workshop Dalibo sur la version 9.6[41].

En version 10, les fonctionnalités majeures sont l'intégration de la réplication logique et le partitionnement déclaratif, longtemps attendus, améliorés dans les versions suivantes. Sont notables aussi les tables de transition ou les améliorations sur la parallélisation.

La version 10 a aussi été l'occasion de renommer plusieurs répertoires et fonctions système, et même des outils. Attention donc si vous rencontrez des requêtes ou des scripts adaptés aux versions précédentes. Entre autres :

- le répertoire `pg_xlog` est devenu `pg_wal` ;
- le répertoire `pg_clog` est devenu `pg_xact` ;
- dans les noms de fonctions, `xlog` a été remplacé par `wal` (par exemple `pg_switch_xlog` est devenue `pg_switch_wal`) ;
- toujours dans les fonctions, `location` a été remplacé par `lsn`.

Pour plus de détails :

- Page officielle des nouveautés de la version 10[42] ;
- Workshop Dalibo sur la version 10[43].

La version 11 (octobre 2018) améliore le partitionnement de la version 10, le parallélisme, la réplication logique… et de nombreux autres points. Elle comprend aussi une première version du JIT (*Just In Time compilation*) pour accélérer les requêtes les plus lourdes en CPU, ou encore les index couvrants.

Pour plus de détails, voir notre workshop sur la version 11[44].

[38] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.5
[39] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.5
[40] https://wiki.postgresql.org/wiki/NewIn96
[41] https://github.com/dalibo/workshops/tree/master/fr
[42] https://wiki.postgresql.org/wiki/New_in_postgres_10
[43] https://dali.bo/workshop10_pdf
[44] https://dali.bo/workshop11_pdf

1.3.7 Version 12

- Octobre 2019 - Novembre 2024
- Amélioration du partitionnement déclaratif
- Amélioration des performances
  - sur la gestion des index
  - sur les CTE (option MATERIALIZED)
- Colonnes générées
- Nouvelles vues de visualisation de la progression des commandes
- Refonte de la configuration de la réplication

La version 12 est sortie le 3 octobre 2019. Elle améliore de nouveau le partitionnement et elle fait surtout un grand pas au niveau des performances et de la supervision.

Le fichier `recovery.conf` (pour la réplication et les restaurations physiques) disparaît. Il est maintenant intégré au fichier `postgresql.conf`. Une source fréquente de ralentissement disparaît, avec l'intégration des CTE (clauses `WITH`) dans la requête principale. Des colonnes d'une table peuvent être automatiquement générées à partir d'autres colonnes.

Pour plus de détails, voir notre workshop sur la version 12[45].

1.3.8 Version 13

- Septembre 2020 - Septembre 2025
- Améliorations :
  - partitionnement déclaratif
  - réplication logique
- Amélioration des performances :
  - index B-tree, objet statistique, tri et agrégat
- Amélioration de l'autovacuum et du VACUUM :
  - gestion complète des tables en insertion seule
  - traitement parallélisé des index lors d'un VACUUM
- Amélioration des sauvegardes :
  - génération d'un fichier manifeste, outil pg_verifybackup
- Nouvelles vues de progression de commandes :
  - pg_stat_progress_basebackup, pg_stat_progress_analyze

La version 13 est sortie le 24 septembre 2020. Elle est remplie de nombreuses petites améliorations sur différents domaines : partitionnement déclaratif, autovacuum, sauvegarde, etc. Les performances sont aussi améliorées grâce à un gros travail sur l'optimiseur, ou la réduction notable de la taille de certains index.

Pour plus de détails, voir notre workshop sur la version 13[46].

1.3.9 Version 14

- Septembre 2021 - Novembre 2026
- Nouvelles vues système & améliorations
  - `pg_stat_progress_copy`, `pg_stat_wal`, `pg_lock.waitstart`, `query_id`…
- Lecture asynchrone des tables distantes
- Paramétrage par défaut adapté aux machines plus récentes
- Améliorations diverses :
  - réplications physique et logique
  - quelques facilités de syntaxe (triggers, tableaux en PL/pgSQL)
- Performances :
  - connexions en lecture seule plus nombreuses
  - index…

La version 14 est remplie de nombreuses petites améliorations sur différents domaines listés ci-dessus.

Pour plus de détails, voir notre workshop sur la version 14[47].

1.3.10 Version 15

- Octobre 2022 - Novembre 2027
- Nombreuses améliorations incrémentales
  - dont en réplication logique
- Commande `MERGE`
- Performances :
  - `DISTINCT` parallélisable
  - pg_dump & sauvegardes, recovery, partitionnement
- Changements notables :
  - `public` n'est plus accessible en écriture à tous
  - sauvegarde PITR exclusive disparaît

La version 15 est également une mise à jour sans grande nouveauté fracassante, mais contenant de très nombreuses améliorations et optimisations sur de nombreux plans, comme par exemple la commande `MERGE` ou l'accélération du *recovery* sur une reprise de restauration.

Signalons deux changements de comportement importants : pour renforcer la sécurité, le schéma `public` n'est plus accessible en écriture par défaut à tous les utilisateurs ; et la sauvegarde physique en mode exclusif n'est plus disponible.

Pour plus de détails, voir notre workshop sur la version 15[48].

1.3.11 Version 16

- Septembre 2023 - Novembre 2028
- Plus de tris incrémentaux (`DISTINCT` …)
- Réplication logique depuis un secondaire
- Expressions régulières dans `pg_hba.conf`
- Vues systèmes améliorées : `pg_stat_io` …
- Compression lz4 ou zstd pour pg_dump
- Optimisation et améliorations diverses (parallélisation…)

La version 16 est parue le 14 septembre 2023. Sa version 16.1 est considérée comme bonne pour la production. Là encore, les améliorations sont incrémentales.

On notera la possibilité de rajouter des expressions régulières dans `pg_hba.conf` pour faciliter la gestion des accès. En réplication logique, un abonnement peut se faire auprès d'un serveur secondaire. La réplication logique peut devenir parallélisable. `pg_dump` acquiert des algorithmes de compression plus modernes. Le travail de parallélisation de nouveaux nœuds se poursuit. Une nouvelle vue de suivi des entrées-sorties apparaît : `pg_stat_io`.

Pour plus de détails :

- workshop Dalibo sur la version 16[49] ;
- blog Dalibo[50] ;
- présentation de Magnus Hagander au PGDay UK 2023[51]

1.3.12 Petit résumé

- Versions 7.x :
  - fondations
  - durabilité
- Versions 8.x :
  - fonctionnalités
  - performances
- Versions 9.x :
  - réplication physique
  - extensibilité
- Versions 10 à 16 :
  - réplication logique
  - parallélisation
  - partitionnement
- … et la 17 est en développement

Si nous essayons de voir cela avec de grosses mailles, les développements des versions 7 ciblaient les fondations d'un moteur de bases de données stable et durable. Ceux des versions 8 avaient pour but

[49] https://dali.bo/workshop16_pdf
[50] https://blog.dalibo.com/2023/09/15/release_postgresql_16.html
[51] https://www.hagander.net/talks/PostgreSQL%2016.pdf

de rattraper les gros acteurs du marché en fonctionnalités et en performances. Enfin, pour les versions 9, on est plutôt sur la réplication et l'extensibilité.

La version 10 se base principalement sur la parallélisation des opérations (développement mené principalement par EnterpriseDB) et la réplication logique (par 2ndQuadrant). Les versions 11 à 16 améliorent ces deux points, entre mille autres améliorations en différents points du moteur, notamment les performances et la facilité d'administration.

1.3.13 Quelle version utiliser en production ?

- 11 et inférieures
  - **Danger !**
  - planifier une migration urgemment !
- 12, 13, 14, 15, 16
  - mises à jour mineures uniquement
- 16
  - nouvelles installations et nouveaux développements
- Tableau comparatif des versions[52]

Si vous avez une version 11 ou inférieure, planifiez le plus rapidement possible une migration vers une version plus récente, comme la 15 ou la 16. La 10 n'est plus maintenue depuis 2022, la 11 ne le sera plus dès novembre 2023. Elles fonctionneront toujours aussi bien, mais il n'y aura plus de correction de bug, y compris pour les failles de sécurité ! Si vous utilisez ces versions ou des versions antérieures, il est impératif d'étudier une migration de version dès que possible.

Les versions 12 à 16 sont celles recommandées pour une production. Le plus important est d'appliquer les mises à jour correctives. Attention, la version 12 ne sera plus supportée dès novembre 2024.

La version 16 est officiellement stable. Cette version est donc celle conseillée pour les nouvelles installations en production. Par expérience, quand une version x.0 paraît à l'automne, elle est généralement stable. Nombre de DBA préfèrent prudemment attendre les premières mises à jour mineures (en novembre généralement) pour la mise en production. Cette prudence est à mettre en balance avec l'intérêt pour les nouvelles fonctionnalités. Pour plus de détails, voir le tableau comparatif des versions[53].

[53] https://www.postgresql.org/about/featurematrix

1.3.14 Versions dérivées / Forks

Entre de nombreux autres :

- Compatibilité Oracle :
  - EnterpriseDB
- Data warehouse :
  - Greenplum, Netezza
- Forks :
  - Amazon RedShift, Aurora…
- Extensions :
  - Citus
  - timescaledb
- Packages avec des outils & support
- Bases compatibles

Il existe de nombreuses versions dérivées de PostgreSQL. Elles sont en général destinées à des cas d'utilisation très spécifiques et offrent des fonctionnalités non proposées par la version communautaire. Leur code est souvent fermé et nécessite l'acquisition d'une licence payante. La licence de PostgreSQL permet cela, et le phénomène existait déjà dès les années 1990 avec divers produits commerciaux comme Illustra.

Modifier le code de PostgreSQL a plusieurs conséquences négatives. Certaines fonctionnalités de PostgreSQL peuvent être désactivées. Il est donc difficile de savoir ce qui est réellement utilisable. De plus, chaque nouvelle version mineure demande une adaptation de leur ajout de code. Chaque nouvelle version majeure demande une adaptation encore plus importante de leur code. C'est un énorme travail, qui n'apporte généralement pas suffisamment de plus-value à la société éditrice pour qu'elle le réalise. La seule société qui le fait de façon complète est EnterpriseDB, qui arrive à proposer des mises à jour régulièrement. Par contre, si on revient sur l'exemple de Greenplum, ils sont restés bloqués pendant un bon moment sur la version 8.0. Ils ont cherché à corriger cela. Fin 2021, Greenplum 6.8 est au niveau de la version 9.4[54], version considérée alors comme obsolète par la communauté depuis plus de deux ans. En janvier 2023, Greenplum 7.0 bêta n'est toujours parvenu qu'au niveau de PostgreSQL 12.12…

Rien ne dit non plus que la société ne va pas abandonner son fork. Par exemple, il a existé quelques forks créés lorsque PostgreSQL n'était pas disponible en natif sous Windows : ces forks ont majoritairement disparu lors de l'arrivée de la version 8.0, qui proposait exactement cette fonctionnalité dans la version communautaire.

[54] https://web.archive.org/web/20211018012643/https://docs.greenplum.org/6-8/security-guide/topics/preface.html

Il y a eu aussi quelques forks créés pour gérer la réplication. Là aussi, la majorité de ces forks ont été abandonnés (et leurs clients avec) quand PostgreSQL a commencé à proposer de la réplication en version 9.0. Cependant, tous n'ont pas été abandonnés, en tout cas immédiatement. Par exemple, Slony est resté très vivant parce qu'il proposait des fonctionnalités que PostgreSQL n'avait pas encore à l'époque (notamment la réplication entre versions majeures différentes, et la réplication partielle). Ces fonctionnalités étant arrivées avec PostgreSQL 10, Slony est en fort déclin (tout comme Londiste, qui a été plus ou moins abandonné quand Skype a été racheté par Microsoft, ou Bucardo qu'on ne voit actuellement nulle part, du moins en France).

Il faut donc bien comprendre qu'à partir du moment où un utilisateur choisit une version dérivée, il dépend fortement (voire uniquement) de la bonne volonté de la société éditrice pour continuer son produit, le mettre à jour avec les dernières corrections et les dernières nouveautés de la version communautaire. Pour éviter ce problème, certaines sociétés ont décidé de transformer leur fork en une extension. C'est beaucoup plus simple à maintenir et n'enferme pas leurs utilisateurs. C'est le cas par exemple de citusdata (racheté par Microsoft) pour son extension de *sharding* ; ou encore de TimescaleDB, avec leur extension spécialisée dans les séries temporelles.

Dans les exemples de fork dédiés aux entrepôts de données, les plus connus historiquement sont Greenplum, de Pivotal (racheté par WMWare), et Netezza, d'IBM. Autant Greenplum tente de se raccrocher au PostgreSQL communautaire toutes les quelques années, autant ce n'est pas le cas de Netezza, optimisé pour du matériel dédié, et qui a forké de PostgreSQL 7.2.

Amazon, avec notamment les versions Redshift[55] ou Aurora, a la particularité de modifier profondément PostgreSQL pour l'adapter à son infrastructure, mais ne diffuse pas ses modifications. Même si certaines incompatibilités sont listées, il est très difficile de savoir où ils en sont et l'impact qu'a leurs modifications.

EDB Postgres Advanced Server d'EnterpriseDB permet de faciliter la migration depuis Oracle. Son code est propriétaire et soumis à une licence payante. Certaines fonctionnalités finissent par atterrir dans le code communautaire (une fois qu'EnterpriseDB le souhaite et que la communauté a validé l'intérêt de cette fonctionnalité et sa possible intégration).

BDR, anciennement de 2nd Quadrant, maintenant EnterpriseDB, est un *fork* visant à fournir une version multimaître de PostgreSQL, mais le code a été refermé dans les dernières versions. Il est très difficile de savoir où ils en sont. Son utilisation implique de prendre le support chez eux.

La société russe Postgres Pro, tout comme EnterpriseDB, propose diverses fonctionnalités dans sa version propre, tout en proposant souvent leur inclusion dans la version communautaire — ce qui n'est pas automatique.

Face au leadership de PostgreSQL, une tendance récente pour certaines bases de données est de se revendiquer « compatibles PostgreSQL ». Certains éditeurs de solutions de bases de données distribuées propriétaires disent que leur produit peut remplacer PostgreSQL sans modification de code côté application. Il convient de rester critique et prudent face à cette affirmation, car ces produits n'ont parfois rien à voir avec PosgreSQL.

[55] https://www.stitchdata.com/blog/how-redshift-differs-from-postgresql/

Cet historique provient en partie de la liste exhaustive des « forks »[56], ainsi de que cette conférence de Josh Berkus[57] de 2009 et des références en bibliographie.

Sauf cas très précis, il est recommandé d'utiliser la version officielle, libre et gratuite. Vous savez exactement ce qu'elle propose et vous choisissez librement vos partenaires (pour les formations, pour le support, pour les audits, etc).

1.4 QUELQUES PROJETS SATELLITES

PostgreSQL n'est que le moteur ! Besoin d'outils pour :

- Administration
- Sauvegarde
- Supervision
- Migration
- SIG

PostgreSQL n'est qu'un moteur de bases de données. Quand vous l'installez, vous n'avez que ce moteur. Vous disposez de quelques outils en ligne de commande (détaillés dans nos modules « Outils graphiques et consoles » et « Tâches courantes ») mais aucun outil graphique n'est fourni.

Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

Par choix, nous ne présenterons ici que des logiciels libres et gratuits. Pour chaque problématique, il existe aussi des solutions propriétaires. Ces solutions peuvent parfois apporter des fonctionnalités inédites. Il faut néanmoins considérer que l'offre de la communauté Open-Source répond à la plupart des besoins des utilisateurs de PostgreSQL.

1.4.1 Administration, Développement, Modélisation

Entre autres, dédiés ou pas :

- Administration :
  - pgAdmin4
  - temBoard
- Développement :
  - DBeaver
- Modélisation :
  - pgModeler

Il existe différents outils graphiques pour l'administration, le développement et la modélisation. Une

liste plus exhaustive est disponible sur le wiki PostgreSQL[58].

pgAdmin4[59] est un outil d'administration dédié à PostgreSQL, qui permet aussi de requêter. (La version 3 est considérée comme périmée.)

temBoard[60] est une console d'administration plus complète. temBoard intègre de la supervision, des tableaux de bord, la gestion des sessions en temps réel, du bloat, de la configuration et l'analyse des performances.

DBeaver[61] est un outil de requêtage courant, utilisable avec de nombreuses bases de données différentes, et adapté à PostgreSQL.

Pour la modélisation, pgModeler[62] est dédié à PostgreSQL. Il permet la modélisation, la rétro-ingénierie d'un schéma existant, la génération de scripts de migration.

1.4.2 Sauvegardes

- Export logique :
  - pg_back[63]
- Sauvegarde physique (PITR) :
  - pgBackRest[64], barman[65]

Les outils listés ci-dessus sont les outils principaux et que nous recommandons pour la réalisation des sauvegardes et la gestion de leur rétention.

Ils se basent sur les outils standards de PostgreSQL de sauvegarde physique ou logique.

[58] https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools
[59] https://www.pgadmin.org/
[60] https://labs.dalibo.com/temboard
[61] https://dbeaver.io/
[62] https://pgmodeler.io/

1.4.3 Supervision

- Nagios/Icinga2 :
  - check_pgactivity
  - check_postgres
- Prometheus : postgres_exporter
- PoWA

Pour ne citer que quelques projets libres et matures :

check_pgactivity[66] est une sonde Nagios pouvant récupérer un grand nombre de statistiques d'activités renseignées par PostgreSQL. Il faut de ce fait un serveur Nagios (ou un de ses nombreux forks ou surcharges) pour gérer les alertes et les graphes. Il existe aussi check_postgres[67].

postgres_exporter[68] est l'exporteur de métriques pour Prometheus.

PoWA[69] est composé d'une extension qui historise les statistiques récupérées par l'extension `pg_stat_statements` et d'une application web qui permet de récupérer les requêtes et leur statistiques facilement.

1.4.4 Audit

- pgBadger
- pgCluu

pgBadger[70] est l'outil de base pour les analyses (à posteriori) des traces de PostgreSQL, dont notamment les requêtes.

pgCluu[71] permet une analyse du système et de PostgreSQL.

[66] https://github.com/OPMDG/check_pgactivity
[67] https://bucardo.org/check_postgres/
[68] https://github.com/prometheus-community/postgres_exporter
[69] https://powa.readthedocs.io/en/latest/
[70] https://pgbadger.darold.net/
[71] https://pgcluu.darold.net/

1.4.5 Migration

- Oracle, MySQL : ora2pg
- MySQL, SQL Server : pgloader

Il existe de nombreux outils pour migrer vers PostgreSQL une base de données utilisant un autre moteur. Ce qui pose le plus problème en pratique est le code applicatif (procédures stockées).

Plusieurs outils libres ou propriétaires, plus ou moins efficaces, existent - ou ont existé. Citons les plus importants :

Ora2Pg[72], de Gilles Darold, convertit le schéma de données, migre les données, et tente même de convertir le code PL/SQL en PL/pgSQL. Il convertit aussi des bases MySQL.

pgloader[73], de Dimitri Fontaine, permet de migrer depuis MySQL, SQLite ou MS SQL Server, et importe les fichiers CSV, DBF (dBase) ou IXF (fichiers d'échange indépendants de la base).

Ces outils sont libres. Des sociétés vivant de la prestation de service autour de la migration ont également souvent développé les leurs.

1.4.6 PostGIS

[72] http://ora2pg.darold.net/
[73] https://pgloader.io/

- Projet indépendant, GPL, https://postgis.net/
- Module spatial pour PostgreSQL
  - Extension pour types géométriques/géographiques & outils
  - La référence des bases de données spatiales
  - « quelles sont les routes qui coupent le Rhône ? »
  - « quelles sont les villes adjacentes à Toulouse ? »
  - « quels sont les restaurants situés à moins de 3 km de la Nationale 12 ? »

PostGIS ajoute le support d'objets géographiques à PostgreSQL. C'est un projet totalement indépendant développé par la société Refractions Research sous licence GPL, soutenu par une communauté active, utilisée par des spécialistes du domaine géospatial (IGN, BRGM, AirBNB, Mappy, Openstreetmap, Agence de l'eau…), mais qui peut convenir pour des projets plus modestes.

Techniquement, c'est une extension transformant PostgreSQL en serveur de données spatiales, qui sera utilisé par un Système d'Information Géographique (SIG), tout comme le SDE de la société ESRI ou bien l'extension Oracle Spatial. PostGIS se conforme aux directives du consortium OpenGIS et a été certifié par cet organisme comme tel, ce qui est la garantie du respect des standards par PostGIS.

PostGIS permet d'écrire des requêtes de ce type :

```
SELECT restaurants.geom, restaurants.name FROM restaurants
  WHERE EXISTS (SELECT 1 FROM routes
                  WHERE ST_DWithin(restaurants.geom, routes.geom, 3000)
                AND route.name = 'Nationale 12')
```

PostGIS fournit les fonctions d'indexation qui permettent d'accéder rapidement aux objets géométriques, au moyen d'index GiST. La requête ci-dessus n'a évidemment pas besoin de parcourir tous les restaurants à la recherche de ceux correspondant aux critères de recherche.

La liste des fonctionnalités comprend le support des coordonnées géodésiques ; des projections et reprojections dans divers systèmes de coordonnées locaux (Lambert93 en France par exemple) ; des opérateurs d'analyse géométrique (enveloppe convexe, simplification…)

PostGIS est intégré aux principaux serveurs de carte, ETL, et outils de manipulation.

La version 3.0 apporte la gestion du parallélisme, un meilleur support de l'indexation SP-GiST et GiST, ainsi qu'un meilleur support du type GeoJSON.

1.5 SPONSORS & RÉFÉRENCES

- Sponsors[74]
- Références :
  - françaises
  - et internationales

Au-delà de ses qualités, PostgreSQL suscite toujours les mêmes questions récurrentes :

- qui finance les développements ? (et pourquoi ?)
- qui utilise PostgreSQL ?

1.5.1 Sponsors principaux

- Sociétés se consacrant à PostgreSQL :
  - Crunchy Data (USA) : Tom Lane, Stephen Frost, Joe Conway…
  - EnterpriseDB (USA) : Bruce Momjian, Robert Haas, Dave Page…
  - 2nd Quadrant (R.U.) : Simon Riggs, Peter Eisentraut…
    - racheté par EDB
  - PostgresPro (Russie) : Oleg Bartunov, Alexander Korotkov
  - Cybertec (Autriche), Dalibo (France), Redpill Linpro (Suède), Credativ (Allemagne)…
- Sociétés vendant un fork ou une extension :
  - Citusdata (Microsoft), Pivotal (VMWare), TimescaleDB

La liste des sponsors de PostgreSQL contribuant activement au développement figure sur la liste officielle des sponsors[75]. Ce qui suit n'est qu'un aperçu.

EnterpriseDB est une société américaine qui a décidé de fournir une version de PostgreSQL propriétaire fournissant une couche de compatibilité avec Oracle. Ils emploient plusieurs développeurs importants du projet PostgreSQL (dont trois font partie de la *Core Team*), et reversent un certain nombre

[75] https://www.postgresql.org/about/sponsors/

de leurs travaux au sein du moteur communautaire. Ils ont aussi un poids financier qui leur permet de sponsoriser la majorité des grands événements autour de PostgreSQL : PGEast et PGWest aux États-Unis, PGDay en Europe.

En 2020, EnterpriseDB rachète 2nd Quadrant, une société anglaise fondée par Simon Riggs, développeur PostgreSQL de longue date. 2nd Quadrant développe de nombreux outils autour de PostgreSQL comme pglogical, des versions dérivées comme Postgres-XL ou BDR, ou des outils annexes comme barman ou repmgr.

Crunchy Data offre sa propre version certifiée et finance de nombreux développements.

De nombreuses autres sociétés dédiées à PostgreSQL existent dans de nombreux pays. Parmi les sponsors officiels, nous pouvons compter Cybertec en Autriche ou Redpill Linpro en Suède. En Russie, PostgresPro maintient une version locale et reverse aussi de nombreuses contributions à la communauté.

En Europe francophone, Dalibo participe pleinement à la communauté. La société est Major Sponsor du projet PostgreSQL[76], ce qui indique un support de longue date. Elle développe et maintient plusieurs outils plébiscités par la communauté, comme autrefois Open PostgreSQL Monitoring (OPM) ou la sonde check_pgactivity[77], plus récemment la console d'administration temBoard[78], avec de nombreux autres projets en cours[79], et une participation active au développement de patchs pour PostgreSQL. Dalibo sponsorise également des événements comme les PGDay français et européens, ainsi que la communauté francophone.

Des sociétés comme Citusdata (racheté par Microsoft), Pivotal (VMWare) ou TimescaleDB proposent ou ont proposé leur version dérivée sous une forme ou une autre, mais « jouent le jeu » et participent au développement de la version communautaire, notamment en cherchant à ce que leur produit n'en diverge pas.

1.5.2 Autres sponsors

- Autres sociétés :
  - VMWare, Rackspace, Heroku, Conova, Red Hat, Microsoft
  - NTT (*streaming replication*), Fujitsu, NEC
- Cloud
  - nombreuses

[76] https://www.postgresql.org/about/sponsors/
[77] https://github.com/OPMDG/check_pgactivity
[78] https://labs.dalibo.com/temboard
[79] https://labs.dalibo.com/about

Contribuent également à PostgreSQL nombre de sociétés non centrées autour des bases de données.

NTT a financé de nombreux patchs pour PostgreSQL.

Fujitsu a participé à de nombreux développements aux débuts de PostgreSQL, et emploie Amit Kapila.

VMWare a longtemps employé le développeur finlandais Heikki Linnakangas, parti ensuite un temps chez Pivotal. VMWare emploie aussi Michael Paquier ou Julien Rouhaud.

Red Hat a longtemps employé Tom Lane à plein temps pour travailler sur PostgreSQL. Il a pu dédier une très grande partie de son temps de travail à ce projet, bien qu'il ait eu d'autres affectations au sein de Red Hat. Tom Lane a travaillé également chez SalesForce, ensuite il a rejoint Crunchy Data Solutions fin 2015.

Il y a déjà plus longtemps, Skype a offert un certain nombre d'outils très intéressants : pgBouncer (pooler de connexion), Londiste (réplication par trigger), etc. Ce sont des outils utilisés en interne et publiés sous licence BSD comme retour à la communauté. Malgré le rachat par Microsoft, certains sont encore utiles et maintenus.

Zalando est connu pour l'outil de haute disponibilité patroni.

De nombreuses sociétés liées au cloud figurent aussi parmi les sponsors, comme Conova (Autriche), Heroku ou Rackspace (États-Unis), ou les mastodontes Google, Amazon Web Services et, à nouveau, Microsoft.

1.5.3 Références

- Météo France
- IGN
- RATP, SNCF
- CNAF
- MAIF, MSA
- Le Bon Coin
- Air France-KLM
- Société Générale
- Carrefour, Leclerc, Leroy Merlin
- Instagram, Zalando, TripAdvisor
- Yandex
- CNES
- …et plein d'autres

Météo France utilise PostgreSQL depuis plus d'une décennie pour l'essentiel de ses bases, dont des

instances critiques de plusieurs téraoctets (témoignage sur postgresql.fr[80]).

L'IGN utilise PostGIS et PostgreSQL depuis 2006[81].

La RATP a fait ce choix depuis 2007 également[82].

La Caisse Nationale d'Allocations Familiales a remplacé ses mainframes par des instances PostgreSQL[83] dès 2010 (4 To et 1 milliard de requêtes par jour).

Instagram utilise PostgreSQL depuis le début[84].

Zalando a décrit plusieurs fois son infrastructure PostgreSQL[85] et annonçait en 2018[86] utiliser pas moins de 300 bases de données en interne et 650 instances dans un cloud AWS. Zalando contribue à la communauté, notamment par son outil de haute disponibilité patroni[87].

Le DBA de TripAdvisor témoigne de leur utilisation de PostgreSQL dans l'interview suivante[88].

Dès 2009, Leroy Merlin migrait vers PostgreSQL des milliers de logiciels de caisse[89].

Yandex, équivalent russe de Google a décrit en 2016 la migration des 300 To de données de Yandex.Mail depuis Oracle vers PostgreSQL[90].

La Société Générale a publié son outil de migration d'Oracle à PostgreSQL[91].

Autolib à Paris utilisait PostgreSQL. Le logiciel est encore utilisé dans les autres villes où le service continue. Ils ont décrit leur infrastructure au PG Day 2018 à Marseille[92].

De nombreuses autres sociétés participent au Groupe de Travail Inter-Entreprises de PostgreSQLFr[93] : Air France, Carrefour, Leclerc, le CNES, la MSA, la MAIF, PeopleDoc, EDF…

Cette liste ne comprend pas les innombrables sociétés qui n'ont pas communiqué sur le sujet. PostgreSQL étant un logiciel libre, il n'existe nulle part de dénombrement des instances actives.

[80] https://www.postgresql.fr/temoignages/meteo_france
[81] https://www.postgresql.fr/temoignages/ign
[82] https://www.journaldunet.com/solutions/dsi/1013631-la-ratp-integre-postgresql-a-son-systeme-d-information/
[83] https://www.silicon.fr/cnaf-debarrasse-mainframes-149897.html?inf_by=5bc488a1671db858728b4c35
[84] https://media.postgresql.org/sfpug/instagram_sfpug.pdf
[85] http://gotocon.com/dl/goto-berlin-2013/slides/HenningJacobs_and_ValentineGogichashvili_WhyZalandoTrustsInPostgreSQL.pdf
[86] https://www.postgresql.eu/events/pgconfeu2018/schedule/session/2135-highway-to-hell-or-stairway-to-cloud/
[87] https://jobs.zalando.com/tech/blog/zalandos-patroni-a-template-for-high-availability-postgresql/
[88] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley
[89] https://wiki.postgresql.org/images/6/63/Adeo_PGDay.pdf
[90] https://www.pgcon.org/2016/schedule/attachments/426_2016.05.19%20Yandex.Mail%20success%20story.pdf
[91] https://github.com/societe-generale/code2pg
[92] https://www.youtube.com/watch?v=vd8B7B-Zca8
[93] https://www.postgresql.fr/entreprises/accueil

1.5.4 Le Bon Coin

- Site de petites annonces
- 4è site le plus consulté en France (2017)
- 27 millions d'annonces en ligne, 800 000 nouvelles chaque jour
- Instance PostgreSQL principale : 3 To de volume, 3 To de RAM
- 20 serveurs secondaires

PostgreSQL tient la charge sur de grosses bases de données et des serveurs de grande taille.

Le Bon Coin privilégie des serveurs physiques dans ses propres datacenters.

Pour plus de détails et l'évolution de la configuration, voir les témoignages de ses directeurs technique[94] (témoignage de juin 2012) et infrastructure[95] (juin 2017), ou la conférence de son DBA Flavio Gurgel au pgDay Paris 2019[96].

Ce dernier s'appuie sur les outils classiques fournis par la communauté : pg_dump (pour archivage, car ses exports peuvent être facilement restaurés), barman, pg_upgrade.

[94] https://www.postgresql.fr/temoignages:le_bon_coin
[95] https://web.archive.org/web/20171222173630/https://www.kissmyfrogs.com/jean-louis-bergamo-leboncoin-ce-qui-a-ete-fait-maison-est-ultra-performant/
[96] https://www.postgresql.eu/events/pgdayparis2019/schedule/session/2376-large-databases-lots-of-servers

1.6 À LA RENCONTRE DE LA COMMUNAUTÉ

- Cartographie du projet
- Pourquoi participer
- Comment participer

1.6.1 PostgreSQL, un projet mondial

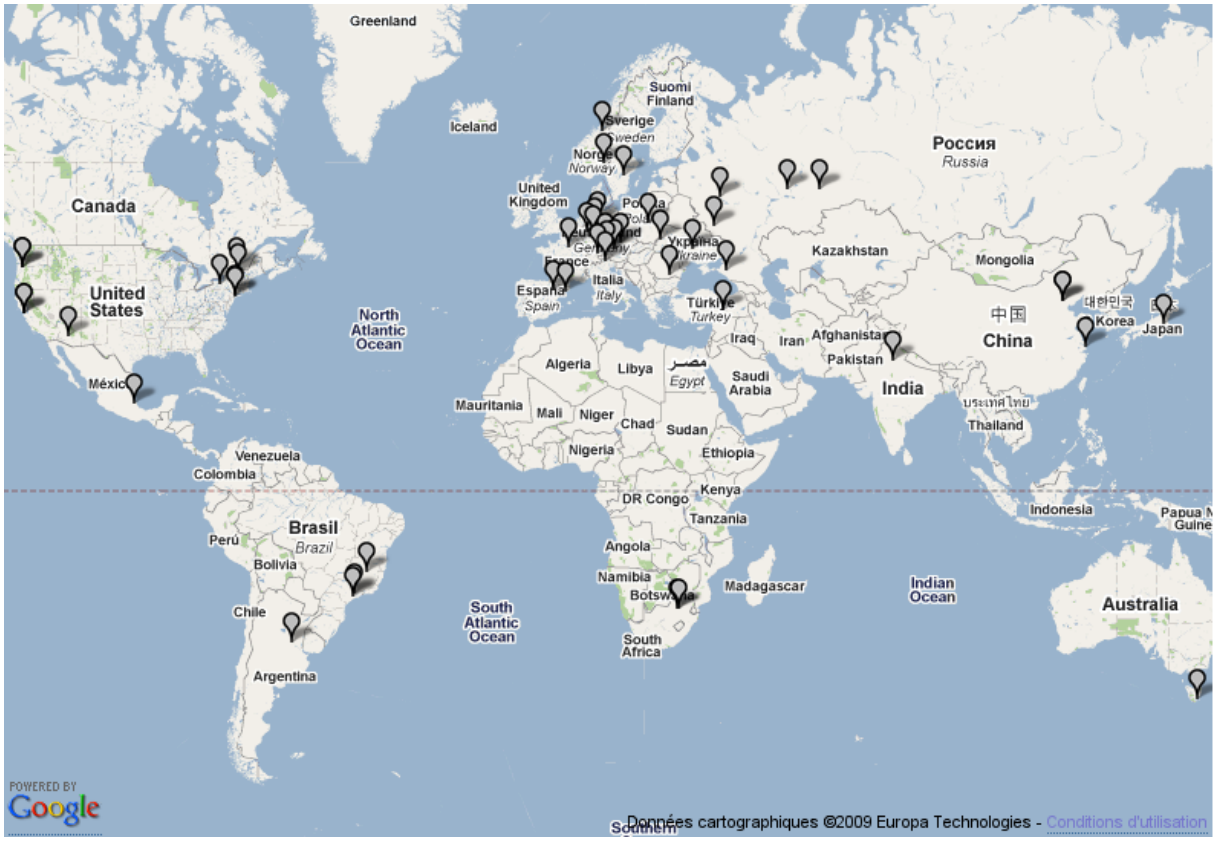


Figure 1/ .2: Carte des hackers

On le voit, PostgreSQL compte des contributeurs sur tous les continents.

Le projet est principalement anglophone. Les *core hackers* sont surtout répartis en Amérique, Europe, Asie (Japon surtout).

Il existe une très grande communauté au Japon, et de nombreux développeurs en Russie.

La communauté francophone est très dynamique, s'occupe beaucoup des outils, mais il n'y a que quelques développeurs réguliers du *core* francophones : Michael Paquier, Julien Rouhaud, Fabien Coelho…

La communauté hispanophone est naissante.

1.6.2 PostgreSQL Core Team

Figure 1/ .3: Core team

Le terme *Core Hackers* désigne les personnes qui sont dans la communauté depuis longtemps. Ces personnes désignent directement les nouveaux membres.

Le terme *hacker* peut porter à confusion, il s'agit ici de la définition « universitaire » : https://fr.wikipedia.org/wiki/Hacker_(programmation)

La *Core Team* est un ensemble de personnes doté d'un pouvoir assez limité. Ils ne doivent pas appartenir en majorité à la même société. Ils peuvent décider de la date de sortie d'une version. Ce sont les personnes qui sont immédiatement au courant des failles de sécurité du serveur PostgreSQL. Exceptionnellement, elles tranchent certains débats si un consensus ne peut être atteint dans la communauté. Tout le reste des décisions est pris par la communauté dans son ensemble après discussion, généralement sur la liste pgsql-hackers.

Les membres actuels de la *Core Team* sont[97] :

- **Tom Lane** (Crunchy Data, Pittsburgh, États-Unis) : certainement le développeur le plus aguerri avec la vision la plus globale, notamment sur l'optimiseur ;

[97] https://www.postgresql.org/community/contributors/

- **Bruce Momjian** (EnterpriseDB, Philadelphie, États-Unis) : a lancé le projet en 1995, écrit du code (pg_upgrade notamment) et s'est beaucoup occupé de la promotion ;
- **Magnus Hagander** (Redpill Linpro, Stockholm, Suède) : développeur, a participé notamment au portage Windows, à l'outil pg_basebackup, à l'administration des serveurs, président de PostgreSQL Europe ;
- **Andres Freund** (Microsoft, San Francisco, États-Unis) : contributeur depuis des années de nombreuses fonctionnalités (JIT, réplication logique, performances…) ;
- **Dave Page** (EnterpriseDB, Oxfordshire, Royaume-Uni) : leader du projet pgAdmin, version Windows, administration des serveurs, secrétaire de PostgreSQL Europe ;
- **Peter Eisentraut** (EnterpriseDB, Dresde, Allemagne) : développement du moteur (internationalisation, SQL/Med…), respect de la norme SQL, etc. ;
- **Jonathan Katz** (Crunchy Data, New York, États-Unis) : promotion du projet, modération, revues de patchs.

1.6.3 Contributeurs

Figure 1/ .4: Contributeurs

Actuellement, PostgreSQL compte une centaine de « contributeurs » qui se répartissent quotidiennement les tâches suivantes :

- développement des projets satellites (Slony, pgAdmin…) ;
- promotion du logiciel ;
- administration de l'infrastructure des serveurs ;

- rédaction de documentation ;
- conférences ;
- traductions ;
- organisation de groupes locaux.

Le *PGDG* a fêté son 10e anniversaire à Toronto en juillet 2006. Ce « PostgreSQL Anniversary Summit » a réuni pas moins de 80 membres actifs du projet. La photo ci-dessus a été prise à l'occasion.

PGCon2009 a réuni 180 membres actifs à Ottawa, et environ 220 en 2018 et 2019.

Voir la liste des contributeurs officiels[98].

1.6.4 Qui contribue du code ?

- Principalement des personnes payées par leur société
- 28 committers[99]
- En 2019, en code :
  - Tom Lane
  - Andres Freund
  - Peter Eisentraut
  - Nikita Glukhov
  - Álvaro Herrera
  - Michael Paquier
  - Robert Haas
  - …et beaucoup d'autres
- Commitfests[100] : tous les 2 mois

À l'automne 2021, on compte 28 *committers*, c'est-à-dire personnes pouvant écrire dans tout ou partie du dépôt de PostgreSQL. Il ne s'agit pas que de leur travail, mais pour une bonne partie de patchs d'autres contributeurs après discussion et validation des fonctionnalités mais aussi des standards propres à PostgreSQL, de la documentation, de la portabilité, de la simplicité, de la sécurité, etc. Ces autres contributeurs peuvent être potentiellement n'importe qui. En général, un patch est relu par plusieurs personnes avant d'être transmis à un *committer*.

Les discussions quant au développement ont lieu principalement (mais pas uniquement) sur la liste pgsql-hackers[101]. Les éventuels bugs sont transmis à la liste pgsql-bugs[102]. Puis les patchs en cours sont revus au moins tous les deux mois lors des Commitfests. Il n'y a pas de *bug tracker* car le fonctionnement actuel est jugé satisfaisant.

[98] https://www.postgresql.org/community/contributors/
[101] https://www.postgresql.org/list/pgsql-hackers/
[102] https://www.postgresql.org/list/pgsql-bugs/

Robert Haas publie chaque année une analyse sur les contributeurs de code et les participants aux discussions sur le développement de PostgreSQL sur la liste pgsql-hackers :

- 2020/2021 : http://rhaas.blogspot.com/2022/01/who-contributed-to-postgresql.html
- 2019 : http://rhaas.blogspot.com/2020/05/who-contributed-to-postgresql.html
- 2018 : http://rhaas.blogspot.com/2019/01/who-contributed-to-postgresql.html
- 2017 : http://rhaas.blogspot.com/2018/06/who-contributed-to-postgresql.html
- 2016 : http://rhaas.blogspot.com/2017/04/who-contributes-to-postgresql.html

1.6.5 Répartition des développeurs

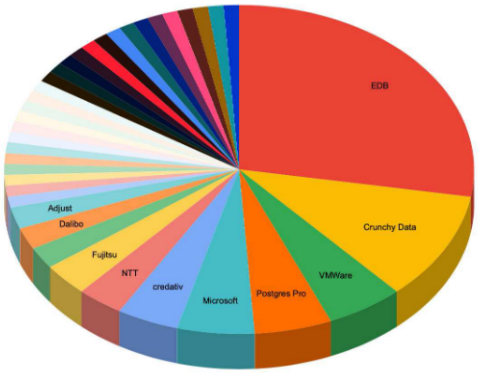


Figure 1/ .5: Répartition des développeurs

Voici une répartition des différentes sociétés qui ont contribué aux améliorations de la version 13. On y voit qu'un grand nombre de sociétés prend part à ce développement. La plus importante est EDB, mais même elle n'est responsable que d'un petit tiers des contributions.

(Source : Future Postgres Challenges[103], Bruce Momjian, 2021)

1.6.6 Utilisateurs

- Vous !
- **Le succès d'un logiciel libre dépend de ses utilisateurs.**

Il est impossible de connaître précisément le nombre d'utilisateurs de PostgreSQL. Toutefois ce nombre est en constante augmentation.

[103] https://momjian.us/main/writings/pgsql/challenges.pdf

Il existe différentes manières de s'impliquer dans une communauté Open-Source. Dans le cas de PostgreSQL, vous pouvez :

- déclarer un bug ;
- tester les versions bêta ;
- témoigner.

1.6.7 Pourquoi participer

- Rapidité des corrections de bugs
- Préparer les migrations / tester les nouvelles versions
- Augmenter la visibilité du projet
- Créer un réseau d'entraide

Au-delà de motivations idéologiques ou technologiques, il y a de nombreuses raisons objectives de participer au projet PostgreSQL.

Envoyer une description d'un problème applicatif aux développeurs est évidemment le meilleur moyen d'obtenir sa correction. Attention toutefois à être précis et complet lorsque vous déclarez un bug sur pgsql-bugs[104] ! Assurez-vous que vous pouvez le reproduire.

Tester les versions « candidates » dans votre environnement (matériel et applicatif) est la meilleure garantie que votre système d'information sera compatible avec les futures versions du logiciel.

Les retours d'expérience et les cas d'utilisations professionnelles sont autant de preuves de la qualité de PostgreSQL. Ces témoignages aident de nouveaux utilisateurs à opter pour PostgreSQL, ce qui renforce la communauté.

S'impliquer dans les efforts de traductions, de relecture ou dans les forums d'entraide ainsi que toute forme de transmission en général est un très bon moyen de vérifier et d'approfondir ses compétences.

1.6.8 Ressources web de la communauté

- Site officiel : https://www.postgresql.org/
- Actualité : https://planet.postgresql.org/
- Des extensions : https://pgxn.org/

[104] https://www.postgresql.org/list/pgsql-bugs/

Le site officiel de la communauté se trouve sur https://www.postgresql.org/. Ce site contient des informations sur PostgreSQL, la documentation des versions maintenues, les archives des listes de discussion, etc.

Le site « Planet PostgreSQL » est un agrégateur réunissant les blogs des *Core Hackers*, des contributeurs, des traducteurs et des utilisateurs de PostgreSQL.

Le site PGXN est l'équivalent pour PostgreSQL du CPAN de Perl, une collection en ligne de librairies et extensions accessibles depuis la ligne de commande.

1.6.9 Documentation officielle

- LA référence, même au quotidien
- Anglais : https://www.postgresql.org/docs/
- Français : https://docs.postgresql.fr/

La documentation officielle sur https://www.postgresql.org/docs/current est maintenue au même titre que le code du projet, et sert aussi au quotidien, pas uniquement pour des cas obscurs.

Elle est versionnée pour chaque version majeure.

La traduction française suit de près les mises à jour de la documentation officielle : https://docs.postgresql.fr/.

1.6.10 Serveurs francophones

- Site officiel : https://www.postgresql.fr/
- Documentation traduite : https://docs.postgresql.fr/
- Forum : https://forums.postgresql.fr/
- Actualité : https://planete.postgresql.fr/
- Association PostgreSQLFr : https://www.postgresql.fr/asso/accueil
- Groupe de Travail Inter-Entreprises (PGGTIE) : https://www.postgresql.fr/entreprises/accueil

Le site postgresql.fr est le site de l'association des utilisateurs francophones du logiciel. La communauté francophone se charge de la traduction de toutes les documentations.

1.6.11 Listes de discussions / Listes d'annonces

- pgsql-announce
- pgsql-general
- pgsql-admin
- pgsql-sql
- pgsql-performance
- pgsql-fr-generale
- pgsql-advocacy
- pgsql-bugs

Les mailing-lists sont les outils principaux de gouvernance du projet. Toute l'activité de la communauté (bugs, promotion, entraide, décisions) est accessible par ce canal. Les développeurs principaux du projets répondent parfois eux-mêmes. Si vous avez une question ou un problème, la réponse se trouve probablement dans les archives !

Pour s'inscrire ou consulter les archives : https://www.postgresql.org/list/.

Si vous pensez avoir trouvé un bug, vous pouvez le remonter sur la liste anglophone pgsql-bugs[105], par le formulaire dédié[106]. Pour faciliter la tâche de ceux qui tenteront de vous répondre, suivez bien les consignes sur les rapports de bug[107] : informations complètes, reproductibilité…

1.6.12 IRC

- Réseau LiberaChat
- IRC anglophone :
  - #postgresql
  - #postgresql-eu
- IRC francophone :
  - #postgresqlfr

Le point d'entrée principal pour le réseau LiberaChat est le serveur **irc.libera.chat**. La majorité des développeurs sont disponibles sur IRC et peuvent répondre à vos questions.

Des canaux de discussion spécifiques à certains projets connexes sont également disponibles, comme par exemple #slony.

Attention ! Vous devez poser votre question en public et ne pas solliciter de l'aide par message privé.

1.6.13 Wiki

- https://wiki.postgresql.org/

Le wiki est un outil de la communauté qui met à disposition une véritable mine d'informations.

Au départ, le wiki avait pour but de récupérer les spécifications écrites par des développeurs pour les grosses fonctionnalités à développer à plusieurs. Cependant, peu de développeurs l'utilisent dans ce cadre. L'utilisation du wiki a changé en passant plus entre les mains des utilisateurs qui y intègrent un bon nombre de pages de documentation (parfois reprises dans la documentation officielle). Le wiki est aussi utilisé par les organisateurs d'événements pour y déposer les slides des conférences. Elle n'est pas exhaustive et, hélas, souffre fréquemment d'un manque de mises à jour.

1.6.14 L'avenir de PostgreSQL

- PostgreSQL est devenu la base de données de référence
- Grandes orientations :
  - réplication logique
  - meilleur parallélisme
  - gros volumes
- Prochaine version, la 17
- Stabilité économique
- De plus en plus de (gros) clients
- Le futur de PostgreSQL dépend de vous !

Le projet avance grâce à de plus en plus de contributions. Les grandes orientations actuelles sont :

- une réplication de plus en plus sophistiquée ;
- une gestion plus étendue du parallélisme ;
- une volumétrie acceptée de plus en plus importante ;
- etc.

PostgreSQL est là pour durer. Le nombre d'utilisateurs, de toutes tailles, augmente tous les jours. Il n'y a pas qu'une seule entreprise derrière ce projet. Il y en a plusieurs, petites et grosses sociétés, qui s'impliquent pour faire avancer le projet, avec des modèles économiques et des marchés différents, garants de la pérennité du projet.

1.7 CONCLUSION

- Un projet de grande ampleur
- Un SGBD complet
- Souplesse, extensibilité
- De belles références
- Une solution **stable**, **ouverte**, **performante** et **éprouvée**
- Pas de dépendance envers UN éditeur

Certes, la licence PostgreSQL implique un coût nul (pour l'acquisition de la licence), un code source disponible et aucune contrainte de redistribution. Toutefois, il serait erroné de réduire le succès de PostgreSQL à sa gratuité.

Beaucoup d'acteurs font le choix de leur SGBD sans se soucier de son prix. En l'occurrence, ce sont souvent les qualités intrinsèques de PostgreSQL qui séduisent :

- sécurité des données (reprise en cas de crash et résistance aux bogues applicatifs) ;
- facilité de configuration ;
- montée en puissance et en charge progressive ;
- gestion des gros volumes de données ;
- pas de dépendance envers un unique éditeur ou prestataire.

1.7.1 Bibliographie

- Documentation officielle (préface)
- Articles fondateurs de M. Stonebracker (1987)
- *Présentation du projet PostgreSQL* (Guillaume Lelarge, 2008)
- *Looking back at PostgreSQL* (J.M. Hellerstein, 2019)

Quelques références :

- Préface de la documentation officielle : 2. Bref historique de PostgreSQL[108]
- *The Design of POSTGRES*[109] , Michael Stonebraker & Lawrence A. Rowe, 1987
- Présentation du projet PostgreSQL[110], Guillaume Lelarge, RMLL 2008

[108] https://docs.postgresql.fr/current/history.html
[109] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[110] https://web.archive.org/web/20160322070704/2008.rmll.info/Presentation-de-PostgreSQL.html

- *Looking Back at PostgreSQL*[111] , Joseph M. Hellerstein, 2019

Iconographie : La photo initiale est le logo officiel de PostgreSQL[112].

1.7.2 Questions

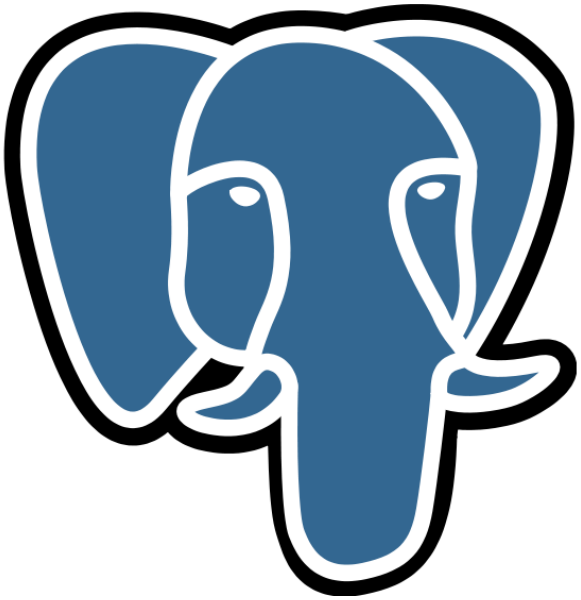
N'hésitez pas, c'est le moment !

[111] https://arxiv.org/pdf/1901.01973.pdf
[112] https://www.postgresql.org/about/policies/trademarks/

1.8 QUIZ

https://dali.bo/a1_quiz

2/ Découverte des fonctionnalités



2.1 AU MENU

- Fonctionnalités du moteur
- Objets SQL
- Connaître les différentes fonctionnalités et possibilités
- Découvrir des exemples concrets

Ce module propose un tour rapide des fonctionnalités principales du moteur : ACID, MVCC, transactions, journaux de transactions… ainsi que des objets SQL gérés (schémas, index, tablespaces, triggers…). Ce rappel des concepts de base permet d'avancer plus facilement lors des modules suivants.

2.2 FONCTIONNALITÉS DU MOTEUR

- Standard SQL
- ACID : la gestion transactionnelle
- Niveaux d'isolation
- Journaux de transactions
- Administration
- Sauvegardes
- Réplication
- Supervision
- Sécurité
- Extensibilité

Cette partie couvre les différentes fonctionnalités d'un moteur de bases de données. Il ne s'agit pas d'aller dans le détail de chacune, mais de donner une idée de ce qui est disponible. Les modules suivants de cette formation et des autres formations détaillent certaines de ces fonctionnalités.

2.2.1 Respect du standard SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, routines, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.

La dernière version du standard SQL est SQL:2023[1]. À ce jour, aucun SGBD ne la supporte complètement, *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de la norme est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

[1] https://en.wikipedia.org/wiki/SQL:2023

2.2.2 ACID

Gestion transactionnelle : la force des bases de données relationnelles :

- **Atomicité** (*Atomic*)
- **Cohérence** (*Consistent*)
- **Isolation** (*Isolated*)
- **Durabilité** (*Durable*)

Les propriétés ACID sont le fondement même de toute bonne base de données. Il s'agit de l'acronyme des quatre règles que toute transaction (c'est-à-dire une suite d'ordres modifiant les données) doit respecter :

- **A** : Une transaction est appliquée en « tout ou rien ».
- **C** : Une transaction amène la base d'un état stable à un autre.
- **I** : Les transactions n'agissent pas les unes sur les autres.
- **D** : Une transaction validée sera conservée de manière permanente.

Les bases de données relationnelles les plus courantes depuis des décennies (PostgreSQL bien sûr, mais aussi Oracle, MySQL, SQL Server, SQLite…) se basent sur ces principes, même si elles font chacune des compromis différents suivant leurs cas d'usage, les compromis acceptés à chaque époque avec la performance et les versions.

Atomicité :

Une transaction doit être exécutée entièrement ou pas du tout, et surtout pas partiellement, même si elle est longue et complexe, même en cas d'incident majeur sur la base de données. L'exemple basique est une transaction bancaire : le montant d'un virement doit être sur un compte ou un autre, et en cas de problème ne pas disparaître ou apparaître en double. Ce principe garantit que les données modifiées par des transactions valides seront toujours visibles dans un état stable, et évite nombre de problèmes fonctionnels comme techniques.

Cohérence :

Un état cohérent respecte les règles de validité définies dans le modèle, c'est-à-dire les contraintes définies dans le modèle : types, plages de valeurs admissibles, unicité, liens entre tables (clés étrangères), etc. Le non-respect de ces règles par l'applicatif entraîne une erreur et un rejet de la transaction.

Isolation :

Des transactions simultanées doivent agir comme si elles étaient seules sur la base. Surtout, elles ne voient pas les données *non validées* des autres transactions. Ainsi une transaction peut travailler sur un état stable et fixe, et durer assez longtemps sans risque de gêner les autres transactions.

Il existe plusieurs « niveaux d'isolation » pour définir précisément le comportement en cas de lectures ou écritures simultanées sur les mêmes données et pour arbitrer avec les contraintes de per-

formances ; le niveau le plus contraignant exige que tout se passe comme si toutes les transactions se déroulaient successivement.

Durabilité :

Une fois une transaction validée par le serveur (typiquement : `COMMIT` ne retourne pas d'erreur, ce qui valide la cohérence et l'enregistrement physique), l'utilisateur doit avoir la garantie que la donnée ne sera pas perdue ; du moins jusqu'à ce qu'il décide de la modifier à nouveau. Cette garantie doit valoir même en cas d'événement catastrophique : plantage de la base, perte d'un disque… C'est donc au serveur de s'assurer autant que possible que les différents éléments (disque, système d'exploitation…) ont bien rempli leur office. C'est à l'humain d'arbitrer entre le niveau de criticité requis et les contraintes de performances et de ressources adéquates (et fiables) à fournir à la base de données.

NoSQL :

À l'inverse, les outils de la mouvance (« NoSQL », par exemple MongoDB ou Cassandra), ne fournissent pas les garanties ACID. C'est le cas de la plupart des bases non-relationnelles, qui reprennent le modèle BASE[2] (*Basically Available*, *Soft State*, *Eventually Consistent*, soit succintement : disponibilité d'abord ; incohérence possible entre les réplicas ; cohérence… à terme, après un délai). Un intérêt est de débarasser le développeur de certaines lourdeurs apparentes liées à la modélisation assez stricte d'une base de données relationnelle. Cependant, la plupart des applications ont d'abord besoin des garanties de sécurité et cohérence qu'offrent un moteur transactionnel classique, et la décision d'utiliser un système ne les garantissant pas ne doit pas être prise à la légère ; sans parler d'autres critères comme la fragmentation du domaine par rapport au monde relationnel et son SQL (à peu près) standardisé. Avec le temps, les moteurs transactionnels ont acquis des fonctionnalités qui faisaient l'intérêt des bases NoSQL (en premier lieu la facilité de réplication et le stockage de JSON), et ces dernières ont tenté d'intégrer un peu plus de sécurité dans leur modèle.

2.2.3 MVCC

- MultiVersion Concurrency Control
- Le « noyau » de PostgreSQL
- Garantit les propriétés ACID
- Permet les accès concurrents sur la même table
  - une lecture ne bloque pas une écriture
  - une écriture ne bloque pas une lecture
  - une écriture ne bloque pas les autres écritures…
  - …sauf pour la mise à jour de la **même ligne**

[2] https://en.wikipedia.org/wiki/Eventual_consistency

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

MVCC maintient toutes les versions nécessaires de chaque ligne, ainsi **chaque transaction voit une image figée de la base** (appelée *snapshot*). Cette image correspond à l'état de la base lors du démarrage de la requête ou de la transaction, suivant le niveau d'*isolation* demandé par l'utilisateur à PostgreSQL pour la transaction.

MVCC fluidifie les mises à jour en évitant les blocages trop contraignants (verrous sur `UPDATE`) entre sessions et par conséquent de meilleures performances en contexte transactionnel.

C'est notamment MVCC qui permet d'exporter facilement une base *à chaud* et d'obtenir un export cohérent alors même que plusieurs utilisateurs sont potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs, les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, moins de contention et un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

2.2.4 Transactions

- Une transaction = ensemble **atomique** d'opérations
- « Tout ou rien »
- `BEGIN` obligatoire pour grouper des modifications
- `COMMIT` pour valider
  - y compris le DDL
- Perte des modifications si :
  - `ROLLBACK` / perte de la connexion / arrêt (brutal ou non) du serveur
- `SAVEPOINT` pour sauvegarde des modifications d'une transaction à un instant `t`
- Pas de transactions imbriquées

L'exemple habituel et très connu des transactions est celui du virement d'une somme d'argent du compte de Bob vers le compte d'Alice. Le total du compte de Bob ne doit pas montrer qu'il a été débité de X euros tant que le compte d'Alice n'a pas été crédité de X euros. Nous souhaitons en fait que les deux opérations apparaissent aux yeux du reste du système comme une seule opération unitaire. D'où l'emploi d'une transaction explicite. En voici un exemple :

```
BEGIN;
UPDATE comptes SET solde=solde-200 WHERE proprietaire='Bob';
UPDATE comptes SET solde=solde+200 WHERE proprietaire='Alice';
COMMIT;
```

Contrairement à d'autres moteurs de bases de données, PostgreSQL accepte aussi les instructions DDL dans une transaction. En voici un exemple :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);

SELECT age FROM capitaines;

 age
  35

ROLLBACK;
SELECT age FROM capitaines;

ERROR:  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines;
                        ^
```

Nous voyons que la table `capitaines` a existé **à l'intérieur** de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final.

Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL : PostgreSQL n'effectue aucun `COMMIT` implicite sur des ordres DDL tels que `CREATE TABLE`, `DROP TABLE` ou `TRUNCATE TABLE`. De ce fait, ces ordres peuvent être annulés au sein d'une transaction.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);
SAVEPOINT insert_sp;
UPDATE capitaines SET age = 45 WHERE nom = 'Haddock';
ROLLBACK TO SAVEPOINT insert_sp;
COMMIT;

SELECT age FROM capitaines WHERE nom = 'Haddock';

 age
  35
```

Malgré le `COMMIT` après l'`UPDATE`, la mise à jour n'est pas prise en compte. En effet, le `ROLLBACK TO SAVEPOINT` a permis d'annuler cet `UPDATE` mais pas les opérations précédant le `SAVEPOINT`.

À partir de la version 12, il est possible de chaîner les transactions avec `COMMIT AND CHAIN` ou `ROLLBACK AND CHAIN`. Cela veut dire terminer une transaction et en démarrer une autre immédiatement après avec les mêmes propriétés (par exemple, le niveau d'isolation).

2.2.5 Niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Nous pouvons spécifier le niveau d'isolation au démarrage d'une transaction
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited` (défaut)
  - `repeatable read`
  - `serializable`

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

Le standard SQL spécifie quatre niveaux, mais PostgreSQL n'en supporte que trois (il n'y a pas de `read uncommitted` : les lignes non encore committées par les autres transactions sont toujours invisibles).

2.2.6 Fiabilité : journaux de transactions

- *Write Ahead Logs* (WAL)
- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Avantages :
  - WAL : écriture séquentielle
  - un seul *sync* sur le WAL
  - fichiers de données : en asynchrone
  - sauvegarde PITR et de la réplication fiables

Les journaux de transactions (appelés souvent WAL, autrefois XLOG) sont une garantie contre les pertes de données.

Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions. Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait en deux temps :

- écriture immédiate dans le journal de transactions ;
- écriture dans le fichier de données, plus tard, lors du prochain *checkpoint*.

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- PostgreSQL vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent.

Plus d'informations, lire cet article[3].

Les écritures dans le journal se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture (sur un disque dur classique, c'est l'opération la plus coûteuse).

De plus, comme nous n'écrivons que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, si le système de fichiers le supporte.

L'écriture définitive dans les fichiers de données, asynchrone et généralement de manière lissée, permet là aussi de gagner du temps.

Mais les performances ne sont pas la seule raison des journaux de transactions. Ces journaux ont aussi permis l'apparition de nouvelles fonctionnalités très intéressantes, comme le PITR et la réplication physique, basés sur le rejeu des informations stockées dans ces journaux.

2.2.7 Sauvegardes

- Sauvegarde des fichiers à froid
  - outils système
- Import/Export logique
  - `pg_dump`, `pg_dumpall`, `pg_restore`
- Sauvegarde physique à chaud
  - `pg_basebackup`
  - sauvegarde PITR

PostgreSQL supporte différentes solutions pour la sauvegarde.

La plus simple revient à sauvegarder à froid tous les fichiers des différents répertoires de données mais cela nécessite d'arrêter le serveur, ce qui occasionne une mise hors production plus ou moins longue, suivant la volumétrie à sauvegarder.

L'export logique se fait avec le serveur démarré. Plusieurs outils sont proposés : `pg_dump` pour sauvegarder une base, `pg_dumpall` pour sauvegarder toutes les bases. Suivant le format de l'export, l'import se fera avec les outils `psql` ou `pg_restore`. Les sauvegardes se font à chaud et sont cohérentes sans blocage de l'activité (seuls la suppression des tables et le changement de leur définition sont interdits).

Enfin, il est possible de sauvegarder les fichiers à chaud. Cela nécessite de mettre en place l'archivage des journaux de transactions. L'outil `pg_basebackup` est conseillé pour ce type de sauvegarde.

Il est à noter qu'il existe un grand nombre d'outils développés par la communauté pour faciliter encore plus la gestion des sauvegardes avec des fonctionnalités avancées comme le PITR (*Point In Time Recovery*) ou la gestion de la rétention, notamment pg_back (sauvegarde logique), pgBackRest ou barman (sauvegarde physique).

2.2.8 Réplication

- Réplication physique
  - instance complète
  - même architecture
- Réplication logique (PG 10+)
  - table par table / colonne par colonne avec ou sans filtre (PG 15)
  - voire opération par opération
- Asynchrones ou synchrone
- Asymétriques

PostgreSQL dispose de la réplication depuis de nombreuses années.

Le premier type de réplication intégrée est la réplication physique. Il n'y a pas de granularité, c'est forcément l'instance complète (toutes les bases de données), et au niveau des fichiers de données. Cette réplication est asymétrique : un seul serveur primaire effectue lectures comme écritures, et les serveurs secondaires n'acceptent que des lectures.

Le deuxième type de réplication est bien plus récent vu qu'il a été ajouté en version 10. Il s'agit d'une réplication logique, où les données elles-mêmes sont répliquées. Cette réplication est elle aussi asymétrique. Cependant, ceci se configure table par table (et non pas au niveau de l'instance comme pour la réplication physique). Avec la version 15, il devient possible de choisir quelles colonnes sont publiées et de filtre les lignes à publier.

La réplication logique n'est pas intéressante quand nous voulons un serveur sur lequel basculer en cas de problème sur le primaire. Dans ce cas, il vaut mieux utiliser la réplication physique. Par contre, c'est le bon type de réplication pour une réplication partielle ou pour une mise à jour de version majeure.

Dans les deux cas, les modifications sont transmises en asynchrone (avec un délai possible). Il est cependant possible de la configurer en synchrone pour tous les serveurs ou seulement certains.

2.2.9 Extensibilité

- Extensions
  - `CREATE EXTENSION monextension ;`
  - nombreuses : contrib, packagées… selon provenance
  - notion de confiance (v13+)
  - dont langages de procédures stockées !
- Système des *hooks*
- *Background workers*

Faute de pouvoir intégrer toutes les fonctionnalités demandées dans PostgreSQL, ses développeurs se sont attachés à permettre à l'utilisateur d'étendre lui-même les fonctionnalités sans avoir à modifier le code principal.

Ils ont donc ajouté la possibilité de créer des extensions. Une extension contient un ensemble de types de données, de fonctions, d'opérateurs, etc. en un seul objet logique. Il suffit de créer ou de supprimer cet objet logique pour intégrer ou supprimer tous les objets qu'il contient. Cela facilite grandement l'installation et la désinstallation de nombreux objets. Les extensions peuvent être codées en différents langages, généralement en C ou en PL/SQL. Elles ont eu un grand succès.

La possibilité de développer des routines dans différents langages en est un exemple : perl, python, PHP, Ruby ou JavaScript sont disponibles. PL/pgSQL est lui-même une extension à proprement parler, toujours présente.

Autre exemple : la possibilité d'ajouter des types de données, des routines et des opérateurs a permis l'émergence de la couche spatiale de PostgreSQL (appelée PostGIS).

Les provenances, rôle et niveau de finition des extensions sont très variables. Certaines sont des utilitaires éprouvés fournis avec PostgreSQL (parmi les « contrib »). D'autres sont des utilitaires aussi complexes que PostGIS ou un langage de procédures stockées. Des éditeurs diffusent leur produit comme une extension plutôt que *forker* PostgreSQL (Citus, timescaledb…). Beaucoup d'extensions peuvent être installées très simplement depuis des paquets disponibles dans les dépôts habituels (de la distribution ou du PGDG), ou le site du concepteur. Certaines sont diffusées comme code source à compiler. Comme tout logiciel, il faut faire attention à en vérifier la source, la qualité, la réputation et la pérennité.

Une fois les binaires de l'extension en place sur le serveur, l'ordre `CREATE EXTENSION` suffit généralement dans la base cible, et les fonctionnalités sont immédiatement exploitables.

Les extensions sont habituellement installées par un administrateur (un utilisateur doté de l'attribut `SUPERUSER`). À partir de la version 13, certaines extensions sont déclarées de confiance `trusted`). Ces extensions peuvent être installées par un utilisateur standard (à condition qu'il dispose des droits de création dans la base et le ou les schémas concernés).

Les développeurs de PostgreSQL ont aussi ajouté des *hooks* pour accrocher du code à exécuter sur certains cas. Cela a permis entre autres de créer l'extension `pg_stat_statements` qui s'accroche au code de l'exécuteur de requêtes pour savoir quelles sont les requêtes exécutées et pour récupérer des statistiques sur ces requêtes.

Enfin, les *background workers* ont vu le jour. Ce sont des processus spécifiques lancés par le serveur PostgreSQL lors de son démarrage et stoppés lors de son arrêt. Cela a permis la création de PoWA (outil qui historise les statistiques sur les requêtes) et une amélioration très intéressante de `pg_prewarm` (sauvegarde du contenu du cache disque à l'arrêt de PostgreSQL, restauration du contenu au démarrage).

Des exemples d'extensions sont décrites dans nos modules Extensions PostgreSQL pour l'utilisateur[4], Extensions PostgreSQL pour la performance[5], Extensions PostgreSQL pour les DBA[6].

2.2.10 Sécurité

- Fichier `pg_hba.conf`
- Filtrage IP
- Authentification interne (MD5, SCRAM-SHA-256)
- Authentification externe (identd, LDAP, Kerberos…)
- Support natif de SSL

Le filtrage des connexions se paramètre dans le fichier de configuration `pg_hba.conf`. Nous pouvons y définir quels utilisateurs (déclarés auprès de PostgreSQL) peuvent se connecter à quelles bases, et depuis quelles adresses IP.

L'authentification peut se baser sur des mots de passe chiffrés propres à PostgreSQL (`md5` ou le plus récent et plus sécurisé `scram-sha-256` en version 10), ou se baser sur une méthode externe (auprès de l'OS, ou notamment LDAP ou Kerberos qui couvre aussi Active Directory).

L'authentification et le chiffrement de la connexion par SSL sont couverts.

[4] https://dali.bo/x1_html
[5] https://dali.bo/x2_html
[6] https://dali.bo/x3_html

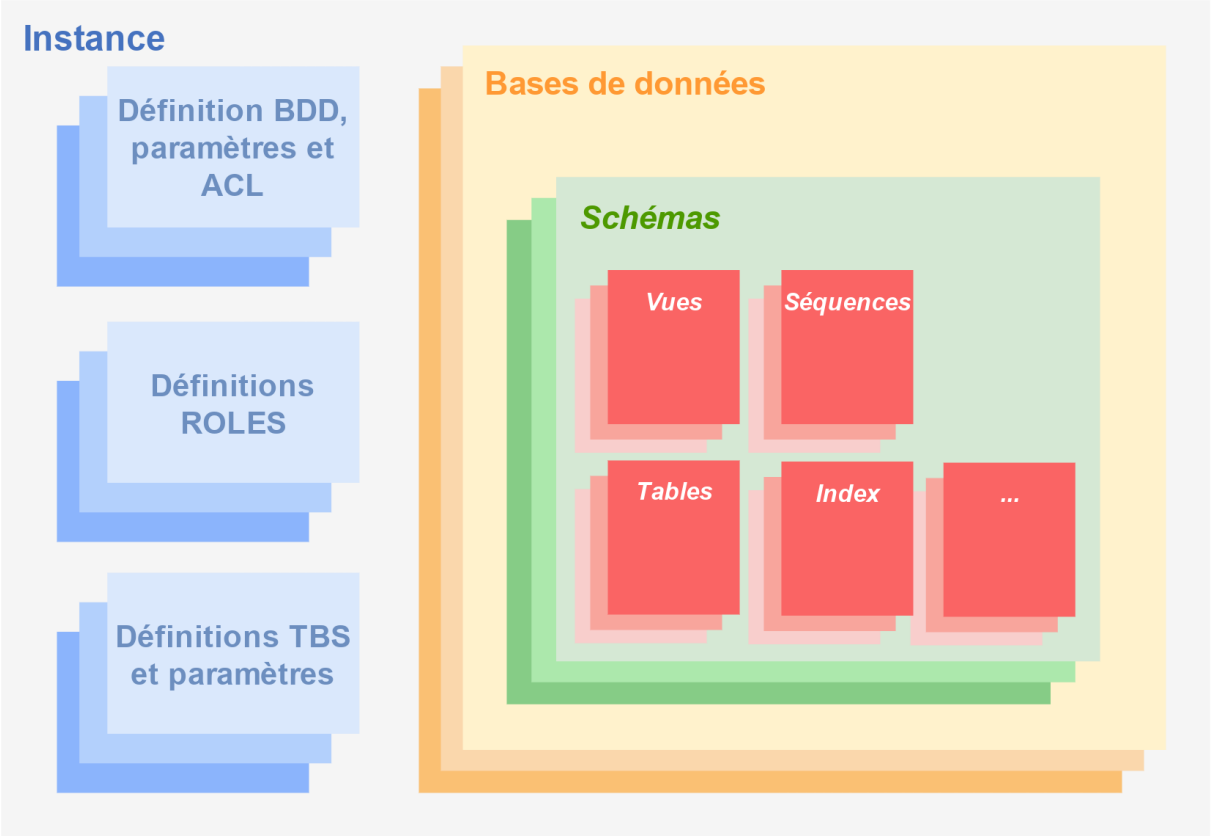
2.3 OBJETS SQL

- Instances
- Objets globaux :
  - Bases
  - Rôles
  - Tablespaces
- Objets locaux :
  - Schémas
  - Tables
  - Vues
  - Index
  - Routines
  - …

Le but de cette partie est de passer en revue les différents objets logiques maniés par un moteur de bases de données PostgreSQL.

Nous allons donc aborder la notion d'instance, les différents objets globaux et les objets locaux. Tous ne seront pas vus, mais le but est de donner une idée globale des objets et des fonctionnalités de PostgreSQL.

2.3.1 Organisation logique



Il est déjà important de bien comprendre une distinction entre les objets. Une instance est un ensemble de bases de données, de rôles et de tablespaces. Ces objets sont appelés des objets globaux parce qu'ils sont disponibles quelque soit la base de données de connexion. Chaque base de données contient ensuite des objets qui lui sont propres. Ils sont spécifiques à cette base de données et accessibles uniquement lorsque l'utilisateur est connecté à la base qui les contient. Il est donc possible de voir les bases comme des conteneurs hermétiques en dehors des objets globaux.

2.3.2 Instances

- Une instance
  - un répertoire de données
  - un port TCP
  - une configuration
  - plusieurs bases de données
- Plusieurs instances possibles sur un serveur

Une instance est un ensemble de bases de données. Après avoir installé PostgreSQL, il est nécessaire de créer un répertoire de données contenant un certain nombre de répertoires et de fichiers qui permettront à PostgreSQL de fonctionner de façon fiable. Le contenu de ce répertoire est créé initialement par la commande `initdb`. Ce répertoire stocke ensuite tous les objets des bases de données de l'instance, ainsi que leur contenu.

Chaque instance a sa propre configuration. Il n'est possible de lancer qu'un seul `postmaster` par instance, et ce dernier acceptera les connexions à partir d'un port TCP spécifique.

Il est possible d'avoir plusieurs instances sur le même serveur, physique ou virtuel. Dans ce cas, chaque instance aura son répertoire de données dédié et son port TCP dédié. Ceci est particulièrement utile quand l'on souhaite disposer de plusieurs versions de PostgreSQL sur le même serveur (par exemple pour tester une application sur ces différentes versions).

2.3.3 Rôles

- Utilisateurs / Groupes
  - Utilisateur : Permet de se connecter
- Différents attributs et droits

Une instance contient un ensemble de rôles. Certains sont prédéfinis et permettent de disposer de droits particuliers (lecture de fichier avec `pg_read_server_files`, annulation d'une requête avec `pg_signal_backend`, etc). Cependant, la majorité est composée de rôles créés pour permettre la connexion des utilisateurs.

Chaque rôle créé peut être utilisé pour se connecter à n'importe quelle base de l'instance, à condition que ce rôle en ait le droit. Ceci se gère directement avec l'attribution du droit `LOGIN` au rôle, et avec

la configuration du fichier d'accès `pg_hba.conf`.

Chaque rôle peut être propriétaire d'objets, auquel cas il a tous les droits sur ces objets. Pour les objets dont il n'est pas propriétaire, il peut se voir donner des droits, en lecture, écriture, exécution, etc par le propriétaire.

Nous parlons aussi d'utilisateurs et de groupes. Un utilisateur est un rôle qui a la possibilité de se connecter aux bases alors qu'un groupe ne le peut pas. Un groupe sert principalement à gérer plus simplement les droits d'accès aux objets.

2.3.4 Tablespaces

- Répertoire physique contenant les fichiers de données de l'instance
- Une base peut
  - se trouver sur un seul tablespace
  - être répartie sur plusieurs tablespaces
- Permet de gérer l'espace disque et les performances
- Pas de quota

Toutes les données des tables, vues matérialisées et index sont stockées dans le répertoire de données principal. Cependant, il est possible de stocker des données ailleurs que dans ce répertoire. Il faut pour cela créer un tablespace. Un tablespace est tout simplement la déclaration d'un autre répertoire de données utilisable par PostgreSQL pour y stocker des données :

```
CREATE TABLESPACE chaud LOCATION '/SSD/tbl/chaud';
```

Il est possible d'avoir un tablespace par défaut pour une base de données, auquel cas tous les objets logiques créés dans cette base seront enregistrés physiquement dans le répertoire lié à ce tablespace. Il est aussi possible de créer des objets en indiquant spécifiquement un tablespace, ou de les déplacer d'un tablespace à un autre. Un objet spécifique ne peut appartenir qu'à un seul tablespace (autrement dit, un index ne pourra pas être enregistré sur deux tablespaces). Cependant, pour les objets partitionnés, le choix du tablespace peut se faire partition par partition.

Le but des tablespaces est de fournir une solution à des problèmes d'espace disque ou de performances. Si la partition où est stocké le répertoire des données principal se remplit fortement, il est possible de créer un tablespace dans une autre partition et donc d'utiliser l'espace disque de cette partition. Si de nouveaux disques plus rapides sont à disposition, il est possible de placer les objets fréquemment utilisés sur le tablespace contenant les disques rapides. Si des disques SSD sont à disposition, il est très intéressant d'y placer les index, les fichiers de tri temporaires, des tables de travail…

Par contre, contrairement à d'autres moteurs de bases de données, PostgreSQL n'a pas de notion de quotas. Les tablespaces ne peuvent donc pas être utilisés pour contraindre l'espace disque utilisé par certaines applications ou certains rôles.

2.3.5 Bases

- Conteneur hermétique
- Un rôle ne se connecte pas à une instance
  - il se connecte forcément à une base
- Une fois connecté, il ne voit que les objets de cette base
  - contournement : foreign data wrappers, dblink

Une base de données est un conteneur hermétique. En dehors des objets globaux, le rôle connecté à une base de données ne voit et ne peut interagir qu'avec les objets contenus dans cette base. De même, il ne voit pas les objets locaux des autres bases. Néanmoins, il est possible de lui donner le droit d'accéder à certains objets d'une autre base (de la même instance ou d'une autre instance) en utilisant les *Foreign Data Wrappers* (`postgres_fdw`) ou l'extension `dblink` .

Un rôle ne se connecte pas à l'instance. Il se connecte forcément à une base spécifique.

2.3.6 Schémas

- Espace de noms
- Sous-ensemble de la base
- Non lié à un utilisateur
- Résolution des objets : `search_path`
- `pg_catalog` , `information_schema`
  - pour catalogues système (lecture seule !)

Les schémas sont des espaces de noms à l'intérieur d'une base de données permettant :

- de grouper logiquement les objets d'une base de données ;
- de séparer les utilisateurs entre eux ;

- de contrôler plus efficacement les accès aux données ;
- d'éviter les conflits de noms dans les grosses bases de données.

Un schéma n'a à priori aucun lien avec un utilisateur donné.

Un schéma est un espace logique sans lien avec les emplacements physiques des données (ne pas confondre avec les *tablespaces*).

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble, tout dépend des droits dont il dispose. Depuis la version 15, un nouvel utilisateur n'a le droit de créer d'objet nulle part. Dans les versions précédentes, il avait accès au schéma `public` de chaque base et pouvait y créer des objets.

Lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête, PostgreSQL recherche les objets dans les schémas listés par le paramètre `search_path` valable pour la session en cours .

Voici un exemple d'utilisation des schémas :

```
-- Création de deux schémas
CREATE SCHEMA s1;
CREATE SCHEMA s2;

-- Création d'une table sans spécification du schéma
CREATE TABLE t1 (id integer);

-- Comme le montre la méta-commande \d, la table est créée dans le schéma public

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres

-- Ceci est dû à la configuration par défaut du paramètre search_path
-- modification du search_path
SET search_path TO s1;

-- création d'une nouvelle table sans spécification du schéma
CREATE TABLE t2 (id integer);

-- Cette fois, le schéma de la nouvelle table est s1
-- car la configuration du search_path est à s1
-- Nous pouvons aussi remarquer que les tables capitaines et s1
-- ne sont plus affichées
-- Ceci est dû au fait que le search_path ne contient que le schéma s1 et
-- n'affiche donc que les objets de ce schéma.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 s1     | t2   | table | postgres
```

```
-- Nouvelle modification du search_path
SET search_path TO s1, public;

-- Cette fois, les deux tables apparaissent

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
CREATE TABLE s2.t3 (id integer);

-- changement du search_path pour voir la table
SET search_path TO s1, s2, public;

-- La table apparaît bien, et le schéma d'appartenance est bien s2

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
-- attention, cette table a un nom déjà utilisé par une autre table
CREATE TABLE s2.t2 (id integer);

-- La création se passe bien car, même si le nom de la table est identique,
-- le schéma est différent
-- Par contre, \d ne montre que la première occurence de la table
-- ici, nous ne voyons t2 que dans s1

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Changeons le search_path pour placer s2 avant s1
SET search_path TO s2, s1, public;

-- Maintenant, la seule table t2 affichée est celle du schéma s2

postgres=# \d
                List of relations
```

```
 Schema |       Name         |   Type   |  Owner
--------+--------------------+----------+----------
 public | capitaines         | table    | postgres
 public | capitaines_id_seq  | sequence | postgres
 public | t1                 | table    | postgres
 s2     | t2                 | table    | postgres
 s2     | t3                 | table    | postgres
```

Tous ces exemples se basent sur des ordres de création de table. Cependant, le comportement serait identique sur d'autres types de commande (`SELECT` , `INSERT` , etc) et sur d'autres types d'objets locaux.

Pour des raisons de sécurité, il est très fortement conseillé de laisser le schéma `public` en toute fin du `search_path` . En effet, avant la version 15, s'il est placé au début, comme tout le monde avait le droit de créer des objets dans `public` , quelqu'un de mal intentionné pouvait placer un objet dans le schéma `public` pour servir de proxy à un autre objet d'un schéma situé après `public` . Même si la version 15 élimine ce risque, il reste la bonne pratique d'adapter le `search_path` pour placer les schémas applicatifs en premier.

Les schémas `pg_catalog` et `information_schema` contiennent des tables utilitaires (« catalogues système ») et des vues. Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables, et les colonnes, et des données de suivi interne. Dans PostgreSQL, ce sont de simples tables. Un simple utilisateur lit fréquemment ces tables, plus ou moins directement, mais n'a aucune raison d'y modifier des données. Toutes les opérations habituelles pour un utilisateur ou administrateur sont disponibles sous la forme de commandes SQL.

Ne modifiez jamais directement les tables et vues système dans les schémas `pg_catalog` et `information_schema` ; n'y ajoutez ni n'y effacez jamais rien !

Même si cela est techniquement possible, seules des exceptions particulièrement ésotériques peuvent justifier une modification directe des tables systèmes (par exemple, une correction de vue système, suite à un bug corrigé dans une version mineure). Ces tables n'apparaissent d'ailleurs pas dans une sauvegarde logique (`pg_dump`).

2.3.7 Tables

Par défaut, une table est :

- Permanente
  - si temporaire, vivra le temps de la session (ou de la transaction)
- Journalisée
  - si *unlogged*, perdue en cas de crash, pas de réplication
- Non partitionnée
  - partitionnement possible par intervalle, valeur ou hachage

Par défaut, les tables sont permanentes, journalisées et non partitionnées.

Il est possible de créer des tables temporaires (`CREATE TEMPORARY TABLE`). Celles-ci ne sont visibles que par la session qui les a créées et seront supprimées par défaut à la fin de cette session. Il est aussi possible de les supprimer automatiquement à la fin de la transaction qui les a créées. Il n'existe pas dans PostgreSQL de notion de table temporaire globale. Cependant, une extension[7] existe pour combler leur absence.

Pour des raisons de performance, il est possible de créer une table non journalisée (`CREATE UNLOGGED TABLE`). La définition de la table est journalisée mais pas son contenu. De ce fait, en cas de crash, il est impossible de dire si la table est corrompue ou non, et donc, au redémarrage du serveur, PostgreSQL vide la table de tout contenu. De plus, n'étant pas journalisée, la table n'est pas présente dans les sauvegardes PITR, ni repliquée vers d'éventuels serveurs secondaires.

Enfin, depuis la version 10, il est possible de partitionner les tables suivant un certain type de partitionnement : par intervalle, par valeur ou par hachage.

[7] https://github.com/darold/pgtt

2.3.8 Vues

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Vues matérialisées
  - à rafraîchir à une certaine fréquence

Le but des vues est de masquer une complexité, qu'elle soit du côté de la structure de la base ou de l'organisation des accès. Dans le premier cas, elles permettent de fournir un accès qui ne change pas même si les structures des tables évoluent. Dans le second cas, elles permettent l'accès à seulement certaines colonnes ou certaines lignes. De plus, les vues étant exécutées avec les mêmes droits que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Voici un exemple d'utilisation :

```
SET search_path TO public;

-- création de l'utilisateur guillaume
-- il n'aura pas accès à la table capitaines
-- par contre, il aura accès à la vue capitaines_anon
CREATE ROLE guillaume LOGIN;

-- ajoutons une colonne à la table capitaines
-- et ajoutons-y des données
ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
INSERT INTO capitaines (nom, age, num_cartecredit)
  VALUES ('Robert Surcouf', 20, '1234567890123456');

-- création de la vue
CREATE VIEW capitaines_anon AS
  SELECT nom, age, substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- ajout du droit de lecture à l'utilisateur guillaume
GRANT SELECT ON TABLE capitaines_anon TO guillaume;

-- connexion en tant qu'utilisateur guillaume
SET ROLE TO guillaume;

-- vérification qu'on lit bien la vue mais pas la table
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
```

```
 Robert Surcouf |  20 | 123456789******
```

```
-- tentative de lecture directe de la table
SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

Il est possible de modifier une vue en lui ajoutant des colonnes à la fin, au lieu de devoir les détruire et recréer (ainsi que toutes les vues qui en dépendent, ce qui peut être fastidieux).

Par exemple :

```
SET ROLE postgres;

CREATE OR REPLACE VIEW capitaines_anon AS SELECT
  nom,age,substring(num_cartecredit,0,10)||'******' AS num_cc_anon,
  md5(substring(num_cartecredit,0,10)) AS num_md5_cc
  FROM capitaines;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
      nom       | age |   num_cc_anon    |            num_md5_cc
----------------+-----+------------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

Nous pouvons aussi modifier les données au travers des vues simples, sans ajout de code et de trigger :

```
UPDATE capitaines_anon SET nom = 'Nicolas Surcouf' WHERE nom = 'Robert Surcouf';

SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
       nom       | age |   num_cc_anon    |            num_md5_cc
-----------------+-----+------------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

```
UPDATE capitaines_anon SET num_cc_anon = '123456789xxxxxx'
  WHERE nom = 'Nicolas Surcouf';

ERROR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

PostgreSQL gère le support natif des vues matérialisées (`CREATE MATERIALIZED VIEW nom_vue_mat AS SELECT …`). Les vues matérialisées sont des vues dont le contenu est figé sur disque, permettant de ne pas recalculer leur contenu à chaque appel. De plus, il est possible de les indexer pour accélérer leur consultation. Il faut cependant faire attention à ce que leur contenu reste synchrone avec le reste des données.

Les vues matérialisées ne sont pas mises à jour automatiquement, il faut demander explicitement le rafraîchissement (`REFRESH MATERIALIZED VIEW`). Avec la clause `CONCURRENTLY` , s'il y a un index d'unicité, le rafraîchissement ne bloque pas les sessions lisant en même temps les données d'une vue matérialisée.

```
-- Suppression de la vue
DROP VIEW capitaines_anon;
```

```
-- Création de la vue matérialisée
CREATE MATERIALIZED VIEW capitaines_anon AS
  SELECT nom,
    age,
    substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- Les données sont bien dans la vue matérialisée
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Mise à jour d'une ligne de la table
-- Cette mise à jour est bien effectuée, mais la vue matérialisée
-- n'est pas impactée
UPDATE capitaines SET nom = 'Robert Surcouf' WHERE nom = 'Nicolas Surcouf';

SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';

 id |      nom       | age | num_cartecredit
----+----------------+-----+------------------
  1 | Robert Surcouf |  20 | 1234567890123456

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Le résultat est le même mais le plan montre bien que PostgreSQL ne passe
-- plus par la table mais par la vue matérialisée :
EXPLAIN SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

                         QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on capitaines_anon  (cost=0.00..20.62 rows=1 width=68)
   Filter: (nom ~~ '%Surcouf'::text)

-- Après un rafraîchissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
REFRESH MATERIALIZED VIEW capitaines_anon;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions :

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;

ERROR:  cannot refresh materialized view "public.capitaines_anon" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns
       of the materialized view.
```

```
-- En effet, il faut un index d'unicité pour faire un rafraîchissement
-- sans bloquer les autres sessions.
CREATE UNIQUE INDEX ON capitaines_anon(nom);

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
```

2.3.9 Index

- Algorithmes supportés
  - B-tree (par défaut)
  - Hash
  - GiST / SP-GiST
  - GIN
  - BRIN (version 9.5)
  - Bloom (version 9.6)
- Type
  - Mono ou multi-colonnes
  - Partiel
  - Fonctionnel
  - Couvrant

PostgreSQL propose plusieurs algorithmes d'index.

Pour une indexation standard, nous utilisons en général un index Btree, de par ses nombreuses possibilités et ses très bonnes performances.

Les index hash sont peu utilisés, essentiellement dans la comparaison d'égalité de grandes chaînes de caractères.

Moins simples d'abord, les index plus spécifiques (GIN, GIST) sont spécialisés pour les grands volumes de données complexes et multidimensionnelles : indexation textuelle, géométrique, géographique, ou de tableaux de données par exemple.

Les index BRIN sont des index très compacts destinés aux grandes tables où les données sont fortement corrélées par rapport à leur emplacement physique sur les disques.

Les index bloom sont des index probabilistes visant à indexer de nombreuses colonnes interrogées simultanément. Ils nécessitent l'ajout d'une extension (nommée `bloom`). Contrairement aux index btree, les index bloom ne dépendant pas de l'ordre des colonnes.

Le module `pg_trgm` permet l'utilisation d'index dans des cas habituellement impossibles, comme les expressions rationnelles et les `LIKE '%...%'`.

Généralement, l'indexation porte sur la valeur d'une ou plusieurs colonnes. Il est néanmoins possible de n'indexer qu'une partie des lignes (index partiel) ou le résultat d'une fonction sur une ou plusieurs colonnes en paramètre. Enfin, il est aussi possible de modifier les index de certaines contraintes (unicité et clé primaire) pour inclure des colonnes supplémentaires.

Plus d'informations :

- Article Wikipédia sur les arbres B[8] ;
- Article Wikipédia sur les tables de hachage[9] ;
- Documentation officielle française[10].

2.3.10 Types de données

- Types de base
  - natif : `int`, `float`
  - standard SQL : `numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`
- Type complexe
  - tableau
  - JSON (`jsonb`), XML
  - vecteur (données LLM, FTS)
- Types métier
  - réseau, géométrique, etc.
- Types créés par les utilisateurs
  - structure SQL, C, Domaine, Enum

PostgreSQL dispose d'un grand nombre de types de base, certains natifs (comme la famille des `integer` et celle des `float`), et certains issus de la norme SQL (`numeric` , `char` , `varchar` , `date` , `time` , `timestamp` , `bool`).

Il dispose aussi de types plus complexes. Les tableaux (`array`) permettent de lister un ensemble de valeurs discontinues. Les intervalles (`range`) permettent d'indiquer toutes les valeurs comprises entre une valeur de début et une valeur de fin. Ces deux types dépendent évidemment d'un type de

base : tableau d'entiers, intervalle de dates, etc. Existent aussi les types complexes les données XML et JSON (préférer le type optimisé `jsonb`).

PostgreSQL sait travailler avec des vecteurs pour des calculs avancé. De base, le type `tsvector` permet la recherche plein texte, avec calcul de proximité de mots dans un texte, pondération des résultats, etc. L'extension `pgvector` permet de stocker et d'indexer des vecteurs utilisé par les algorithmes LLM implémentés dans les IA génératives.

Enfin, il existe des types métiers ayant trait principalement au réseau (adresse IP, masque réseau), à la géométrie (point, ligne, boite). Certains sont apportés par des extensions.

Tout ce qui vient d'être décrit est natif. Il est cependant possible de créer ses propres types de données, soit en SQL soit en C. Les possibilités et les performances ne sont évidemment pas les mêmes.

Voici comment créer un type en SQL :

```
CREATE TYPE serveur AS (
  nom             text,
  adresse_ip      inet,
  administrateur  text
);
```

Ce type de données va pouvoir être utilisé dans tous les objets SQL habituels : table, routine, opérateur (pour redéfinir l'opérateur `+` par exemple), fonction d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

(Il faut au préalable avoir défini le type `stock` et la fonction `stock_fusion`.)

Il est aussi possible de définir des domaines. Ce sont des types créés par les utilisateurs à partir d'un type de base et en lui ajoutant des contraintes supplémentaires.

En voici un exemple :

```
CREATE DOMAIN code_postal_francais AS text CHECK (value ~ '^\d{5}$');
ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
UPDATE capitaines SET cp = '35400' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET cp = '1420' WHERE nom = 'Haddock';

ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"

UPDATE capitaines SET cp = '01420' WHERE nom = 'Haddock';
SELECT * FROM capitaines;

 id |      nom       | age | num_cartecredit  |  cp
----+----------------+-----+------------------+-------
  1 | Robert Surcouf |  20 | 1234567890123456 | 35400
  1 | Haddock        |  35 |                  | 01420
```

Les domaines permettent d'intégrer la déclaration des contraintes à la déclaration d'un type, et donc de simplifier la maintenance de l'application si ce type peut être utilisé dans plusieurs tables : si la définition du code postal est insuffisante pour une évolution de l'application, il est possible de la modifier par un `ALTER DOMAIN`, et définir de nouvelles contraintes sur le domaine. Ces contraintes seront vérifiées sur l'ensemble des champs ayant le domaine comme type avant que la nouvelle version du type ne soit considérée comme valide.

Le défaut par rapport à des contraintes `CHECK` classiques sur une table est que l'information ne se trouvant pas dans la table, les contraintes sont plus difficiles à lister sur une table.

Enfin, il existe aussi les enums. Ce sont des types créés par les utilisateurs composés d'une liste ordonnée de chaînes de caractères.

En voici un exemple :

```
CREATE TYPE jour_semaine
  AS ENUM ('Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi',
  'Samedi', 'Dimanche');

ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;

UPDATE capitaines SET jour_sortie = 'Mardi' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET jour_sortie = 'Samedi' WHERE nom LIKE 'Haddock';

SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
```

```
 id |   nom   | age | num_cartecredit | cp | jour_sortie
----+---------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
```

Les *enums* permettent de déclarer une liste de valeurs statiques dans le dictionnaire de données plutôt que dans une table externe sur laquelle il faudrait rajouter des jointures : dans l'exemple, nous aurions pu créer une table `jour_de_la_semaine`, et stocker la clé associée dans `planning`. Nous aurions pu tout aussi bien positionner une contrainte `CHECK`, mais nous n'aurions plus eu une liste ordonnée.

Conférence de Heikki Linakangas sur la création d'un type color[11].

2.3.11 Contraintes

- `CHECK`
  - `prix > 0`
- `NOT NULL`
  - `id_client NOT NULL`
- Unicité
  - `id_client UNIQUE`
- Clés primaires
  - `UNIQUE NOT NULL` ==> `PRIMARY KEY (id_client)`
- Clés étrangères
  - `produit_id REFERENCES produits(id_produit)`
- `EXCLUDE`
  - `EXCLUDE USING gist (room WITH =, during WITH &&)`

Les contraintes sont la garantie de conserver des données de qualité ! Elles permettent une vérification qualitative des données, beaucoup plus fine qu'en définissant uniquement un type de données.

Les exemples ci-dessus reprennent :

- un prix qui doit être strictement positif ;
- un identifiant qui ne doit pas être vide (sinon des jointures filtreraient des lignes) ;
- une valeur qui doit être unique (comme des numéros de clients ou de facture) ;
- une clé primaire (unique non nulle), qui permet d'identifier précisément une ligne ;
- une clé étrangère vers la clé primaire d'une autre table (là encore pour garantir l'intégrité des jointures) ;
- une contrainte d'exclusion interdisant que deux plages temporelles se recouvrent dans la réservation de la même salle de réunion.

Les contraintes d'exclusion permettent un test sur plusieurs colonnes avec différents opérateurs (et non uniquement l'égalité, comme dans le cas d'une contrainte unique, qui n'est qu'une contrainte d'exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

Une contrainte peut porter sur plusieurs champs et un champ peut être impliqué dans plusieurs

contraintes :

```
CREATE TABLE commandes (
    no_commande     varchar(16) CHECK (no_commande ~ '^[A-Z0-9]*$'),
    id_entite_commerciale int REFERENCES entites_commerciales,
    id_client       int       REFERENCES clients,
    date_commande   date      NOT NULL,
    date_livraison  date      CHECK (date_livraison >= date_commande),
    PRIMARY KEY (no_commande, id_entite_commerciale)
);
```

```
\d commandes
                              Table « public.commandes »
        Colonne        |         Type          | … | NULL-able | Par défaut
-----------------------+-----------------------+---+-----------+------------
 no_commande           | character varying(16) |   | not null  |
 id_entite_commerciale | integer               |   | not null  |
 id_client             | integer               |   |           |
 date_commande         | date                  |   | not null  |
 date_livraison        | date                  |   |           |
Index :
    "commandes_pkey" PRIMARY KEY, btree (no_commande, id_entite_commerciale)
Contraintes de vérification :
    "commandes_check" CHECK (date_livraison >= date_commande)
    "commandes_no_commande_check" CHECK (no_commande::text ~ '^[A-Z0-9]*$'::text)
Contraintes de clés étrangères :
    "commandes_id_client_fkey" FOREIGN KEY (id_client) REFERENCES clients(id_client)
    "commandes_id_entite_commerciale_fkey" FOREIGN KEY (id_entite_commerciale)
 ↪  REFERENCES entites_commerciales(id_entite_commerciale)
```

Les contraintes doivent être vues comme la dernière ligne de défense de votre application face aux bugs. En effet, le code d'une application change beaucoup plus souvent que le schéma, et les données survivent souvent à l'application, qui peut être réécrite entretemps. Quoi qu'il se passe, des contraintes judicieuses garantissent qu'il n'y aura pas d'incohérence logique dans la base.

Si elles sont gênantes pour le développeur (car elles imposent un ordre d'insertion ou de mise à jour), il faut se rappeler que les contraintes peuvent être « débrayées » le temps d'une transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED ;
…
COMMIT ;
```

Les contraintes ne seront validées qu'au `COMMIT`.

Sur le sujet, voir par exemple *Constraints: a Developer's Secret Weapon*[12] de Will Leinweber (pgDay Paris 2018) (slides[13], vidéo[14]).

Du point de vue des performances, les contraintes permettent au planificateur d'optimiser les requêtes. Par exemple, le planificateur sait ne pas prendre en compte certaines jointures, notamment grâce à l'existence d'une contrainte d'unicité. (Sur ce point, la version 15 améliore les contraintes

[12] https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2192-constraints-a-developers-secret-weapon/
[13] https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
[14] https://youtu.be/hWh8QoV8z8k

d'unicité en permettant de choisir si la valeur NULL est considérée comme unique ou pas. Par défaut et historiquement, une valeur NULL n'étant pas égal à une valeur NULL, les valeurs NULL sont considérées distinctes, et donc on peut avoir plusieurs valeurs NULL dans une colonne ayant une contrainte d'unicité.)

2.3.12 Colonnes à valeur générée

- Valeur calculée à l'insertion
- `DEFAULT`
- Identité
  - `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`
- Expression
  - `GENERATED ALWAYS AS ( generation_expr ) STORED`

Une colonne a par défaut la valeur `NULL` si aucune valeur n'est fournie lors de l'insertion de la ligne. Il existe néanmoins trois cas où le moteur peut substituer une autre valeur.

Le plus connu correspond à la clause `DEFAULT`. Dans ce cas, la valeur insérée correspond à la valeur indiquée avec cette clause si aucune valeur n'est indiquée pour la colonne. Si une valeur est précisée, cette valeur surcharge la valeur par défaut. L'exemple suivant montre cela :

```
CREATE TABLE t2 (c1 integer, c2 integer, c3 integer DEFAULT 10);
INSERT INTO t2 (c1, c2, c3) VALUES (1, 2, 3);
INSERT INTO t2 (c1) VALUES (2);
SELECT * FROM t2;
```

```
 c1 | c2 | c3
----+----+----
  1 |  2 |  3
  2 |    | 10
```

La clause `DEFAULT` ne peut pas être utilisée avec des clauses complexes, notamment des clauses comprenant des requêtes.

Pour aller un peu plus loin, à partir de PostgreSQL 12, il est possible d'utiliser `GENERATED ALWAYS AS ( expression )`. Cela permet d'avoir une valeur calculée pour la colonne, valeur qui ne peut pas être surchargée, ni à l'insertion, ni à la mise à jour (mais qui est bien stockée sur le disque).

Comme exemple, nous allons reprendre la table `capitaines` et lui ajouter une colonne ayant comme valeur la version modifiée du numéro de carte de crédit :

```
ALTER TABLE capitaines
  ADD COLUMN num_cc_anon text
```

```
  GENERATED ALWAYS AS (substring(num_cartecredit, 0, 10) || '******') STORED;

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

      nom       | num_cartecredit  |   num_cc_anon
----------------+------------------+-----------------
 Robert Surcouf | 1234567890123456 | 123456789******
 Haddock        |                  |

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi', 'test');

ERROR:  cannot insert into column "num_cc_anon"
DETAIL:  Column "num_cc_anon" is a generated column.

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi');

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

          nom          | num_cartecredit  |   num_cc_anon
-----------------------+------------------+-----------------
 Robert Surcouf        | 1234567890123456 | 123456789******
 Haddock               |                  |
 Joseph Pradere-Niquet | 9876543210987654 | 987654321******
```

Enfin, `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY` permet d'obtenir une colonne d'identité, bien meilleure que ce que le pseudo-type `serial` propose. Si `ALWAYS` est indiqué, la valeur n'est pas modifiable.

```
ALTER TABLE capitaines
  ADD COLUMN id2 integer GENERATED ALWAYS AS IDENTITY;

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3

INSERT INTO capitaines (nom) VALUES ('Tom Souville');

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3
 Tom Souville          |   4
```

Le type `serial` est remplacé par le type `integer` et une séquence comme le montre l'exemple suivant. C'est un problème dans la mesure ou la déclaration qui est faite à la création de la table produit un résultat différent en base et donc dans les exports de données.

```
CREATE TABLE tserial(s serial);
```

```
                          Table "public.tserial"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 s      | integer |           | not null | nextval('tserial_s_seq'::regclass)
```

2.3.13 Langages

- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQL
- Extensions officielles : Perl, Python
- Mais aussi Java, Ruby, Javascript…
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

- PL/pgSQL ;
- PL/Perl[15] ;
- PL/Python[16] ;
- PL/Tcl.

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

- PL/sh[17] ;
- PL/R[18] ;
- PL/Java[19] ;
- PL/lolcode ;
- PL/Scheme ;
- PL/PHP ;
- PL/Ruby ;
- PL/Lua[20] ;
- PL/pgPSM ;
- PL/v8[21] (Javascript).

[15] https://docs.postgresql.fr/current/plperl.html
[16] https://docs.postgresql.fr/current/plpython.html
[17] https://github.com/petere/plsh
[18] https://github.com/postgres-plr/plr
[19] https://tada.github.io/pljava/
[20] https://github.com/pllua/pllua
[21] https://github.com/plv8/plv8

Tableau des langages supportés[22].

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

2.3.14 Fonctions & procédures

- Fonction
  - renvoie une ou plusieurs valeurs
  - `SETOF` ou `TABLE` pour plusieurs lignes
- Procédure (v11+)
  - ne renvoie rien
  - peut gérer le transactionnel dans certains cas

Historiquement, PostgreSQL ne proposait que l'écriture de fonctions. Depuis la version 11, il est aussi possible de créer des procédures. Le terme « routine » est utilisé pour signifier procédure ou fonction.

Une fonction renvoie une donnée. Cette donnée peut comporter une ou plusieurs colonnes. Elle peut aussi avoir plusieurs lignes dans le cas d'une fonction `SETOF` ou `TABLE`.

Une procédure ne renvoie rien. Elle a cependant un gros avantage par rapport aux fonctions dans le fait qu'elle peut gérer le transactionnel. Elle peut valider ou annuler la transaction en cours. Dans ce cas, une nouvelle transaction est ouverte immédiatement après la fin de la transaction précédente.

2.3.15 Opérateurs

- Dépend d'un ou deux types de données
- Utilise une fonction prédéfinie :

```
CREATE OPERATOR //
  (FUNCTION=division0,
  LEFTARG=integer,
  RIGHTARG=integer);
```

Il est possible de créer de nouveaux opérateurs sur un type de base ou sur un type utilisateur. Un opérateur exécute une fonction, soit à un argument pour un opérateur unitaire, soit à deux arguments pour un opérateur binaire.

Voici un exemple d'opérateur acceptant une division par zéro sans erreur :

```
-- définissons une fonction de division en PL/pgSQL
CREATE FUNCTION division0 (p1 integer, p2 integer) RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
  IF p2 = 0 THEN
      RETURN NULL;
  END IF;

  RETURN p1 / p2;
END
$$;

-- créons l'opérateur
CREATE OPERATOR // (FUNCTION = division0, LEFTARG = integer, RIGHTARG = integer);

-- une division normale se passe bien

SELECT 10/5;

 ?column?
----------
        2

SELECT 10//5;

 ?column?
----------
        2
```

```
-- une division par 0 ramène une erreur avec l'opérateur natif
SELECT 10/0;

ERROR:  division by zero

-- une division par 0 renvoie NULL avec notre opérateur
SELECT 10//0;

 ?column?
----------

(1 row)
```

2.3.16 Triggers

- Opérations : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- Trigger sur :
  - une colonne, et/ou avec condition
  - une vue
  - DDL
- Tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)
- N'importe quel langage supporté

Les triggers peuvent être exécutés avant (`BEFORE`) ou après (`AFTER`) une opération.

Il est possible de les déclencher pour chaque ligne impactée (`FOR EACH ROW`) ou une seule fois pour l'ensemble de la requête (`FOR STATEMENT`). Dans le premier cas, il est possible d'accéder à la ligne impactée (ancienne et nouvelle version). Dans le deuxième cas, il a fallu attendre la version 10 pour disposer des tables de transition qui donnent à l'utilisateur une vision des lignes avant et après modification.

Par ailleurs, les triggers peuvent être écrits dans n'importe lequel des langages de routine supportés par PostgreSQL (C, PL/pgSQL, PL/Perl, etc.)

Exemple :

```
ALTER TABLE capitaines ADD COLUMN salaire integer;

CREATE FUNCTION verif_salaire()
RETURNS trigger AS $verif_salaire$
```

```
BEGIN
  -- Nous verifions que les variables ne sont pas vides
  IF NEW.nom IS NULL THEN
    RAISE EXCEPTION 'Le nom ne doit pas être null.';
  END IF;

  IF NEW.salaire IS NULL THEN
    RAISE EXCEPTION 'Le salaire ne doit pas être null.';
  END IF;

  -- pas de baisse de salaires !
  IF NEW.salaire < OLD.salaire THEN
    RAISE EXCEPTION 'Pas de baisse de salaire !';
  END IF;

  RETURN NEW;
END;
$verif_salaire$ LANGUAGE plpgsql;

CREATE TRIGGER verif_salaire BEFORE INSERT OR UPDATE ON capitaines
  FOR EACH ROW EXECUTE PROCEDURE verif_salaire();

UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 3000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';

ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

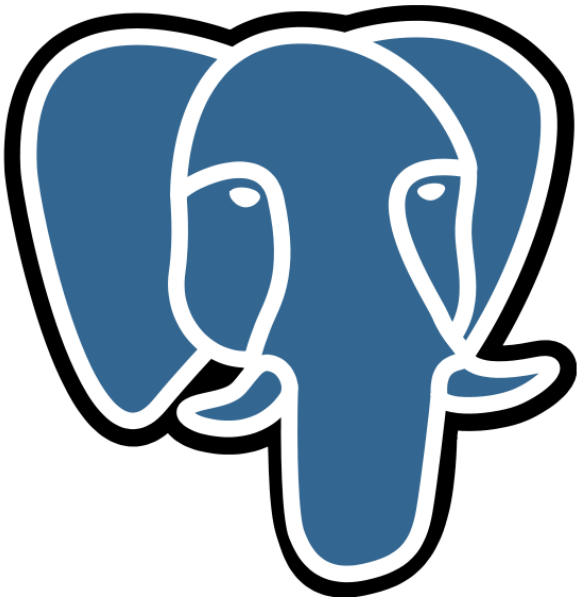
2.3.17 Questions

N'hésitez pas, c'est le moment !

2.4 QUIZ

https://dali.bo/a2_quiz

3/ Installation de PostgreSQL



3.1 INTRODUCTION

- Pré-requis
- Installation depuis les sources
- Installation depuis les binaires
  - installation à partir des paquets
  - installation sous Windows
- Premiers réglages
- Mises à jours

Il existe trois façons d'installer PostgreSQL :

- Les installeurs graphiques :
  - uniquement Windows et macOS ;
  - avantages : installation facile, idéale pour les nouveaux venus ;
  - inconvénients : pas d'intégration avec le système de paquets du système d'exploitation.
- les paquets du système :
  - avantages : meilleure intégration avec les autres logiciels, idéal pour un serveur en production ;
  - inconvénients : aucun ?
- Le code source :
  - avantages : configuration très fine, ajout de patchs, intéressant pour les utilisateurs expérimentés et les testeurs, ou pour embarquer PostgreSQL au sein d'un ensemble de logiciels ;
  - inconvénients : nécessite un environnement de compilation, ainsi que de configurer utilisateurs et script de démarrage.

Nous allons maintenant détailler chaque façon d'installer PostgreSQL.

3.2 PRÉ-REQUIS MINIMAUX POUR UNE INSTANCE POSTGRESQL

- À peu près n'importe quel OS actuel
  - Linux (conseillé)
  - Unix propriétaires (dont macOS)
  - Windows
- N'importe quelle machine
  - …selon les besoins
- Stockage fiable
- Pas d'antivirus !

Il n'existe pas de configuration minimale pour une installation de PostgreSQL. La configuration de base est très conservatrice (128 Mo de cache). PostgreSQL peut fonctionner sur n'importe quelle machine actuelle, sur x86, 32 ou 64 bits. La configuration dépend plutôt des performances et volumétries attendues.

Les plate-formes officiellement supportées[1] incluent les principaux dérivés d'Unix, en premier lieu Linux, mais aussi les FreeBSD, OpenBSD, macOS, Solaris ou AIX ; ainsi que Windows.

Linux est la plate-forme privilégiée par les développeurs, celle disposant du plus d'outils annexes, et est donc recommandée pour faire tourner PostgreSQL.

Debian et Red Hat et leurs dérivés sont les principales distributions vues en production (à côté d'Alpine pour les images docker).

Si vous devez absolument installer un antivirus, il faut impérativement exclure de son analyse tous les répertoires, fichiers et processus de PostgreSQL. L'interaction avec des antivirus a régulièrement mené à des problèmes de performance voire de corruption.

[1] https://www.postgresql.org/docs/current/supported-platforms.html

3.3 INSTALLATION À PARTIR DES SOURCES

Étapes :

- Téléchargement
- Vérification des prérequis
- Compilation
- Installation

Même si les utilisateurs compilent rarement PostgreSQL, c'est l'occasion de voir quelques concepts techniques importants.

Nous allons aborder ici les différentes étapes à réaliser pour installer PostgreSQL à partir des sources :

- trouver les fichiers sources ;
- préparer le serveur pour accueillir PostgreSQL ;
- compiler le serveur ;
- vérifier le résultat de la compilation ;
- installer les fichiers compilés.

3.3.1 Téléchargement

- Disponible depuis postgresql.org[2]
- Télécharger `postgresql-<version>.tar.bz2`

Les fichiers sources et les instructions de compilation sont disponibles sur le site officiel du projet[3] (ou plus directement https://www.postgresql.org/ftp/source/ ou https://ftp.postgresql.org/pub/source). Le nom du fichier à télécharger se présente toujours sous la forme `postgresql-<version>.tar.bz2` où `<version>` représente la version de PostgreSQL (par exemple : https://ftp.postgresql.org/pub/source/v15.4/postgresql-15.4.tar.bz2)

Lorsque la future version du logiciel est en phase de test (versions bêta), les sources sont accessibles à l'adresse suivante : https://www.postgresql.org/developer/beta. (Il existe bien sûr un dépôt git[4].)

[3] https://www.postgresql.org/download/
[4] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary

3.3.2 Phases de compilation/installation

- Processus standard :

```
$ tar xvfj postgresql-<version>.tar.bz2
$ cd postgresql-<version>
$ ./configure         # beaucoup d'options !
$ make
$ sudo make install   # vers /usr/local/pgsql
$ cd contrib
$ make
$ sudo make install   # vers /usr/local/pgsql/…/
```

La compilation de PostgreSQL suit un processus classique.

Comme pour tout programme fourni sous forme d'archive tar, nous commençons par décompacter l'archive dans un répertoire. Le répertoire de destination pourra être celui de l'utilisateur **postgres** (`~`) ou bien dans un répertoire partagé dédié aux sources (`/usr/src/postgres` par exemple) afin de donner l'accès aux sources du programme ainsi qu'à la documentation à tous les utilisateurs du système.

```
cd ~
tar xvfj postgresql-<version>.tar.bz2
```

Une fois l'archive extraite, il faut dans un premier temps lancer le script d'auto-configuration des sources. Les options sont décrites plus bas mais à minima il suffit de ceci :

```
cd postgresql-<version>
./configure
```

Les dernières lignes de la phase de configuration doivent correspondre à la création d'un certain nombre de fichiers, dont notamment le Makefile :

```
configure: creating ./config.status
config.status: creating GNUmakefile
config.status: creating src/Makefile.global
config.status: creating src/include/pg_config.h
config.status: creating src/include/pg_config_ext.h
config.status: creating src/interfaces/ecpg/include/ecpg_config.h
config.status: linking src/backend/port/tas/dummy.s to src/backend/port/tas.s
config.status: linking src/backend/port/posix_sema.c to src/backend/port/pg_sema.c
config.status: linking src/backend/port/sysv_shmem.c to src/backend/port/pg_shmem.c
config.status: linking src/include/port/linux.h to src/include/pg_config_os.h
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
```

Vient ensuite la phase de compilation des sources de PostgreSQL pour en construire les différents exécutables :

```
make
```

Cette phase est la plus longue, mais ne dure que quelques minutes sur du matériel récent, surtout en utilisant une compilation parallélisée grâce à l'option `--jobs`.

Sur certains systèmes, comme Solaris, AIX ou les BSD, la commande `make` issue des outils GNU s'appelle en fait `gmake`. Sous Linux, elle est habituellement renommée en `make`.

Si une erreur s'est produite, il est nécessaire de la corriger avant de continuer. Sinon, on peut installer le résultat de la compilation :

```
sudo make install
```

Cette commande installe les fichiers dans les répertoires spécifiés à l'étape de configuration, notamment via l'option `--prefix`. Sans précision dans l'étape de configuration, les fichiers sont installés dans le répertoire `/usr/local/pgsql`.

Le répertoire `contrib` contient des modules et extensions gérés par le projet, dont l'installation est chaudement conseillée. Leur compilation s'effectue de la même manière.

```
cd contrib
make
sudo make install
```

3.3.3 Options pour ./configure

- Quelques options de configuration notables :
  - `--prefix=` **répertoire**
  - `--with-pgport=` **port**
  - `--with-openssl`
  - `--enable-nls`
  - `--with-perl`, `--with-python`
- Pour les retrouver à postériori :

```
$ pg_config --configure
```

Le script de configuration de la compilation `./configure` possède de nombreux paramètres optionnels, notamment :

- `--prefix=` **répertoire** : permet de définir un répertoire d'installation personnalisé (par défaut, il s'agit de `/usr/local/pgsql`) ;
- `--with-pgport` =**port** : permet de définir un port par défaut différent de 5432 ;
- `--with-openssl` : permet d'activer le support d'OpenSSL pour bénéficier de connexions chiffrées ;
- `--enable-nls` : permet d'activer le support de la langue utilisateur pour les messages provenant du serveur et des applications ;
- `--with-perl`, `--with-python` : permettent d'installer les langages PL correspondants.

On voudra généralement activer ces trois dernières options… et beaucoup d'autres.

En cas de compilation pour la mise à jour d'une version déjà installée, il est important de connaître les options utilisées lors de la précédente compilation. L'outil `pg_config` (un des binaires compilés) le permet ainsi :

```
$ pg_config --configure
 '--build=x86_64-linux-gnu'
 '--prefix=/usr' '--includedir=${prefix}/include'
 '--mandir=${prefix}/share/man' '--infodir=${prefix}/share/info'
 '--sysconfdir=/etc' '--localstatedir=/var'
 '--disable-option-checking' '--disable-silent-rules'
 '--libdir=${prefix}/lib/x86_64-linux-gnu'
 '--runstatedir=/run' '--disable-maintainer-mode'
 '--disable-dependency-tracking'
 '--with-tcl' '--with-perl' '--with-python'
 '--with-pam' '--with-openssl'
 '--with-libxml' '--with-libxslt'
 '--mandir=/usr/share/postgresql/15/man'
 '--docdir=/usr/share/doc/postgresql-doc-15'
 '--sysconfdir=/etc/postgresql-common'
 '--datarootdir=/usr/share/'
 '--datadir=/usr/share/postgresql/15'
 '--bindir=/usr/lib/postgresql/15/bin'
 '--libdir=/usr/lib/x86_64-linux-gnu/'
 '--libexecdir=/usr/lib/postgresql/'
 '--includedir=/usr/include/postgresql/'
 '--with-extra-version= (Debian 15.4-1.pgdg120+1)'
 '--enable-nls'
 '--enable-thread-safety' '--enable-debug' '--enable-dtrace'
 '--disable-rpath'
 '--with-uuid=e2fs' '--with-gnu-ld' '--with-gssapi' '--with-ldap'
 '--with-pgport=5432'
 '--with-system-tzdata=/usr/share/zoneinfo'
 'AWK=mawk' 'MKDIR_P=/bin/mkdir -p' 'PROVE=/usr/bin/prove'
 'PYTHON=/usr/bin/python3' 'TAR=/bin/tar' 'XSLTPROC=xsltproc --nonet'
 'CFLAGS=-g -O2 -fstack-protector-strong -Wformat -Werror=format-security
  ↪  -fno-omit-frame-pointer' 'LDFLAGS=-Wl,-z,relro -Wl,-z,now'
 '--enable-tap-tests'
 '--with-icu'
```

```
 '--with-llvm' 'LLVM_CONFIG=/usr/bin/llvm-config-14'
 'CLANG=/usr/bin/clang-14'
 '--with-lz4' '--with-zstd'
 '--with-systemd' '--with-selinux'
 'build_alias=x86_64-linux-gnu'
 'CPPFLAGS=-Wdate-time -D_FORTIFY_SOURCE=2' 'CXXFLAGS=-g -O2
  ↪  -fstack-protector-strong -Wformat -Werror=format-security'
```

Une version compilée sans option à `./configure` ne renverra rien. Ce qui précède correspond à une version 15.4 compilée sur Debian 12. Les versions des paquets des distributions activent en effet le maximum d'options.

Si PostgreSQL est démarré, la vue système `pg_config` fournit le même résultat :

```
SELECT regexp_split_to_table(setting, ' ') FROM pg_config WHERE name = 'CONFIGURE';
```

3.3.4 Tests de non régression

- Exécution de tests unitaires
- Permet de vérifier l'état des exécutables construits
- Action `check` de la commande `make`

```
$ make check
```

Il est possible d'effectuer des tests avec les exécutables fraîchement construits grâce à la commande suivante :

```
$ make check
[...]
rm -rf ./testtablespace
mkdir ./testtablespace
PATH="/home/dalibo/git.postgresql/tmp_install/usr/local/pgsql/bin:$PATH"
  LD_LIBRARY_PATH="/home/dalibo/git.postgresql/tmp_install/usr/local/pgsql/lib"
  ../../../src/test/regress/pg_regress --temp-instance=./tmp_check --inputdir=.
  --bindir=  --dlpath=. --max-concurrent-tests=20
  --schedule=./parallel_schedule
============== creating temporary instance            ==============
============== initializing database system           ==============
============== starting postmaster                    ==============
running on port 60849 with PID 31852
============== creating database "regression"         ==============
CREATE DATABASE
ALTER DATABASE
============== running regression test queries        ==============
test tablespace                   ... ok          134 ms
parallel group (20 tests):  char varchar text boolean float8 name money pg_lsn
       float4 oid uuid txid bit regproc int2 int8 int4 enum numeric rangetypes
```

```
     boolean                      ... ok           43 ms
     char                         ... ok           25 ms
     name                         ... ok           60 ms
     varchar                      ... ok           25 ms
     text                         ... ok           39 ms
[...]
     partition_join               ... ok          835 ms
     partition_prune              ... ok          793 ms
     reloptions                   ... ok           73 ms
     hash_part                    ... ok           43 ms
     indexing                     ... ok          828 ms
     partition_aggregate          ... ok          799 ms
     partition_info               ... ok          106 ms
     tuplesort                    ... ok         1137 ms
     explain                      ... ok           80 ms
test event_trigger                ... ok           64 ms
test fast_default                 ... ok           89 ms
test stats                        ... ok          571 ms
============== shutting down postmaster               ==============
============== removing temporary instance            ==============

======================
 All 201 tests passed.
======================
[...]
```

Les tests de non régression sont une suite de tests qui vérifient que PostgreSQL fonctionne correctement sur la machine cible. Ces tests ne peuvent pas être exécutés en tant qu'utilisateur **root**. Le fichier `src/test/regress/README` et la documentation contiennent des détails sur l'interprétation des résultats de ces tests.

3.3.5 Création de l'utilisateur

- Jamais **root**
- Utilisateur dédié
  - propriétaire des répertoires et fichiers
  - lancer PostgreSQL
  - traditionnellement : **postgres**
- Variables d'environnement :

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
# Données :
export PGDATA=/usr/local/pgsql/data
```

Le serveur PostgreSQL ne peut pas être exécuté par l'utilisateur **root**. Pour des raisons de sécurité, il est nécessaire de passer par un utilisateur sans droits particuliers. Cet utilisateur sera le seul propriétaire des répertoires et fichiers gérés par le serveur PostgreSQL. Il sera aussi le compte qui permettra de lancer PostgreSQL. Cet utilisateur est généralement appelé **postgres** mais ce n'est pas une obligation.

Une façon de distinguer différentes instances installées sur le même serveur physique ou virtuel est d'utiliser un compte différent par instance, surtout si l'on utilise les variables d'environnement. (Avec un seul compte système, il est facile de nommer les instances avec le paramètre `cluster_name`, dont le contenu apparaîtra dans les noms des processus.)

Il est aussi nécessaire de positionner un certain nombre de variables d'environnement dans `${HOME}/.profile` ou dans `/etc/profile` :

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
export PGDATA=/usr/local/pgsql/data
```

- `/usr/local/pgsql/bin` est le chemin par défaut ou le chemin indiqué à l'option `--prefix` lors de l'étape `configure`. L'ajouter à `PATH` permet de rendre l'exécution de PostgreSQL possible depuis n'importe quel répertoire.
- `LD_LIBRARY_PATH` indique au système où trouver les différentes bibliothèques nécessaires à l'exécution des programmes.
- `MANPATH` indique le chemin de recherche des pages de manuel ;
- `PDGATA` est une spécificité de PostgreSQL : cette variable indique le répertoire des fichiers de données de PostgreSQL, c'est-à-dire les données de l'utilisateur. Plus rigoureusement : elle pointe l'emplacement du `postgresql.conf`, généralement dans ce répertoire (mais `postgresql.conf` peut pointer encore ailleurs).

3.3.6 Création du répertoire de données de l'instance

```
$ initdb -D /usr/local/pgsql/data
```

- Une seule instance !
- Options d'emplacement :
  - `--data` pour les fichiers de données
  - `--waldir` pour les journaux de transactions
- Autres options :
  - `--data-checksums` : sommes de contrôle (conseillé !)
  - et : chemin des journaux, mot de passe, encodage…

Répertoire :

La commande `initdb` doit être exécutée sous le compte de l'utilisateur système PostgreSQL décrit dans la section précédente (généralement **postgres**).

Elle permet de créer les fichiers d'une nouvelle instance avec une première base de données dans le répertoire indiqué.

Ce répertoire ne doit être utilisé que par une seule instance (processus) à la fois !
PostgreSQL vérifie au démarrage qu'aucune autre instance du même serveur n'utilise les fichiers indiqués, mais cette protection n'est pas absolue, notamment avec des accès depuis des systèmes différents. Faites donc bien attention de ne lancer PostgreSQL qu'une seule fois sur un répertoire de données.

Si plusieurs instances cohabitent sur le serveur, elles devront avoir chacune leur répertoire.

Si le répertoire n'existe pas, `initdb` tentera de le créer, s'il a les droits pour le faire. S'il existe, il doit être vide.

Attention : pour des raisons de sécurité et de fiabilité, les répertoires choisis pour les données de votre instance **ne doivent pas** être à la racine d'un point de montage. Que ce soit le répertoire PGDATA, le répertoire `pg_wal` ou les éventuels *tablespaces*. Si un ou plusieurs points de montage sont dédiés à l'utilisation de PostgreSQL, positionnez toujours les données dans un sous-répertoire, voire deux niveaux en-dessous du point de montage, couramment : point de montage/version majeure/nom instance. Exemples :

```
# si /mnt/donnees est le point de montage
/mnt/donnees/15/dbprod
# si /var/lib/postgresql est une partition
# (chemin par défaut du packaging Debian)
/var/lib/postgresql/15/main
# si /var/lib/pgsql est une partition
# (chemin par défaut du packaging Red Hat)
/var/lib/pgsql/15/data
```

À ce propos, voir :

- chapitre *Use of Secondary File Systems*[5] ;
- le détail des raisons techniques[6].

Lancement de initdb :

Voici ce qu'affiche cette commande :

```
$ initdb --data /usr/local/pgsql/data --data-checksums

The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locales
  COLLATE:  en_US.UTF-8
  CTYPE:    en_US.UTF-8
  MESSAGES: en_US.UTF-8
  MONETARY: fr_FR.UTF-8
  NUMERIC:  fr_FR.UTF-8
  TIME:     fr_FR.UTF-8
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

creating directory /usr/local/pgsql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Europe/Paris
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
```

```
initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /usr/local/pgsql/data -l logfile start
```

Avec ces informations, nous pouvons conclure que `initdb` fait les actions suivantes :

- détection de l'utilisateur, de l'encodage et de la locale ;
- création du répertoire PGDATA (`/usr/local/pgsql/data` dans ce cas) ;
- création des sous-répertoires ;
- création et modification des fichiers de configuration ;
- exécution du script bootstrap ;
- synchronisation sur disque ;
- affichage de quelques informations supplémentaires.

Authentification par défaut :

Il est possible de changer la méthode d'authentification par défaut avec les paramètres en ligne de commande `--auth` , `--auth-host` et `--auth-local` . Les options `--pwprompt` ou `--pwfile` permettent d'assigner un mot de passe à l'utilisateur **postgres**.

Sommes de contrôle :

Il est possible d'activer les sommes de contrôle des fichiers de données avec l'option `--data-checksums` . Ces sommes de contrôle sont vérifiées à chaque lecture d'un bloc. Leur activation est chaudement conseillée pour détecter une corruption physique le plus tôt possible. Si votre processeur n'est pas trop ancien et supporte le jeu d'instruction SSE 4.2 (voir dans `/proc/cpuinfo`), il ne devrait pas y avoir d'impact notable sur les performances. Les journaux générés seront cependant plus nombreux. Il est possible de vérifier, activer ou désactiver toutes les sommes de contrôle grâce à l'outil `pg_checksums` (nommé `pg_verify_checksums` en version 11). Cependant, l'opération d'activation sur une instance existante impose un arrêt et une réécriture complète des fichiers. L'opération est même impossible avant PostgreSQL 12. Pensez-y donc dès l'installation.

Emplacement des journaux :

L'option `--waldir` indique l'emplacement des journaux de transaction, par défaut directement dans le PGDATA sous `pg_wal` . Un lien symbolique sera créé vers le répertoire voulu.

Taille des segments :

Les fichiers des journaux de transaction ont une taille par défaut de 16 Mo. Augmenter leur taille avec `--wal-segsize` n'a d'intérêt que pour les très grosses installations générant énormément de journaux pour optimiser leur archivage.

3.3.7 Lancement et arrêt

- Avec le script de l'OS (recommandé) ou `pg_ctl` :

```
systemctl [action] postgresql      # systemd
/etc/init.d/postgresql [action]    # SysV Init
service postgresql [action]        # idem
…

$ pg_ctl --pgdata /usr/local/pgsql/data --log logfile [action]
         --mode [smart|fast|immediate]
```

- `[action]` dépend du besoin :
  - `start` / `stop` / `restart`
  - `reload` pour recharger la configuration
  - `status`
  - `promote`, `logrotate`, `kill`…

(Re)démarrage et arrêt :

La méthode recommandée est d'utiliser un script de démarrage adapté à l'OS, (voir plus bas les outils les commandes `systemd` ou celles propres à Debian) , surtout si l'on a installé PostgreSQL par les paquets. Au besoin, un script d'exemple existe dans le répertoire des sources (`contrib/start-scripts/`) pour les distributions Linux et pour les distributions BSD. Ce script est à exécuter en tant qu'utilisateur **root**.

Sinon, il est possible d'exécuter `pg_ctl` avec l'utilisateur créé précédemment. C'est ce que font au final les commandes système.

Les deux méthodes partagent certaines des actions présentées ci-dessus : `start`, `stop`, `restart` (aux sens évidents), ou `reload` (pour recharger la configuration sans redémarrer PostgreSQL ni couper les connexions).

L'option `--mode` permet de préciser le mode d'arrêt parmi les trois disponibles :

- `smart` : pour vider le cache de PostgreSQL sur disque, interdire de nouvelles connexions et attendre la déconnexion des clients et d'éventuelles sauvegardes ;
- `fast` (par défaut) : pour vider le cache sur disque et déconnecter les clients sans attendre (les transactions en cours sont annulées) ;
- `immediate` : équivalent à un arrêt brutal : tous les processus serveur sont tués et donc, au redémarrage, le serveur devra rejouer ses journaux de transactions.

Rechargement de la configuration :

Pour recharger la configuration après changement du paramétrage, la commande :

```
pg_ctl reload -D /repertoire_pgdata
```

est équivalente à cet ordre SQL :

```
SELECT pg_reload_conf() ;
```

Il faut aussi savoir que quelques paramètres nécessitent un redémarrage de PostgreSQL et non un simple rechargement, ils sont indiqués dans les commentaires de `postgresql.conf`.

3.4 INSTALLATION À PARTIR DES PAQUETS LINUX

- Packages Debian
- Packages RPM

Pour une utilisation en environnement de production, il est généralement préférable d'installer les paquets binaires préparés spécialement pour la distribution utilisée. Les paquets sont préparés par des personnes différentes, suivant les recommandations officielles de la distribution. Il y a donc des différences, parfois importantes, entre les paquets.

3.4.1 Paquets Debian officiels

- Nombreux paquets disponibles :
  - serveur, client, contrib, docs
  - extensions, outils
- `apt install postgresql-<version majeure>`
  - installe les binaires
  - crée l'utilisateur **postgres**
  - exécute `initdb`
  - démarre le serveur

Sur Debian et les versions dérivées (Ubuntu notamment), l'installation de PostgreSQL a été découpée en plusieurs paquets (ici pour la version majeure 15) :

- le serveur : `postgresql-15` ;
- les clients : `postgresql-client-15` ;
- la documentation : `postgresql-doc-15`.

La version majeure dans le nom des paquets (`9.6` , `10` , `12` , `15` …) permet d'installer plusieurs versions majeures sur le même serveur physique ou virtuel.

> Par défaut, sans autre dépôt, une seule version majeure sera disponible dans une version de distribution. Par exemple, `apt install postgresql` sur Debian 12 installera en fait `postgresql-15` (il est en dépendance).

Il existe aussi des paquets pour les outils, les extensions, etc. Certains langages de procédures stockées sont disponibles dans des paquets séparés :

- PL/python dans `postgresql-plpython3-15`
- PL/perl dans `postgresql-plperl-15`
- PL/Tcl dans `postgresql-pltcl-15`
- etc.

Pour compiler des outils liés à PostgreSQL, il est recommandé d'installer également les bibliothèques de développement qui font partie du paquet `postgresql-server-dev-15`.

Quand le paquet `postgresql-15` est installé, plusieurs opérations sont exécutées :

- téléchargement du paquet (dans la dernière version mineure) ;
- installation des binaires contenus dans le paquet ;
- création de l'utilisateur **postgres** (s'il n'existe pas déjà) ;
- paramétrage d'une première instance nommée `main` ;
- création du répertoire des données, lancement de l'instance.

Les exécutables sont installés dans :

```
/usr/lib/postgresql/15/bin
```

Chaque instance porte un nom, qui se retrouve dans le paramètre `cluster_name` et permet d'identifier les processus dans un `ps` ou un `top`. Le nom de la première instance de chaque version majeure est par défaut `main`. Pour cette instance :

- les données sont dans :

```
/var/lib/postgresql/15/main
```

- les fichiers de configuration (pas tous ! certains restent dans le répertoire des données) sont dans :

```
/etc/postgresql/15/main
```

- les traces sont gérées par l'OS sous ce nom :

```
/var/log/postgresql/postgresql-15-main.log
```

- un fichier PID, la socket d'accès local, et l'espace de travail temporaire des statistiques d'activité figurent dans `/var/run/postgresql`.

Tout ceci vise à respecter le plus possible la norme FHS[7] (*Filesystem Hierarchy Standard*).

En cas de mise à jour d'un paquet, le serveur PostgreSQL est redémarré après mise à jour des binaires.

3.4.2 Paquets Debian : spécificités

- Plusieurs versions majeures installables
- Wrappers/scripts pour la gestion des différentes instances :
  - `pg_lsclusters`
  - `pg_ctlcluster`
    - ou : `systemctl stop|start postgresql-15@main`
  - `pg_createcluster`
  - etc.
- Respect de la FHS
- Configuration dans `/etc/postgresql/`

Numéroter les paquets permet d'installer plusieurs versions majeures (mais pas mineures) de PostgreSQL au besoin sur le même système, si les dépôts les contiennent.

Les mainteneurs des paquets Debian ont écrit des scripts pour faciliter la création, la suppression et la gestion de différentes instances sur le même serveur. Les principaux sont :

- `pg_lsclusters` liste les instances ;
- `pg_createcluster <version majeure> <nom instance>` crée une instance de la version majeure et du nom voulu ;
- `pg_dropcluster <version majeure> <nom instance>` détruit l'instance désignée ;
- `/etc/postgresql-common/createcluster.conf` permet de centraliser les paramètres par défaut des instances ;
- la gestion d'une instance est réalisée avec la commande `pg_ctlcluster` :

```
pg_ctlcluster <version majeure> <nom instance> start|stop|reload|status|promote
```

Ce dernier script interagit avec systemd, qui peut être utilisé pour arrêter ou démarrer séparément chaque instance. Ces deux commandes sont équivalentes :

```
sudo pg_ctlcluster 15 main start
sudo systemctl start postgresql@15-main
```

[7] https://fr.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

3.4.3 Paquets Debian communautaires

- La communauté met des paquets Debian à disposition :
  - https://wiki.postgresql.org/wiki/Apt
- Synchrone avec le projet PostgreSQL
- Ajout du dépôt dans `/etc/apt/sources.list.d/pgdg.list`
- Utilisation chaudement conseillée

La distribution Debian préfère des paquets testés et validés, y compris sur des versions assez anciennes, que d'adopter la dernière version dès qu'elle est disponible. Par exemple, Debian 11 ne contiendra jamais que PostgreSQL 13 et ses versions mineures, et Debian 12 ne contiendra que PostgreSQL 15.

Pour faciliter les mises à jour, la communauté PostgreSQL met à disposition son propre dépôt de paquets Debian. Elle en assure le maintien et le support. Les paquets de la communauté ont la même provenance et le même contenu que les paquets officiels Debian, avec d'ailleurs les mêmes mainteneurs. La seule différence est que `apt.postgresql.org` est mis à jour plus fréquemment, en liaison directe avec la communauté PostgreSQL, et contient beaucoup plus d'outils et extensions.

Le wiki[8] indique quelle est la procédure, qui peut se résumer à :

```
sudo apt install -y postgresql-common
sudo /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

```
…
Setting up postgresql-common (248) ...
Creating config file /etc/postgresql-common/createcluster.conf with new version
Building PostgreSQL dictionaries from installed myspell/hunspell packages...
Removing obsolete dictionary files:
Created symlink /etc/systemd/system/multi-user.target.wants/postgresql.service →
 ↪  /lib/systemd/system/postgresql.service.
Processing triggers for man-db (2.11.2-2) ...
This script will enable the PostgreSQL APT repository on apt.postgresql.org on
your system. The distribution codename used will be bookworm-pgdg.
Press Enter to continue, or Ctrl-C to abort.
```

Si l'on continue, ce dépôt sera ajouté au système (ici sous Debian 12) :

```
$ cat /etc/apt/sources.list.d/pgdg.sources
Types: deb
URIs: https://apt.postgresql.org/pub/repos/apt
Suites: bookworm-pgdg
Components: main
Signed-By: /usr/share/postgresql-common/pgdg/apt.postgresql.org.gpg
```

[8] https://wiki.postgresql.org/wiki/Apt

et la version choisie de PostgreSQL sera immédiatement installable :

```
sudo apt install postgresql-14
```

3.4.4 Paquets Red Hat communautaires : yum.postgresql.org

- Préférer les paquets distribués par la communauté :
  - https://yum.postgresql.org/
  - https://yum.postgresql.org/howto/
  - plus complets que les Appstream
- Ajout du dépôt comme paquet RPM

Les versions majeures de Red Hat et de ses dérivés (Rocky Linux, AlmaLinux, Fedora, CentOS, Scientific Linux…) sont très espacées, et la version par défaut de PostgreSQL n'est parfois plus supportée. Même les versions disponibles en AppStream (avec `dnf module`) sont parfois en retard ou ne contiennent que certaines versions majeures. Les dépôts de la communauté sont donc fortement conseillés. Ils contiennent aussi beaucoup plus d'utilitaires, toutes les versions majeures supportées de PostgreSQL simultanément et collent au plus près des versions publiées par la communauté.

Ce dépôt convient pour les dérivés de Red Hat comme Fedora, CentOS, Rocky Linux…

3.4.5 Paquets Red Hat communautaires : installation

```
sudo dnf install -y \
     https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-
 ↪  x86_64/pgdg-redhat-repo-latest.noarch.rpm
sudo dnf -qy module disable postgresql

sudo dnf install -y postgresql15-server
# dont : utilisateur postgres

sudo /usr/pgsql-15/bin/postgresql-15-setup initdb
sudo systemctl enable postgresql-15
sudo systemctl start postgresql-15
```

L'installation de la configuration du dépôt de la communauté est très simple. Les commandes peuvent même être générées en fonction des versions sur https://www.postgresql.org/download/linux/redhat/. L'exemple ci-dessus installe PostgreSQL 15 sur Rocky 9.

3.4.6 Paquets Red Hat communautaires : spécificités

- Paquets séparés serveur, client, contrib
- `/usr/pgsql-XX/bin` : binaires
- Initialisation manuelle (`postgresql-15-setup initdb`)
  - vers : `/var/lib/pgsql/XX/data`
- Gestion par systemd
- Particularités :
  - plusieurs versions majeures installables
  - configuration : dans le répertoire de données

Sous Red Hat et les versions dérivées, les dépôts communautaires ont été découpés en plusieurs paquets, disponibles pour chacune des versions majeures supportées :

- le serveur : `postgresqlXX-server` ;
- les clients : `postgresqlXX` ;
- les modules contrib : `postgresqlXX-contrib` ;
- la documentation : `postgresqlXX-docs` .

où XX est la version majeure (par exemple `11` ou `15`).

Il existe aussi des paquets pour les outils, les extensions, etc. Certains langages de procédures stockées sont disponibles dans des paquets séparés :

- PL/python dans `postgresqlXX-plpython3` ;
- PL/perl dans `postgresqlXX-plperl` ;
- PL/Tcl dans `postgresqlXX-pltcl` ;
- etc.

Pour compiler des outils liés à PostgreSQL, il est recommandé d'installer également les bibliothèques de développement qui font partie du paquet `postgresqlXX-devel` .

Ce nommage sous-entend qu'il est possible d'installer plusieurs versions majeures sur le même serveur physique ou virtuel. Les exécutables sont installés dans le répertoire `/usr/pgsql-XX/bin` , les traces dans `/var/lib/pgsql/XX/data/log` (utilisation du `logger process` de PostgreSQL), les données dans `/var/lib/pgsql/XX/data` . Ce dernier est le répertoire par défaut des données, mais il est possible de le surcharger.

Sur un système type Red Hat sans dépôt communautaire, les noms des paquets ne comportent pas le numéro de version et installent tous les binaires cités ici dans `/usr/bin`.

Quand le paquet serveur est installé, plusieurs opérations sont exécutées : téléchargement du paquet, installation des binaires contenus dans le paquet, et création de l'utilisateur **postgres** (s'il n'existe pas déjà).

Le répertoire des données n'est pas créé. Cela reste une opération à réaliser par la personne qui a installé PostgreSQL sur le serveur. Lancer le script `/usr/psql-XX/bin/postgresqlXX-setup` en tant que **root** :

```
PGSETUP_INITDB_OPTIONS="--data-checksums" \
 /usr/pgsql-15/bin/postgresql-15-setup initdb
```

Plutôt que de respecter la norme FHS (*Filesystem Hierarchy Standard*), les mainteneurs ont fait le choix de respecter l'emplacement des fichiers utilisé par défaut par les développeurs PostgreSQL. La configuration de l'instance (`postgresql.conf` entre autres) est donc directement dans le PGDATA.

Pour installer plusieurs instances, il faudra créer manuellement des services systemd différents.

En cas de mise à jour d'un paquet, le serveur PostgreSQL n'est pas redémarré après mise à jour des binaires.

3.5 UTILISER POSTGRESQL DANS UN CONTENEUR

```
docker run --name pg16 -e POSTGRES_PASSWORD=mdpsuperfort -d postgres

docker exec -it pg16 -U postgres
```

- Image officielle Docker Inc (pas PGDG !) : https://hub.docker.com/_/postgres
- Très pratique pour le développement
- Beaucoup de possibilités avancées avec `docker-compose`
- Ne jamais lancer 2 conteneurs Docker sur un même PGDATA
- Une base de données est-elle faite pour du docker ?
  - supervision système délicate

Les conteneurs comme docker et assimilés (podman, containerd, etc.) permettent d'isoler l'installation de PostgreSQL sur un poste sans avoir à gérer de machine virtuelle. Une fois configuré, le conteneur permet d'avoir un contexte d'exécution de PostgreSQL identique peu importe son support. C'est idéal dans le cas de machines « jetables », par exemple les chaînes de CI/CD.

Exemple :

À titre d'exemple, voici les étapes nécessaires à l'installation d'une instance PostgreSQL sous docker.

D'abord, récupérer l'image[9] de la dernière version de PostgreSQL maintenue par la communauté *PostgreSQL docker* :

```
docker pull postgres
```

La commande permet de créer et de lancer un nouveau conteneur à partir d'une image donnée, ici `postgres:16.0`. Certaines options sont passées en paramètres à `initdb` :

```
docker run \
--network net16 \
--name pg16 \
-p 127.0.0.1:16501:5432 \
--env-file password.env \
-e POSTGRES_INITDB_ARGS='--data-checksums --wal-segsize=1' \
-v '/var/lib/postgresql/docker/16/pg16':'/var/lib/postgresql/data' \
-d postgres:16.0 \
-c 'work_mem=10MB' \
-c 'shared_buffers=256MB' \
-c 'cluster_name=pg16'
```

[9] https://hub.docker.com/_/postgres

Cette commande permet au conteneur d'être attaché à un réseau dédié. Celui-ci doit avoir été créé au préalable avec la commande :

```
docker network create --subnet=192.168.122.0/24 net16
```

L'option `--name` permet de nommer le conteneur pour le rendre plus facilement accessible.

L'option `-p` permet de faire suivre le port 5432 ouvert par le container sur le port 16501 du serveur.

L'option `-d` permet de faire de l'image un démon en arrière-plan.

Les options `-e` définissent des variables d'environnement, moyen systématique de passer des informations au conteneur. Ici l'option est utilisée pour le mot de passe et certaines des options d'`initdb`. On préférera utiliser un fichier `.env` pour `docker compose` ou `docker run --env-file` pour éviter de définir un secret dans la ligne de commande.

L'option `--env-file` permet de passer en paramètre un fichier contenant des variables d'environnement. Ici nous passons un fichier contenant le mot de passe dont le contenu est le suivant :

```
# fichier password.env
POSTGRES_PASSWORD=mdpsuperfort
```

L'option `-v '/var/lib/postgresql/docker/16/pg16':'/var/lib/postgresql/data'` lie un répertoire du disque sur le `PGDATA` du container : ainsi les fichiers de la base survivront à la disparition du conteneur.

Les paramètres de PostgreSQL dans les `-c` sont transmis directement à la ligne de commande du postmaster.

Une fois l'image téléchargée et le conteneur instancié, il est possible d'accéder directement à psql via la commande suivante :

```
docker exec -it pg16 psql -U postgres
```

Ou directement via le serveur hôte s'il dispose de psql :

```
psql -h 127.0.0.1 -p 16501 -U postgres
```

En production, il est recommandé de dédier un serveur à chaque instance PostgreSQL. De ce fait, docker permet de reproduire cette isolation pour des cas d'usage de développement et prototypage où dédier une machine à PostgreSQL est soit trop coûteux, soit trop complexe.

Exemple avec docker compose :

Il est évidement possible de passer par l'outil `docker compose` pour déployer une instance PostgreSQL conteneurisée. Voici la commande `docker run` précédente portée sous `docker compose` dans un fichier YAML.

Fichier `docker-compose.yml`

```
version: '3.3'
```

```
networks:
  net16:
    ipam:
      driver: default
      config:
        - subnet: 192.168.122.0/24

services:
    pg16:
      networks:
          - net16
      container_name: pg16
      ports:
          - '127.0.0.1:16501:5432'
      env_file:
          - password.env
      environment:
          - POSTGRES_INITDB_ARGS=--data-checksums --wal-segsize=1
      volumes:
          - /var/lib/postgresql/docker/16/pg16:/var/lib/postgresql/data'
      image: postgres:16.0
      command: [
      -c, work_mem=10MB,
      -c, shared_buffers=256MB,
      ]
```

La commande `docker compose --file docker-compose.yml up -d` permet la création, ou le lancement, des objets définis dans le fichier `docker-compose.yml`. L'option `-d` permet de lancer les conteneurs en arrière-plan.

```
# Au premier lancement
docker compose --file docker-compose.yml up -d
Creating network "postgresql_net16" with the default driver
Creating pg16 ... done

# Aux lancements suivants
docker compose --file docker-compose.yml up -d
Starting pg16 ... done
```

Pour arrêter, il faut utiliser l'option `stop` :

```
docker compose --file docker-compose.yml stop
Stopping pg16 ... done
```

Limites de l'utilisation de PostgreSQL sous docker :

Ne lancez jamais 2 instances de PostgreSQL sous docker sur le même PGDATA ! Les sécurités habituelles de PostgreSQL ne fonctionnent pas et les deux instances écriront toutes les deux dans les fichiers. La corruption est garantie.

En production, si l'utilisation de PostgreSQL sous docker est de nos jours très fréquente, ce n'est pas

toujours une bonne idée. Une base de données est l'inverse total d'une machine sans état et jetable, et pour les performances, il vaut mieux en première intention gonfler la base (*scale up*) que multiplier les instances (*scale out*), qui sont de toute façon toutes dépendantes de l'instance primaire. Si le choix est fait de fonctionner sous docker, voire Kubernetes, pour des raisons architecturales, la base de données est sans doute le dernier composant à migrer.

De plus, pour les performances, la supervision au niveau système devient très compliquée, et le DBA est encore plus aveugle qu'avec une virtualisation classique. L'ajout d'outillage ou d'extensions non fournies avec PostgreSQL devient un peu plus compliquée (`docker stats` n'est qu'un bon début).

3.6 INSTALLATION SOUS WINDOWS

- Un seul installeur graphique disponible, proposé par EnterpriseDB
- Ou archive des binaires

Le portage de PostgreSQL sous Windows a justifié à lui seul le passage de la branche 7 à la branche 8 du projet. Le système de fichiers NTFS est obligatoire car, contrairement à la VFAT, il gère les liens symboliques (appelés jonctions sous Windows).

L'installateur n'existe plus qu'en version 64 bits depuis PostgreSQL 11.

Étant donné la quantité de travail nécessaire pour le développement et la maintenance de l'installeur graphique, la communauté a abandonné l'installeur graphique qu'elle a proposé un temps. EntrepriseDB a continué de proposer gratuitement le sien, pour la version communautaire comme pour leur version payante. D'autres installateurs ont été proposés par d'autres éditeurs.

Il contient le serveur PostgreSQL avec les modules contrib ainsi que pgAdmin 4, et aussi un outil appelé StackBuilder permettant d'installer d'autres outils comme des pilotes JDBC, ODBC, C#, ou PostGIS.

Pour installer PostgreSQL sans installateur ni compilation, EBD propose aussi une archive des binaires compilés[10].

[10] https://www.enterprisedb.com/download-postgresql-binaries

3.6.1 Installeur graphique

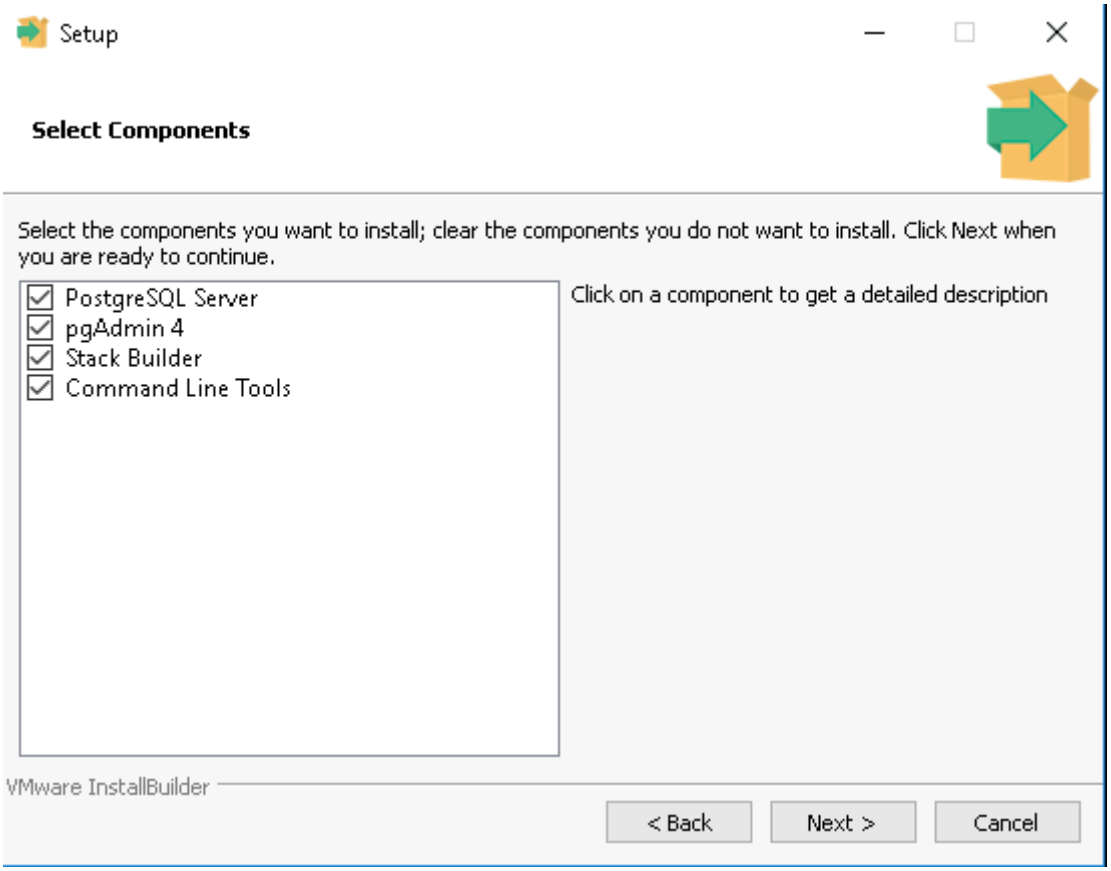


Figure 3/ .1: Installeur graphique - bienvenue

Son utilisation est tout à fait classique. Il y a plusieurs écrans de saisie d'informations :

- le répertoire d'installation des binaires ;
- le choix des outils (copie d'écran ci-dessus), notamment des outils en ligne de commande (à conserver impérativement), des pilotes et de pgAdmin 4 ;
- le répertoire des données de la première instance ;
- le mot de passe de l'utilisateur **postgres** ;
- le numéro de port ;
- la locale par défaut.

Le répertoire d'installation a une valeur par défaut généralement convenable car il n'existe pas vraiment de raison d'installer les binaires PostgreSQL en dehors du répertoire *Program Files*.

Par contre, le répertoire des données de l'instance PostgreSQL. n'a pas à être dans ce même répertoire *Program Files* ! Il est souvent modifié pour un autre disque que le disque système.

Le numéro de port est par défaut le 5432, sauf si d'autres instances sont déjà installées. Dans ce cas, l'installeur propose un numéro de port non utilisé.

Le mot de passe est celui de l'utilisateur **postgres** au sein de PostgreSQL. En cas de mise à jour, il faut saisir l'ancien mot de passe. Le service lui-même et tous ses processus tourneront avec le compte système générique **NETWORK SERVICE** (Compte de service réseau).

La commande `initdb` est exécutée pour créer le répertoire des données. Un service est ajouté pour lancer automatiquement le serveur au démarrage de Windows.

Un sous-menu du menu *Démarrage* contient le nécessaire pour interagir avec le serveur PostgreSQL, comme une icône pour recharger la configuration et surtout pgAdmin 4, qui se lancera dans un navigateur.

L'outil StackBuilder, lancé dans la foulée, permet de télécharger et d'installer d'autres outils : pilotes pour différents langages (Npgsql pour C#, pgJDBC, psqlODBC), Slony-I, PostGIS… installés dans le répertoire de l'utilisateur en cours.

L'installateur peut être utilisé uniquement en ligne de commande (voir les options avec `--help`).

Cet installeur existe aussi sous macOS. Autant il est préférable de passer par les paquets de sa distribution Linux, autant il est recommandé d'utiliser cet installeur sous macOS.

3.7 INDUSTRIALISATION AVEC PGLIFT

- Déploiement et *infrastructure as code* :
  - ligne de commande
  - collections Ansible
- Couverture fonctionnelle :
  - Sauvegardes avec pgBackRest[11]
  - Supervision avec Prometheus[12]
  - Administration avec temBoard[13]
  - Analyse avec PoWA[14]
  - Haute disponibilité avec Patroni[15]
  - Intégration système avec `systemd` ou `rsyslog`

pglift[16] est un outil permettant de déployer et d'exploiter PostgreSQL à grande échelle. Le projet fournit à la fois une interface en ligne de commande pour gérer le cycle de vie des instance et une collection de modules Ansible pour piloter une *infrastructure-as-code* dans un contexte de production.

L'élément fondamental de pglift est l'instance. Celle-ci est constituée d'une instance PostgreSQL et inclut des composants satellites, facultatifs, permettant d'exploiter PostgreSQL à grande échelle. Dans sa version 1.0, pglift supporte les composants suivants :

pgBackRest[17] permet de prendre en charge les sauvegardes physiques PITR (*Point In Time Recovery*) de PostgreSQL.

Prometheus postgres_exporter[18] est un service de supervision permettant de remonter des informations à l'outil de surveillance Prometheus[19].

temBoard[20] est une console web de supervision et d'administration dédiée aux instances PostgreSQL.

PoWA[21] est une console web permettant d'analyser l'activité des instances PostgreSQL en direct.

Patroni[22] est un outil permettant de construire un agrégat d'instances PostgreSQL résilient offrant un service de haute disponibilité.

Tous les composants satellites supportés par pglift sont des logiciels libres. Le projet pglift est lui aussi nativement open source, sous licence GPLv3. Son développement se passe en public sur https://gitl

[16] https://pglift.readthedocs.io/en/latest/
[17] https://pgbackrest.org/
[18] https://github.com/prometheus-community/postgres_exporter
[19] https://prometheus.io/
[20] https://temboard.readthedocs.io/en/latest/
[21] https://powa.readthedocs.io/en/latest/#
[22] https://patroni.readthedocs.io/en/latest

ab.com/dalibo/pglift/ pour l'API Python et l'interface en ligne de commande et sur https://gitlab.com/dalibo/pglift-ansible/ pour la collection Ansible `dalibo.pglift`.

Références :

- Présentation sur le blog Dalibo[23] (Denis Laxalde)
- Documentation[24]

3.7.1 pglift : fichier de configuration

```
prefix: /srv # fichier /etc/pglift/settings.yaml
postgresql:
  auth:
    host: scram-sha-256
prometheus:
  execpath: /usr/bin/prometheus-postgres-exporter
pgbackrest:
  repository:
    mode: path
    path: /srv/pgsql-backups
powa: {}
systemd: {}
rsyslog: {}
```

À coté de PostgreSQL, l'instance inclut un ensemble d'outils nécessaires à son utilisation. L'intégration de ces outils satellites est configurée localement via un fichier YAML `settings.yaml`.

Ce fichier définit comment les différents composants de l'instance sont configurés, installés et exécutés. Il permet de aussi de définir quels composants satellites facultatifs supportés par pgLift sont inclus dans l'instance. Si un élément est listé dans ce fichier sans paramètre associé, il sera exploité dans sa configuration par défaut.

Dans l'exemple ci-dessus, temBoard et Patroni ne sont pas installés, et PoWA est laissé à sa configuration par défaut.

[23] https://blog.dalibo.com/2023/10/17/pglift-intro.html
[24] https://pglift.readthedocs.io/

3.7.2 pglift : exemples de commandes

- Initialisation

```
pglift instance create main --pgbackrest-stanza=main
```

- Modification de configuration

```
pglift pgconf -i main set log_connections=on
```

- Sauvegarde physique

```
pglift instance backup main
```

- Utilisation des outils de l'instance

```
pglift instance exec main -- psql
pglift instance exec main -- pgbackrest info
```

Interface impérative en ligne de commande :

La commande suivante permet la création d'une instance pglift :

```
$ pglift instance create main --pgbackrest-stanza=main
INFO     initializing PostgreSQL
INFO     configuring PostgreSQL authentication
INFO     configuring PostgreSQL
INFO     starting PostgreSQL 16-main
INFO     creating role 'powa'
INFO     creating role 'prometheus'
INFO     creating role 'backup'
INFO     altering role 'backup'
INFO     creating 'powa' database in 16/main
INFO     creating extension 'btree_gist' in database powa
INFO     creating extension 'pg_qualstats' in database powa
INFO     creating extension 'pg_stat_statements' in database powa
INFO     creating extension 'pg_stat_kcache' in database powa
INFO     creating extension 'powa' in database powa
INFO     configuring Prometheus postgres_exporter 16-main
INFO     configuring pgBackRest stanza 'main' for
         pg1-path=/srv/pgsql/16/main/data
INFO     creating pgBackRest stanza main
INFO     starting Prometheus postgres_exporter 16-main
```

L'instance pglift inclut l'instance PostgreSQL ainsi que l'ensemble des modules définis dans le fichier de configuration `settings.yaml`. pglift gère aussi l'intégration au système avec `systemd` ou `rsyslog` comme dans notre exemple. Tout ceci fonctionne sans privilège **root** pour une meilleure séparation des responsabilités et une meilleure sécurité.

pglift permet de récupérer l'état d'une instance à un moment donné :

```
$ pglift instance get main -o json
{
  "name": "main",
  "version": "16",
  "port": 5432,
  "settings": {
    "unix_socket_directories": "/run/user/1000/pglift/postgresql",
    "shared_buffers": "1 GB",
    "wal_level": "replica",
    "archive_mode": true,
    "archive_command": "/usr/bin/pgbackrest --config-path=/etc/pgbackrest \
    --stanza=main --pg1-path=/srv/pgsql/16/main/data archive-push %p",
    "effective_cache_size": "4 GB",
    "log_destination": "syslog",
    "logging_collector": true,
    "log_directory": "/var/log/postgresql",
    "log_filename": "16-main-%Y-%m-%d_%H%M%S.log",
    "syslog_ident": "postgresql-16-main",
    "cluster_name": "main",
    "lc_messages": "C",
    "lc_monetary": "C",
    "lc_numeric": "C",
    "lc_time": "C",
    "shared_preload_libraries": "pg_qualstats, pg_stat_statements, pg_stat_kcache"
  },
  "data_checksums": false,
  "locale": "C",
  "encoding": "UTF8",
  "standby": null,
  "state": "started",
  "pending_restart": false,
  "wal_directory": "/srv/pgsql/16/main/wal",
  "prometheus": {
    "port": 9187
  },
  "data_directory": "/srv/pgsql/16/main/data",
  "powa": {},
  "pgbackrest": {
    "stanza": "main"
  }
}
```

ou de modifier l'instance :

```
# activation du paramètres log_connections
$ pglift pgconf -i main set log_connections=on
INFO     configuring PostgreSQL
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
log_connections: None -> True
# changement du port prometheus
$ pglift instance alter main --prometheus-port 8188
INFO     configuring PostgreSQL
INFO     reconfiguring Prometheus postgres_exporter 16-main
```

```
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
INFO     starting Prometheus postgres_exporter 16-main
$ pglift instance get main # vérification
 name  version  port  data_checksums  locale  encoding  pending_restart  prometheus
  ↪  pgbackrest
 main  16       5432  False           C       UTF8      False            port: 8188
  ↪  stanza: main
```

Les instances et objets PostgreSQL peuvent être manipulés à l'aide des outils *natifs* de PostgreSQL depuis la ligne de commande :

```
$ pglift instance exec main -- pgbench -i bench
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.18 s (drop tables 0.00 s, create tables 0.01 s, ... vacuum 0.04 s, primary
 ↪  keys 0.05 s).
$ pglift instance exec main -- pgbench bench
pgbench (16.0 (Debian 16.0-1.pgdg120+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
number of failed transactions: 0 (0.000%)
latency average = 1.669 ms
initial connection time = 4.544 ms
tps = 599.125277 (without initial connection time)
```

Ceci s'applique aussi à des outils tiers, par exemple avec pgBackRest :

```
$ pglift instance exec main -- pgbackrest info
stanza: main
    status: ok
    cipher: none

    db (current)
        wal archive min/max (16): 000000010000000000000001/000000010000000000000007

        full backup: 20231016-092726F
            timestamp start/stop: 2023-10-16 09:27:26+02 / 2023-10-16 09:27:31+02
            wal start/stop: 000000010000000000000004 / 000000010000000000000004
            database size: 32.0MB, database backup size: 32.0MB
            repo1: backup set size: 4.2MB, backup size: 4.2MB

        diff backup: 20231016-092726F_20231016-092821D
            timestamp start/stop: 2023-10-16 09:28:21+02 / 2023-10-16 09:28:24+02
            wal start/stop: 000000010000000000000007 / 000000010000000000000007
            database size: 54.5MB, database backup size: 22.6MB
```

```
            repo1: backup set size: 6MB, backup size: 1.8MB
            backup reference list: 20231016-092726F
```

Le tutoriel de la ligne de commande[25] de la documentation recense tous les exemples d'utilisation des commandes usuelles de *pglift*.

Interface déclarative avec Ansible :

pglift comporte une collection de modules Ansible, sous l'espace de noms `dalibo.pglift`.Voici un exemple de playbook illustrant ses capacités :

```
- name: Set up database instances
hosts: dbserver
tasks:
  - name: main instance
    dalibo.pglift.instance:
      name: main
      state: started
      port: 5444
      settings:
        max_connections: 100
        shared_buffers: 1GB
        shared_preload_libraries: 'pg_stat_statements, passwordcheck'
      surole_password: ''
      pgbackrest:
        stanza: main
        password: ''
      prometheus:
        password: ''
        port: 9186
      roles:
        - name: admin
          login: true
          password: ''
          connection_limit: 10
          validity: '2025-01-01T00:00'
          in_roles:
            - pg_read_all_stats
            - pg_signal_backend
      databases:
        - name: main
          owner: admin
          settings:
            work_mem: 3MB
          extensions:
            - name: unaccent
              schema: public
```

Le module `dalibo.pglift.instance` permet de gérer l'instance, et les objets reliés comme des rôles ou des bases de données. Les données sensibles peuvent être prises en charge par Ansible vault[26]. Les modules Ansible permettent un contrôle plus important que la ligne de commandes, grâce aux

[25] https://pglift.readthedocs.io/en/latest/tutorials/cli.html
[26] https://docs.ansible.com/ansible/latest/vault_guide/index.html

champs imbriqués, pouvant inclure la configuration de l'instance, des bases de données, des extensions, etc.

Comme tout module Ansible en général, cette interface est complètement déclarative, idempotente et sans état. Ces modules fonctionnent avec d'autres modules Ansible, tels que `community.postgresql`.
Le tutoriel Ansible[27] de la documentation recense davantage d'exemples d'utilisation.

[27] https://pglift.readthedocs.io/en/latest/tutorials/ansible.html

3.8 PREMIERS RÉGLAGES

- Sécurité
- Configuration minimale
- Démarrage
- Test de connexion

3.8.1 Sécurité

- Politique d'accès :
  - pour l'utilisateur **postgres** système
  - pour le rôle **postgres**
- Règles d'accès à l'instance dans `pg_hba.conf`

Selon l'environnement et la politique de sécurité interne à l'entreprise, il faut potentiellement initialiser un mot de passe pour l'utilisateur système **postgres** :

```
$ passwd postgres
```

Sans mot de passe, il faudra passer par un système comme `sudo` pour pouvoir exécuter des commandes en tant qu'utilisateur **postgres**, ce qui sera nécessaire au moins au début.

Le fait de savoir qu'un utilisateur existe sur un serveur permet à un utilisateur hostile de tenter de forcer une connexion par force brute. Par exemple, ce billet de blog[28], montre que l'utilisateur **postgres** est dans le top 10 des logins attaqués.

La meilleure solution pour éviter ce type d'attaque est de ne pas définir de mot de passe pour l'utilisateur OS **postgres** et de se connecter uniquement par des échanges de clés SSH.

Il est conseillé de ne fixer aucun mot de passe pour l'utilisateur système. Il en va de même pour le rôle **postgres** dans l'instance. Une fois connecté au système, nous pourrons utiliser le mode d'authentification local `peer` pour nous connecter au rôle **postgres**. Ce mode permet de limiter la surface d'attaque sur son instance.

En cas de besoin d'accès distant en mode superutilisateur, il sera possible de créer des rôles supplémentaires avec des droits superutilisateur. Ces noms ne doivent pas être facile à deviner par de potentiels attaquants. Il faut donc éviter les rôles **admin** ou **root**.

[28] https://blog.sucuri.net/2013/07/ssh-brute-force-the-10-year-old-attack-that-still-persists.html

Si vous avez besoin de créer des mots de passe, ils doivent bien sûr être longs et complexes (par exemple en les générant avec les utilitaires `pwgen` ou `apg`).

Si vous avez utilisé l'installeur proposé par EnterpriseDB, l'utilisateur système et le rôle PostgreSQL ont déjà un mot de passe, celui demandé par l'installeur. Il n'est donc pas nécessaire de leur en configurer un autre.

Enfin, il est important de vérifier les règles d'accès au serveur contenues dans le fichier `pg_hba.conf`. Ces règles définissent les accès à l'instance en se basant sur plusieurs paramètres : utilisation du réseau ou du socket fichier, en SSL ou non, depuis quel réseau, en utilisant quel rôle, pour quelle base de données et avec quelle méthode d'authentification.

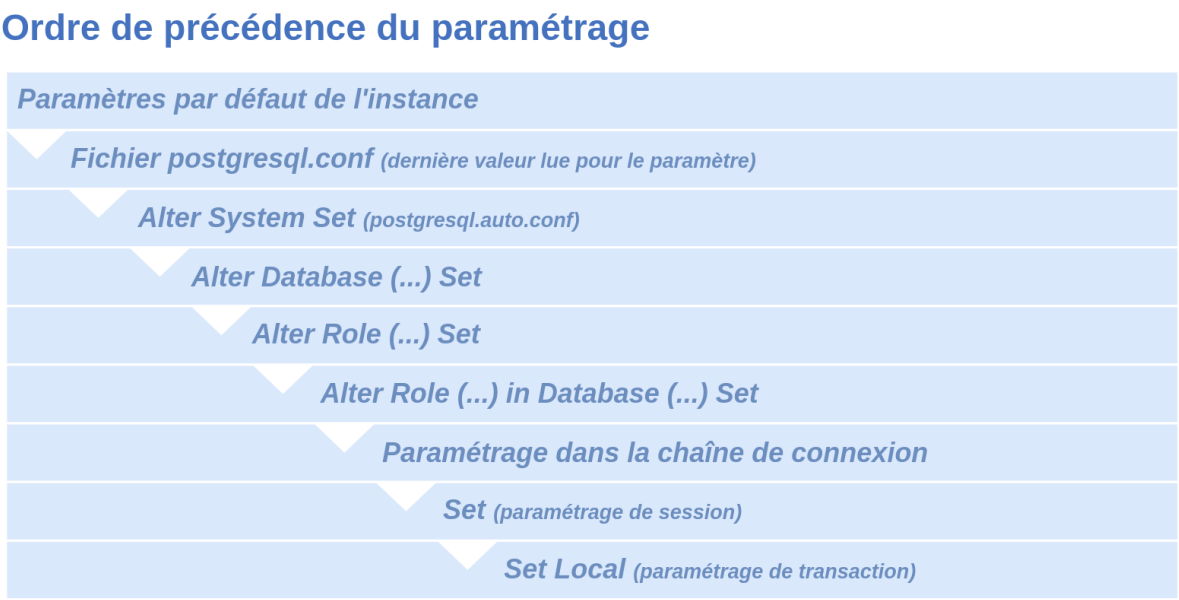
3.8.2 Configuration minimale

- Fichier `postgresql.conf`
- Configuration du moteur
- Plus de 300 paramètres
- Quelques paramètres essentiels

La configuration du moteur se fait via un seul fichier, le fichier `postgresql.conf`. Il se trouve généralement dans le répertoire des données du serveur PostgreSQL. Sous certaines distributions (Debian et affiliés principalement), il est déplacé dans `/etc/postgresql/`.

Ce fichier contient beaucoup de paramètres, plus de 300, mais seuls quelques-uns sont essentiels à connaître pour avoir une instance fiable et performante.

3.8.3 Précédence des paramètres



PostgreSQL offre une certaine granularité dans sa configuration, ainsi certains paramètres peuvent être surchargés par rapport au fichier `postgresql.conf`. Il est utile de connaître l'ordre de précédence. Par exemple, un utilisateur peut spécifier un paramètre dans sa session avec l'ordre `SET`, celui-ci sera prioritaire par rapport à la configuration présente dans le fichier `postgresql.conf`.

3.8.4 Configuration des connexions : accès au serveur

- `listen_addresses = '*'` (systématique)
- `port = 5432`
- `password_encryption` = `scram-sha-256` (v10+)

Ouvrir les accès :

Par défaut, une instance PostgreSQL n'écoute que sur l'interface de boucle locale (`localhost`) et pas sur les autres interfaces réseaux. Pour autoriser les connexions externes à PostgreSQL, il faut modifier le paramètre `listen_addresses`, en général ainsi :

```
listen_addresses = '*'
```

La valeur `*` est un joker indiquant que PostgreSQL doit écouter sur toutes les interfaces réseaux disponibles au moment où il est lancé. Il est aussi possible d'indiquer les interfaces, une à une, en les sé-

parant avec des virgules. Cette méthode est intéressante lorsqu'on veut éviter que l'instance écoute sur une interface donnée. Par prudence il est possible de se limiter aux interfaces destinées à être utilisées :

```
listen_addresses = 'localhost, 10.1.123.123'
```

La restriction par `listen_addresses` est un premier niveau de sécurité. Elle est complémentaire de la méthode plus fine par `pg_hba.conf`, par les IP clientes, utilisateur et base, qu'il faudra de toute façon déployer. De plus, modifier `listen_addresses` impose de redémarrer l'instance.

Port :

Le port par défaut des connexions TCP/IP est le `5432`. C'est la valeur traditionnelle et connue de tous les outils courants.

La modifier n'a d'intérêt que si vous voulez exécuter plusieurs instances PostgreSQL sur le même serveur (physique ou virtuel). En effet, plusieurs instances sur une même machine ne peuvent pas écouter sur le même couple adresse IP et port.

Une instance PostgreSQL n'écoute jamais que sur ce seul port, et tous les clients se connectent dessus. Il n'existe pas de notion de *listener* ou d'outil de redirection comme sur d'autres bases de données concurrentes, du moins sans outil supplémentaire (par exemple le pooler pgBouncer).

S'il y a plusieurs instances dans une même machine, elles devront posséder chacune un couple adresse IP/port unique. En pratique, il vaut mieux attribuer un port par instance. Bien sûr, PostgreSQL refusera de démarrer s'il voit que le port est déjà occupé.

Ne confondez pas la connexion à `localhost` (soit `::1` en IPv6 ou `127.0.0.1` en IPv4), qui utilise les ports TCP/IP, et la connexion dite `local`, passant par les *sockets* de l'OS (par défaut `/var/run/postgresql/.s.PGSQL.5432` sur les distributions les plus courantes). La distinction est importante dans `pg_hba.conf` notamment.

Chiffrage des mots de passe :

À partir de la version 10 et avant la version 14, le paramètre `password_encryption` est à modifier dès l'installation. Il définit l'algorithme de chiffrement utilisé pour le stockage des mots de passe. La valeur `scram-sha-256` permettra d'utiliser la nouvelle norme, plus sécurisée que l'ancien `md5`. Ce n'est plus nécessaire à partir de la version 14 car c'est la valeur par défaut. Avant toute modification, vérifiez quand même que vos outils clients sont compatibles. Au besoin, vous pouvez revenir à `md5` pour un utilisateur donné.

3.8.5 Configuration du nombre de connexions

- `max_connections = 100`
- 1 connexion = 1 processus serveur
- Compromis entre
  - CPU / nombre requêtes actives / RAM / complexité
- Danger si trop haut !
  - performances (même avec des connexions inactives)
  - risque de saturation
- Possibilité de réserver quelques connexions pour l'administration

Le nombre de connexions simultanées est limité par le paramètre `max_connections`. Dès que ce nombre est atteint, les connexions suivantes sont refusées avec un message d'erreur, et ce jusqu'à ce qu'un utilisateur connecté se déconnecte.

`max_connections` vaut par défaut 100, et c'est généralement suffisant en première intention.

Noter qu'il existe un paramètre `superuser_reserved_connections` (à 3 par défaut) qui réserve quelques connexions au superutilisateur pour qu'il puisse se connecter malgré une saturation. Depuis PostgreSQL 16, il existe un autre paramètre nommé `reserved_connections`, (à 0 par défaut) pour réserver quelques connexions aux utilisateurs à qui l'on aura attribué un rôle spécifique, nommé `pg_use_reserved_connections`. Ce peut être utile pour des utilisateurs non applicatifs (supervision et sauvegarde notamment) à qui l'on ne veut ou peut pas donner le rôle `SUPERUSER`.

Il peut être intéressant de diminuer `max_connections` pour interdire d'avoir trop de connexions actives. Cela permet de soulager les entrées-sorties, ou de monter `work_mem` (la mémoire de tri). À l'inverse, il est possible d'augmenter `max_connections` pour qu'un plus grand nombre d'utilisateurs ou d'applications puisse se connecter en même temps.

Au niveau mémoire, un processus consomme par défaut 2 Mo de mémoire vive. Cette consommation peut augmenter suivant son activité.

Il faut surtout savoir qu'à chaque connexion se voit associée un processus sur le serveur, processus qui n'est vraiment actif qu'à l'exécution d'une requête. Il s'agit donc d'arbitrer entre :

- le nombre de requêtes à exécuter à un instant T ;

- le nombre de CPU disponibles ;
- la complexité et la longueur des requêtes ;
- et même le nombre de processus que peut gérer l'OS.

L'établissement a un certain coût également. Il faut éviter qu'une application se connecte et se déconnecte sans cesse.

Il ne sert à rien d'autoriser des milliers de connexions s'il n'y a que quelques processeurs, ou si les requêtes sont lourdes. Si le nombre de requêtes réellement actives augmente fortement, le serveur peut s'effondrer. Restreindre les connexions permet de préserver le serveur, même si certaines connexions sont refusées.

Le paramétrage est compliqué par le fait qu'une même requête peut mobiliser plusieurs processeurs si elle est parallélisée. Certaines requêtes seront limitées par le CPU, d'autres par la bande passante des disques.

Enfin, même si une connexion inactive ne consomme pas de CPU et peu de RAM, elle a tout de même un impact. En effet, une connexion active va générer assez fréquemment ce qu'on appelle un snapshot (ou une image) de l'état des transactions de la base. La durée de création de ce snapshot dépend principalement du nombre de connexions, actives ou non, sur le serveur. Donc une connexion active consommera plus de CPU s'il y a 399 autres connexions, actives ou non, que s'il y a 9 connexions, actives ou non. Ce comportement est partiellement corrigé avec la version 14. Mais il vaut mieux éviter d'avoir des milliers de connexions ouvertes « au cas où ».

Intercaler un « pooler » comme pgBouncer entre les clients et l'instance peut se justifier dans certains cas :

- connexions/déconnexions très fréquentes ;
- centaines, voire milliers, de connexions généralement inactives ;
- limitation du nombre de connexions actives avec mise en attente au niveau du pooler (sans erreur).

3.8.6 Configuration de la mémoire partagée

- `shared_buffers = (?)GB`
  - 25 % de la RAM en première intention
  - max 40 %
  - complémentaire du cache OS

Shared buffers :

Chaque fois que PostgreSQL a besoin de lire ou d'écrire des données, il les charge d'abord dans son cache interne. Ce cache ne sert qu'à ça : stocker des blocs disques qui sont accessibles à tous les

processus PostgreSQL, ce qui permet d'éviter de trop fréquents accès disques car ces accès sont lents. La taille de ce cache dépend d'un paramètre appelé `shared_buffers`.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle[29] donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine dédiée de 32 Go de RAM, cela donne donc :

```
shared_buffers = 8GB
```

Le défaut de 128 Mo n'est donc pas adapté à un serveur sur une machine récente.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Modifier `shared_buffers` impose de redémarrer l'instance.

Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres comme `max_wal_size`, et de s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…).

3.8.7 Configuration : mémoire des processus

- `work_mem`
  - par processus, voire nœud
  - valeur très dépendante de la charge et des requêtes
  - fichiers temporaires vs saturation RAM
- × `hash_mem_multiplier`
- `maintenance_work_mem`
- Pas de limite stricte à la consommation mémoire des sessions
  - Augmenter prudemment & superviser

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Le paramètre le plus important est `work_mem`, qui définit la taille de la mémoire de travail d'un processus lors d'une requête, principalement lors d'opérations de tri : `ORDER BY`, certaines jointures, déduplication… Autre paramètre capital, `maintenance_work_mem` est la mémoire pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, ajouts de clé étrangère…

Cette mémoire est rendue immédiatement après la fin de l'ordre concerné.

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go sur les machines récentes, car il concerne des opérations lourdes rarement exécutées plusieurs fois simultanément. Monter au-delà est rare, mais peut avoir un intérêt dans les créations de très gros index.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit[30], Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

[30] https://dali.bo/j1_html#configuration-du-oom

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[31] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

À partir de PostgreSQL 13, un paramètre multiplicateur peut s'appliquer à certaines opérations particulières (le hachage, lors de jointures ou agrégations). Nommé `hash_mem_multiplier`, il vaut 1 par

[31] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

défaut en versions 13 et 14, et 2 à partir de la 15. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`.

Il existe d'autres paramètres influant sur les besoins en mémoires, moins importants pour une première approche.

3.8.8 Configuration des journaux de transactions 1/2

`fsync` = `on` (si vous tenez à vos données)

À chaque fois qu'une transaction est validée (`COMMIT`), PostgreSQL écrit les modifications qu'elle a générées dans les journaux de transactions.

Afin de garantir la durabilité, PostgreSQL effectue des écritures synchrones des journaux de transaction, donc une écriture physique des données sur le disque. Cela a un coût important sur les performances en écritures s'il y a de nombreuses transactions mais c'est le prix de la sécurité.

Le paramètre `fsync` permet de désactiver l'envoi de l'ordre de synchronisation au système d'exploitation. Ce paramètre **doit** rester à `on` en production. Dans le cas contraire, un arrêt brutal de la machine peut mener à la perte des journaux non encore enregistrés et à la corruption de l'instance. D'autres paramètres et techniques existent pour gagner en performance (et notamment si certaines données peuvent être perdues) sans pour autant risquer de corrompre l'instance.

3.8.9 Configuration des journaux de transactions 2/2

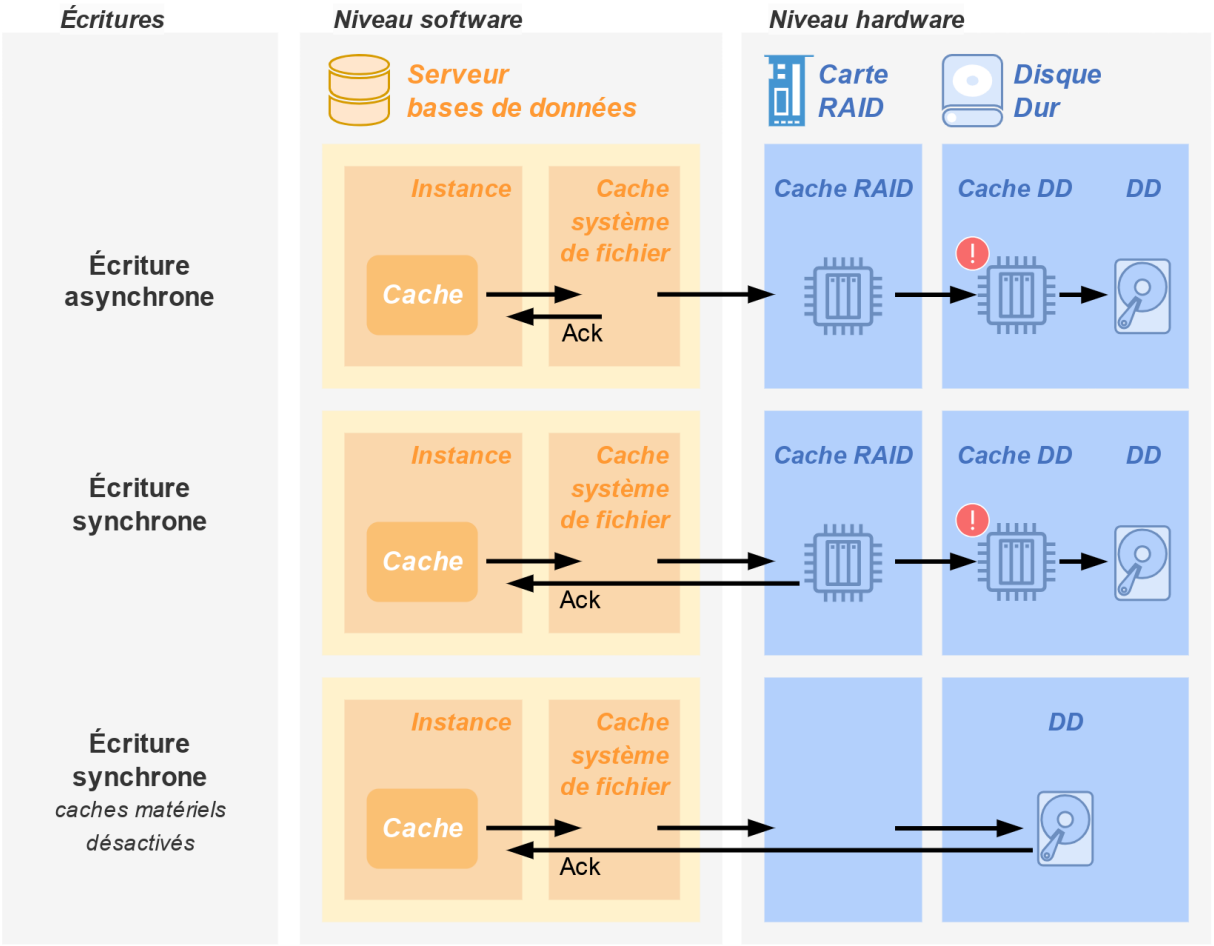


Figure 3/ .2: Niveaux de cache et fsync

Une écriture peut être soit synchrone soit asynchrone. Pour comprendre ce mécanisme, nous allons simplifier le cheminement de l'écriture d'un bloc :

- Dans le cas d'une écriture **asynchrone** : Un processus qui modifie un fichier écrit en fait d'abord dans le cache du système de fichiers du système d'exploitation (OS), cache situé en RAM (mémoire volatile). L'OS confirme tout de suite au processus que l'écriture a été réalisée pour lui rendre la main au plus vite : il y a donc un gain en performance important. Cependant, le bloc ne sera écrit sur disque que plus tard afin notamment de grouper les demandes d'écritures des autres processus, et de réduire les déplacements des têtes de lecture/écriture des disques, qui sont des opérations coûteuses en temps. Entre la confirmation de l'écriture et l'écriture réelle sur les disques, il peut se passer un certain délai : si une panne survient durant celui-ci, les données soi-disant écrites seront perdues, car pas encore physiquement sur le disque.
- Dans le cas d'une écriture **synchrone** : Un processus écrit dans le cache du système

d'exploitation, puis demande explicitement à l'OS d'effectuer la synchronisation (écriture physique) sur disque. Les blocs sont donc écrits sur les disques immédiatement et le processus n'a la confirmation de l'écriture qu'une fois cela fait. Il attendra donc pendant la durée de cette opération, mais il aura la garantie que la donnée est bien présente physiquement sur les disques. Cette synchronisation est très coûteuse et lente (encore plus avec un disque dur classique et ses têtes de disques à déplacer).

Un phénomène équivalent peut se produire à nouveau au niveau matériel (hors du contrôle de l'OS) : pour gagner en performance, les constructeurs ont rajouté un système de cache au sein des cartes RAID. L'OS (et donc le processus qui écrit) a donc confirmation de l'écriture dès que la donnée est présente dans ce cache, alors qu'elle n'est pas encore écrite sur disque. Afin d'éviter la perte de donnée en cas de panne électrique, ce cache est secouru par une batterie qui laissera le temps d'écrire le contenu du cache. Vérifiez qu'elle est bien présente sur vos disques et vos cartes contrôleur RAID.

3.8.10 Configuration des traces

- Selon système/distribution :
  - `log_destination`
  - `logging_collector`
  - emplacement et nom différent pour `postgresql-????.log`
- `log_line_prefix` à compléter :
  - `log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '`
- `lc_messages` = `C` (anglais)

PostgreSQL dispose de plusieurs moyens pour enregistrer les traces : soit il les envoie sur la sortie des erreurs (`stderr`, `csvlog` et `jsonlog`), soit il les envoie à syslog (`syslog`, seulement sous Unix), soit il les envoie au journal des événements (`eventlog`, sous Windows uniquement). Dans le cas où les traces sont envoyées sur la sortie des erreurs, il peut récupérer les messages via un démon appelé *logger process* qui va enregistrer les messages dans des fichiers. Ce démon s'active en configurant le paramètre `logging_collector` à `on`.

Tout cela est configuré par défaut différemment selon le système et la distribution. Red Hat active `logging_collector` et PostgreSQL dépose ses traces dans des fichiers journaliers `$PGDATA/log/postgresql-<jour de la semaine>.log`. Debian utilise `stderr` sans autre paramétrage et c'est le système qui dépose les traces dans `/var/log/postgresql/postgresql-VERSION-nominstance.log`. Les deux variantes fonctionnent. En fonction des habitudes et contraintes locales, il est possible de préférer et d'activer l'une ou l'autre.

L'entête de chaque ligne des traces doit contenir au moins la date et l'heure exacte (`%t` ou `%m` suivant la précision désirée) : des traces sans date et heure ne servent à rien. Des entêtes complets sont suggérés par la documentation de l'analyseur de log pgBadger :

```
log_line_prefix = '%t [%p]: [%l-1] db=%d,user=%u,app=%a,client=%h '
```

Beaucoup d'utilisateurs français récupèrent les traces de PostgreSQL en français. Bien que cela semble une bonne idée au départ, cela se révèle être souvent un problème. Non pas à cause de la qualité de la traduction, mais plutôt parce que les outils de traitement des traces fonctionnent uniquement avec des traces en anglais. Même un outil comme pgBadger, pourtant écrit par un Français, ne sait pas interpréter des traces en français. De plus, la moindre recherche sur Internet ramènera plus de liens si le message est en anglais. Positionnez donc `lc_messages` à `C`.

3.8.11 Configuration des tâches de fond

Laisser ces deux paramètres à `on` :

- `autovacuum`
- `track_counts`

En dehors du *logger process*, PostgreSQL dispose d'autres tâches de fond.

Les processus autovacuum jouent un rôle important pour disposer de bonnes performances : ils empêchent une fragmentation excessive des tables et index, et mettent à jour les statistiques sur les données (statistiques servant à l'optimiseur de requêtes).

La récupération des statistiques sur l'activité permet le bon fonctionnement de l'autovacuum et donne de nombreuses informations importantes à l'administrateur de bases de données.

Ces deux tâches de fond devraient toujours être activés.

3.8.12 Se faciliter la vie

- Création automatique de configuration
  - pgtune et https://pgtune.leopard.in.ua/
  - http://pgconfigurator.cybertec.at/
- Documentation et analyse de configuration
  - https://postgresqlco.nf

pgtune existe en plusieurs versions. La version en ligne de commande va détecter automatiquement le nombre de CPU et la quantité de RAM, alors que la version web nécessitera que ces informations soient saisies. Suivant le type d'utilisation, pgtune proposera une configuration adaptée. Cette configuration n'est évidemment pas forcément optimale par rapport à vos applications, tout simplement parce qu'il ne connaît que les ressources et le type d'utilisation, mais c'est généralement un bon point de départ.

pgconfigurator est un outil plus récent, un peu plus graphique, mais il remplit exactement le même but que pgtune.

Enfin, le site postgresql.co.nf[32] est un peu particulier. C'est en quelque sorte une encyclopédie sur les paramètres de PostgreSQL, mais il est aussi possible de lui faire analyser une configuration. Après analyse, des informations supplémentaires seront affichées pour améliorer cette configuration, que ce soit pour la stabilité du serveur comme pour ses performances.

[32] https://postgresqlco.nf

3.9 MISE À JOUR

- Recommandations
- Mise à jour mineure
- Mise à jour majeure

3.9.1 Recommandations

- Les *Release Notes*
- Intégrité des données
- Bien redémarrer le serveur !

Chaque nouvelle version de PostgreSQL est accompagnée d'une note expliquant les améliorations, les corrections et les innovations apportées par cette version, qu'elle soit majeure ou mineure. Ces notes contiennent toujours une section dédiée aux mises à jour dans laquelle se trouvent des conseils essentiels.

Les *Releases Notes* sont présentes dans l'annexe E de la documentation officielle[33].

Les données de votre instance PostgreSQL sont toujours compatibles d'une version mineure à l'autre. Ainsi, les mises à jour vers une version mineure supérieure peuvent se faire sans migration de données, sauf cas exceptionnel qui serait alors précisé dans les notes de version. Par exemple, de la 15.3 à la 15.4, il a été recommandé de reconstruire les index de type BRIN[34] pour prendre en compte une correction de bug les concernant. Autre exemple : à partir de la 10.3[35], `pg_dump` a imposé des noms d'objets qualifiés pour des raisons de sécurité, ce qui a posé problème pour certains réimports.

Pensez éventuellement à faire une sauvegarde préalable par sécurité.

À contrario, si la mise à jour consiste en un changement de version majeure (par exemple, de la 9.6 à la 16), il est nécessaire de s'assurer que les données seront transférées correctement sans incompatibilité. Là encore, il est important de lire les *Releases Notes* **avant** la mise à jour.

Le site https://why-upgrade.depesz.com/, basé sur les release notes, permet de compiler les différences entre plusieurs versions de PostgreSQL.

Dans tous les cas, pensez à bien redémarrer le serveur. Mettre à jour les binaires ne suffit pas.

[33] https://docs.postgresql.fr/current/release.html
[34] https://www.postgresql.org/docs/release/15.4/
[35] https://www.postgresql.org/docs/release/10.3/

3.9.2 Mise à jour mineure

- Méthode
  - arrêter PostgreSQL
  - mettre à jour les binaires
  - redémarrer PostgreSQL
- Pas besoin de s'occuper des données, sauf cas exceptionnel
  - bien lire les *Release Notes* pour s'en assurer

Faire une mise à jour mineure est simple et rapide.

La première action est de lire les *Release Notes* pour s'assurer qu'il n'y a pas à se préoccuper des données. C'est généralement le cas mais il est préférable de s'en assurer avant qu'il ne soit trop tard.

La deuxième action est de faire la mise à jour. Tout dépend de la façon dont PostgreSQL a été installé :

- par compilation, il suffit de remplacer les anciens binaires par les nouveaux ;
- par paquets précompilés, il suffit d'utiliser le système de paquets (`apt` sur Debian et affiliés, `yum` ou `dnf` sur Red Hat et affiliés) ;
- par l'installeur graphique, en le ré-exécutant.

Ceci fait, un redémarrage du serveur est nécessaire. Il est intéressant de noter que les paquets Debian s'occupent directement de cette opération. Il n'est donc pas nécessaire de le refaire.

3.9.3 Mise à jour majeure

- Bien lire les *Release Notes*
- Bien tester l'application avec la nouvelle version
  - rechercher les régressions en terme de fonctionnalités et de performances
  - penser aux extensions et aux outils
- Pour mettre à jour
  - mise à jour des binaires
  - et mise à jour/traitement des fichiers de données
- 3 méthodes
  - dump/restore
  - réplication logique, externe (Slony) ou interne
  - `pg_upgrade`

Faire une mise à jour majeure est une opération complexe à préparer prudemment.

La première action là-aussi est de lire les *Release Notes* pour bien prendre en compte les régressions potentielles en terme de fonctionnalités et/ou de performances. Cela n'arrive presque jamais mais c'est possible malgré toutes les précautions mises en place.

La deuxième action est de mettre en place un serveur de tests où se trouve la nouvelle version de PostgreSQL avec les données de production. Ce serveur sert à tester PostgreSQL mais aussi, et même surtout, l'application. Le but est de vérifier encore une fois les régressions possibles.

N'oubliez pas de tester les extensions non officielles, voire développées en interne, que vous avez installées. Elles sont souvent moins bien testées.

N'oubliez pas non plus de tester les outils d'administration, de monitoring, de modélisation. Ils nécessitent souvent une mise à jour pour être compatibles avec la nouvelle version installée.

Une fois que les tests sont concluants, arrive le moment de la mise en production. C'est une étape qui peut être longue car les fichiers de données doivent être traités. Il existe plusieurs méthodes que nous détaillerons après.

3.9.4 Mise à jour majeure par dump/restore

- Méthode historique
- Simple et sans risque
  - mais d'autant plus longue que le volume de données est important
- Outils :
  - `pg_dumpall -g` puis `pg_dump`
  - `psql` puis `pg_restore`

Il s'agit de la méthode la plus ancienne et la plus sûre. L'idée est de sauvegarder l'ancienne version avec l'outil de sauvegarde de la nouvelle version. `pg_dumpall` peut suffire, mais `pg_dump` est malgré tout recommandé. Le problème de lenteur vient surtout de la restauration. `pg_restore` est un outil assez lent pour des volumétries importantes. Il convient donc de sélectionner cette solution si le volume de données n'est pas conséquent (pas plus d'une centaine de Go) ou si les autres méthodes ne sont pas possibles. Cependant, il est possible d'accélérer la restauration en utilisant la parallélisation (option `--jobs`). Ceci n'est possible que si la sauvegarde a été faite avec `pg_dump -Fd` ou `-Fc`. Il est à noter que cette sauvegarde peut elle aussi être parallélisée (option `--jobs` là encore).

3.9.5 Mise à jour majeure par Slony

- Nécessite d'utiliser l'outil de réplication Slony
- Permet un retour en arrière immédiat sans perte de données

La méthode Slony est certainement la méthode la plus compliquée. C'est aussi une méthode qui permet un retour arrière vers l'ancienne version sans perte de données.

L'idée est d'installer la nouvelle version de PostgreSQL normalement, sur le même serveur ou sur un autre serveur. Il faut installer Slony sur l'ancienne et la nouvelle instance, et déclarer la réplication de l'ancienne instance vers la nouvelle. Les utilisateurs peuvent continuer à travailler pendant le transfert initial des données. Ils n'auront pas de blocages, tout au plus une perte de performances dues à la lecture et à l'envoi des données vers le nouveau serveur. Une fois le transfert initial réalisé, les données modifiées entre temps sont transférées vers le nouveau serveur.

Une fois arrivé à la synchronisation des deux serveurs, il ne reste plus qu'à déclencher un *switchover*.

La réplication aura lieu ensuite entre le nouveau serveur et l'ancien serveur, ce qui permet un retour en arrière sans perte de données. Une fois acté que le nouveau serveur donne pleine satisfaction, il suffit de désinstaller Slony des deux côtés.

3.9.6 Mise à jour majeure par réplication logique

- Possible entre versions 10 et supérieures
- Remplace Slony, Bucardo…
- Bascule très rapide
- Et retour possible

La réplication logique rend possible une migration entre deux instances de version majeure différente avec une indisponibilité très courte.

La réplication logique n'est disponible en natif qu'à partir de la version 10, la base à migrer doit donc être en version 10 ou supérieure.

Le même principe que les outils de réplication par trigger comme Slony ou Bucardo est utilisé, mais plus simplement et avec les outils du moteur. Le principe est de répliquer une base à l'identique alors que la production tourne. Des clés primaires sur chaque table sont souhaitables mais pas forcément obligatoires.

Lors de la bascule, il suffit d'attendre que les dernières données soient répliquées, ce qui peut être très rapide, et de connecter les applications au nouveau serveur. La réplication peut alors être inversée pour garder l'ancienne production synchrone, permettant de rebasculer dessus en cas de problème sans perdre les données modifiées depuis la bascule.

3.9.7 Mise à jour majeure par pg_upgrade

- `pg_upgrade` : fourni avec PostgreSQL
- Prérequis : pas de changement de format des fichiers entre versions
- Nécessite les deux versions sur le même serveur
- Support des serveurs PostgreSQL à migrer :
  - version minimale 9.2 pour pg_upgrade v15
  - version minimale 8.4 sinon

`pg_upgrade` est certainement l'outil le plus rapide pour une mise à jour majeure.

Il profite du fait que les formats des fichiers de données n'évolue pas, ou de manière rétrocompatible, entre deux versions majeures. Il n'est donc pas nécessaire de tout réécrire.

Grossièrement, son fonctionnement est le suivant. Il récupère la déclaration des objets sur l'ancienne instance avec un `pg_dump` du schéma de chaque base et de chaque objet global. Il intègre la déclaration des objets dans la nouvelle instance. Il fait un ensemble de traitement sur les identifiants d'objets et de transactions. Puis, il copie les fichiers de données de l'ancienne instance vers la nouvelle instance. La copie est l'opération la plus longue, mais comme il n'est pas nécessaire de reconstruire les index et de vérifier les contraintes, cette opération est bien plus rapide que la restauration d'une sauvegarde style `pg_dump`. Pour aller encore plus rapidement, il est possible de créer des liens physiques à la place de la copie des fichiers. Ceci fait, la migration est terminée.

En 2010, Stefan Kaltenbrunner et Bruce Momjian avaient mesuré qu'une base de 150 Go mettait 5 heures à être mise à jour avec la méthode historique (sauvegarde/restauration). Elle mettait 44 minutes en mode copie et 42 secondes en mode lien lors de l'utilisation de `pg_upgrade`.

Vu ses performances, ce serait certainement l'outil à privilégier. Cependant, c'est un outil très complexe et quelques bugs particulièrement méchants ont terni sa réputation. Notre recommandation est de bien tester la mise à jour avant de le faire en production, et uniquement sur des bases suffisamment volumineuses permettant de justifier l'utilisation de cet outil.

Lors du développement de la version 15, les développeurs ont supprimé certaines vieilles parties du code, ce qui le rend à présent incompatible avec des versions très anciennes (de la 8.4 à la 9.1).

3.9.8 Mise à jour de l'OS

Si vous migrez aussi l'OS ou déplacez les fichiers d'une instance :

- compatibilité architecture
- compatibilité librairies
  - réindexation parfois nécessaire
  - ex : Debian 10 et glibc 2.28

Un projet de migration PostgreSQL est souvent l'occasion de mettre à jour le système d'exploitation. Vous pouvez également en profiter pour déplacer l'instance sur un autre serveur à l'OS plus récent en copiant (à froid) le PGDATA.

Il faut bien sûr que l'architecture physique (32/64 bits, *big/little indian*) reste la même. Cependant, même entre deux versions de la même distribution, certains composants du système d'exploitation peuvent avoir une influence, à commencer par la `glibc`. Cette dernière définit l'ordre des caractères, ce qui se retrouve dans les index. Une incompatibilité entre deux versions sur ce point oblige donc à

reconstruire les index, sous peine d'incohérence avec les fonctions de comparaison sur le nouveau système et de corruption à l'écriture.

Daniel Vérité détaille sur son blog[36] le problème pour les mises à jour entre Debian 9 et 10, à cause de la mise à jour de la `glibc`. L'utilisation des collations ICU[37] dans les index contourne le problème mais elles sont encore peu répandues.

Ce problème ne touche bien sûr pas les migrations ou les restaurations avec `pg_dump` / `pg_restore` : les données sont alors transmises de manière logique, indépendamment des caractéristiques physiques des instances source et cible, et les index sont systématiquement reconstruits sur la machine cible.

[36] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html
[37] https://blog-postgresql.verite.pro/2018/07/27/icu_ext.html

3.10 CONCLUSION

- L'installation est simple….
- … mais elle doit être soigneusement préparée
- Préférer les paquets officiels
- Attention aux données lors d'une mise à jour !

3.10.1 Pour aller plus loin

- Documentation officielle, chapitre `Installation`
- Documentation Dalibo, pour l'installation sur Windows

Vous pouvez retrouver la documentation en ligne sur https://docs.postgresql.fr/current/installation.html.

La documentation de Dalibo pour l'installation de PostgreSQL sur Windows est disponible sur https://public.dalibo.com/archives/etudes/installer_postgresql_9.0_sous_windows.pdf.

3.10.2 Questions

N'hésitez pas, c'est le moment !

3.11 QUIZ

https://dali.bo/b_quiz

3.12 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

3.12.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

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

- la version majeure de PostgreSQL (ici la 16) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

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

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 16 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql16-server postgresql16-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql16-llvmjit
```

Création d'une première instance :

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

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb
# cat /var/lib/pgsql/16/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-16/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/16/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-16
# systemctl stop postgresql-16
# systemctl status postgresql-16
# systemctl reload postgresql-16
# systemctl restart postgresql-16
```

Test rapide de bon fonctionnement et connexion à psql :

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

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

```
# systemctl enable postgresql-16
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

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

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

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

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/16/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-16.service \
        /etc/systemd/system/postgresql-16-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/16/infocentre
```

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

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb postgresql-16-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/16/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-16-infocentre
# systemctl [enable|disable] postgresql-16-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

3.12.2 Sur Debian / Ubuntu

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

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

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 16 :

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

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

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

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-16 postgresql-client-16
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/16/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/16/main`
Fichiers de configuration	dans `/etc/postgresql/16/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

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

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 16 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/16/main/start.conf`.

Ouverture du firewall :

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

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

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

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

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

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

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

- créer une instance :

```
# pg_createcluster 16 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 16 infocentre \
  --port=12345 \
  --datadir=/PGDATA/16/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/16/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/16/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 16 infocentre start
```

3.12.3 Accès à l'instance depuis le serveur même (toutes distributions)

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

```
$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=#
```

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

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

```
local   all             postgres                                trust
```

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

```
dalibo:~$ psql -U postgres
psql (16.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (16.0)
Type "help" for help.
postgres=#
```

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

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

Ce fichier doit être protégé des autres utilisateurs :

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

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

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

Rappels :

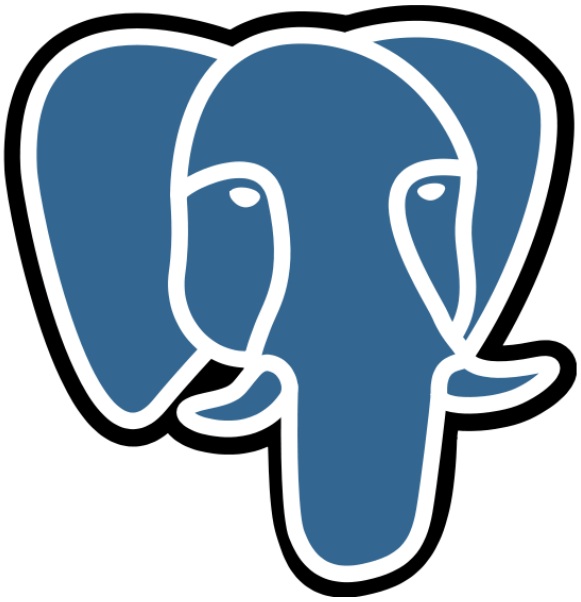
- en cas de problème, consulter les traces (dans `/var/lib/pgsql/16/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-16

root:~# pg_ctlcluster 16 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

4/ Outils graphiques et console



4.1 PRÉAMBULE

Les outils graphiques et console :

- les outils graphiques d'administration
- la console
- les outils de contrôle de l'activité
- les outils DDL
- les outils de maintenance

Ce module nous permet d'approcher le travail au quotidien du DBA et de l'utilisateur de la base de données. Nous verrons les différents outils disponibles, en premier lieu la console texte, `psql`.

4.1.1 Plan

- Outils en ligne de commande de PostgreSQL
- Réaliser des scripts
- Outils graphiques

4.2 OUTILS CONSOLE DE POSTGRESQL

- Plusieurs outils PostgreSQL en ligne de commande existent
  - une console interactive
  - des outils de maintenance
  - des outils de sauvegardes/restauration
  - des outils de gestion des bases

Les outils console de PostgreSQL que nous allons voir sont fournis avec la distribution de PostgreSQL. Ils permettent de tout faire : exécuter des requêtes manuelles, maintenir l'instance, sauvegarder et restaurer les bases.

4.2.1 Outils : Gestion des bases

- `createdb` : ajouter une nouvelle base de données
- `createuser` : ajouter un nouveau compte utilisateur
- `dropdb` : supprimer une base de données
- `dropuser` : supprimer un compte utilisateur

Chacune de ces commandes est un « alias », un raccourci qui permet d'exécuter certaines commandes SQL directement depuis le shell sans se connecter explicitement au serveur. L'option `--echo` permet de voir les ordres SQL envoyés.

Par exemple, la commande système `dropdb` est équivalente à la commande SQL `DROP DATABASE`. L'outil `dropdb` se connecte à la base de données nommée `postgres` et exécute l'ordre SQL et enfin se déconnecte.

La création d'une base se fait en utilisant l'outil `createdb` et en lui indiquant le nom de la nouvelle base, de préférence avec un utilisateur dédié. `createuser` crée ce que l'on appelle un « rôle », et appelle `CREATE ROLE` en SQL. Nous verrons plus loin les droits de connexion, de superutilisateur, etc.

Une création de base depuis le shell peut donc ressembler à ceci :

```
$ createdb --echo --owner erpuser  erp_prod
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE erp_prod OWNER erpuser;
```

Alors qu'une création de rôle peut ressembler à ceci :

```
$ createuser --echo --login --no-superuser erpuser
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE erpuser NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Et si le `pg_hba.conf` le permet :

```
$ psql -U erpuser erp_prod  < script_installation.sql
```

4.2.2 Outils : Sauvegarde / Restauration

- Sauvegarde logique, pour une instance
  - `pg_dumpall` : sauvegarder l'instance PostgreSQL
- Sauvegarde logique, pour une base de données
  - `pg_dump` : sauvegarder une base de données
  - `pg_restore` : restaurer une base de données PostgreSQL
- Sauvegarde physique :
  - `pg_basebackup`
  - `pg_verifybackup`

Ces commandes sont essentielles pour assurer la sécurité des données du serveur.

Comme son nom l'indique, `pg_dumpall` sauvegarde l'instance complète, autrement dit toutes les bases mais aussi les objets globaux. À partir de la version 12, il est cependant possible d'exclure une ou plusieurs bases de cette sauvegarde.

Pour ne sauvegarder qu'une seule base, il est préférable de passer par l'outil `pg_dump`, qui possède plus d'options. Il faut évidemment lui fournir le nom de la base à sauvegarder. Pour sauvegarder notre base b1, il suffit de lancer la commande suivante :

```
$ pg_dump -f b1.sql b1
```

Pour la restauration d'une sauvegarde, l'outil habituel est `pg_restore`. `psql` est utilisé pour la restauration d'une sauvegarde faite en mode texte (script SQL).

Ces deux outils réalisent des sauvegardes logiques, donc au niveau des objets logiques (tables, index, etc).

La sauvegarde physique (donc au niveau des fichiers) à chaud est possible avec `pg_basebackup`, qui copie un serveur en fonctionnement, journaux de transaction inclus. Son fonctionnement est nettement plus complexe qu'un simple `pg_dump`. `pg_basebackup` est utilisé par les outils de sauvegarde

PITR, et pour créer des serveurs secondaires. `pg_verifybackup` permet de vérifier une sauvegarde réalisée avec `pg_basebackup`.

4.2.3 Outils : Maintenance

- Maintenance des bases
  - `vacuumdb` : récupérer l'espace inutilisé, statistiques
  - `clusterdb` : réorganiser une table en fonction d'un index
  - `reindexdb` : réindexer

`reindexdb`, là encore, est un alias lançant des ordres `REINDEX`. Une réindexation périodique des index peut être utile. Par exemple, pour lancer une réindexation de la base **b1** en affichant la commande exécutée :

```
$ reindexdb --echo --concurrently b1
SELECT pg_catalog.set_config('search_path', '', false);
REINDEX DATABASE CONCURRENTLY b1;
WARNING:  cannot reindex system catalogs concurrently, skipping all
```

`vacuumdb` permet d'exécuter les différentes variantes du `VACUUM` (`FULL`, `ANALYZE`, `FREEZE`…) depuis le shell, principalement le nettoyage des lignes mortes, la mise à jour des statistiques sur les données, et la reconstruction de tables. L'usage est ponctuel, le démon autovacuum s'occupant de cela en temps normal.

`clusterdb` lance un ordre `CLUSTER`, soit une reconstruction de la table avec tri selon un index. L'usage est très spécifique.

Rappelons que ces opérations posent des verrous qui peuvent être très gênants sur une base active.

4.2.4 Outils : Maintenance de l'instance

- `initdb` : création d'instance
- `pg_ctl` : lancer, arrêter, relancer, promouvoir l'instance
- `pg_upgrade` : migrations majeures
- `pg_config`, `pg_controldata` : configuration

Ces outils sont rarement utilisés directement, car on passe généralement par les outils du système d'exploitation et ceux fournis par les paquets, qui les utilisent. Ils peuvent toutefois servir et il faut les connaître.

`initdb` crée une instance, c'est-à-dire crée tous les fichiers nécessaires dans le répertoire indiqué (`PGDATA`). Les options permettent d'affecter certains paramètres par défaut. La plus importante (car on ne peut corriger plus tard qu'à condition que l'instance soit arrêtée, donc en arrêt de production) est l'option `--data-checksums` activant les sommes de contrôle, dont l'activation est généralement conseillée.

`pg_ctl` est généralement utilisé pour démarrer/arrêter une instance, pour recharger les fichiers de configuration après modification, ou pour promouvoir une instance secondaire en primaire. Toutes les actions possibles sont documentées ici[1].

`pg_upgrade` est utilisée pour convertir une instance existante lors d'une migration entre versions majeures. La version minimale supportée est la 8.4, sauf à partir de pg_upgrade 15 (version 9.2 dans ce cas).

`pg_config` fournit des informations techniques sur les binaires installés (chemins notamment).

`pg_controldata` fournit des informations techniques de base sur une instance.

4.2.5 Autres outils en ligne de commande

- `pgbench` pour des tests
- Outils liés à la réplication/sauvegarde physique, aux tests, analyse…

`pgbench` est l'outil de base pour tester la charge et l'influence de paramètres. Créez les tables de travail avec l'option `-i`, fixez la volumétrie avec `-s`, et lancez `pgbench` en précisant le nombre de clients, de transactions… L'outil vous calcule le nombre de transactions par secondes et diverses informations statistiques. Les requêtes utilisées sont basiques mais vous pouvez fournir les vôtres.

D'autres outils sont liés à l'archivage (`pg_receivewal`) et/ou à la réplication par *log shipping* (`pg_archivecleanup`) ou logique (`pg_recvlogical`), au sauvetage d'instances secondaires (`pg_rewind`), à la vérification de la disponibilité (`pg_isready`), à des tests de la configuration matérielle (`pg_test_fsync`, `pg_test_timing`), ou d'intégrité (`pg_checksums`), à l'analyse (`pg_waldump`).

[1] https://www.postgresql.org/docs/current/app-pg-ctl.html

4.3 CHAÎNES DE CONNEXION

Pour se connecter à une base :

- paramètres propres aux outils
- via la libpq
  - variables d'environnement
  - par chaînes clés/valeur
  - par chaînes URI
  - idem en Python,PHP,Perl
- JDBC/.NET/ODBC ont des syntaxes spécifiques

Les types de connexion connus de PostgreSQL et de sa librairie cliente (libpq) sont, au choix, les paramètres explicites (assez variables suivants les outils), les variables d'environnement, les chaînes clés/valeur, et les URI (`postgresql://…`).

Les pilotes PHP[2]…), Perl[3], ou Python[4] utilisent la libpq, comme bien sûr les outils du projet PostgreSQL, et ce qui suit est directement utilisable.

Nous ne traiterons pas ici des syntaxes propres aux outils n'utilisant pas la libpq, comme les pilotes JDBC[5] (dont les URI sont proches mais différente de celles décrites plus bas) ou .NET[6], ou encore les différents pilotes ODBC (comme psqlODBC[7], très limité).

4.3.1 Paramètres

Outils habituels, et très souvent :

```
$ psql -h serveur -d mabase -U nom -p 5432
```

Option	Variable	Valeur par défaut
`-h` HÔTE	$PGHOST	`/tmp` , `/var/run/postgresql`

[2] https://www.php.net/manual/en/function.pg-connect.php
[3] https://metacpan.org/pod/DBD::Pg#data_sources
[4] https://www.psycopg.org/docs/module.html#psycopg2.connect
[5] https://jdbc.postgresql.org/documentation/head/connect.html
[6] https://www.npgsql.org/doc/connection-string-parameters.html
[7] https://odbc.postgresql.org/

Option	Variable	Valeur par défaut
`-p` PORT	$PGPORT	`5432`
`-U` NOM	$PGUSER	nom de l'utilisateur OS
`-d` base	$PGDATABASE	nom de l'utilisateur PG
	$PGOPTIONS	options de connexions

Les options de connexion permettent d'indiquer comment trouver l'instance (serveur, port), puis d'indiquer l'utilisateur et la base de données concernés parmi les différentes de l'instance. Ces deux derniers champs doivent passer le filtre du `pg_hba.conf` du serveur pour que la connexion réussisse.

Lorsque l'une de ces options n'est pas précisée, la bibliothèque cliente PostgreSQL cherche une variable d'environnement correspondante et prend sa valeur. Si elle ne trouve pas de variable, elle se rabat sur une valeur par défaut.

Les paramètres et variables d'environnement qui suivent sont utilisés par les outils du projet, et de nombreux autres outils de la communauté.

La problématique du mot de passe est laissée de côté pour le moment.

Hôte :

Le paramètre `-h <hôte>` ou la variable d'environnement `$PGHOST` permettent de préciser le nom ou l'adresse IP de la machine qui héberge l'instance.

Sans précision, sur Unix, le client se connecte sur la socket Unix, généralement dans `/var/run/postgresql` (défaut sur Debian et Red Hat) ou `/tmp` (défaut de la version compilée). Le réseau n'est alors pas utilisé, et il y a donc une différence entre `-h localhost` (via ::1 ou 127.0.0.1 donc) et `-h /var/run/postgresql` (défaut), ce qui peut avoir un résultat différent selon la configuration du `pg_hba.conf`. Par défaut, l'accès par le réseau exige un mot de passe.

Sous Windows, le comportement par défaut est de se connecter à **localhost**.

Serveur :

`-p <port>` ou `$PGPORT` permettent de préciser le port sur lequel l'instance écoute les connexions entrantes. Sans indication, le port par défaut est le 5432.

Utilisateur :

`-U <nom>` ou `$PGUSER` permettent de préciser le nom de l'utilisateur, connu de PostgreSQL, qui doit avoir été créé préalablement sur l'instance.

Sans indication, le nom d'utilisateur PostgreSQL est le nom de l'utilisateur système connecté.

Base de données :

`-d base` ou `$PGDATABASE` permettent de préciser le nom de la base de données utilisée pour la connexion.

Sans précision, le nom de la base de données de connexion sera celui de l'utilisateur PostgreSQL (qui peut découler de l'utilisateur connecté au système).

Exemples :

- Chaîne de connexion classique, implicitement au port 5432 en local sur le serveur :

```
$ psql -U jeanpierre -d comptabilite
```

- Connexion à un serveur distant pour une sauvegarde :

```
$ pg_dump -h serveur3 -p 5435 -U postgres -d basecritique -f fichier.dump
```

- Connexion sans paramètre via l'utilisateur système **postgres**, et donc implicitement en tant qu'utilisateur **postgres** de l'instance à la base **postgres** (qui existent généralement par défaut).

```
$ sudo -iu postgres  psql
```

Dans les configurations par défaut courantes, cette commande est généralement la seule qui fonctionne sur une instance fraîchement installée.

- Utilisation des variables d'environnement pour alléger la syntaxe dans un petit script de maintenance :

```
#! /bin/bash
export PGHOST=/var/run/postgresql
export PGPORT=5435
export PGDATABASE=comptabilite
export PGUSER=superutilisateur
# liste des bases
psql -l
# nettoyage
vacuumdb
# une sauvegarde
pg_dump -f fichier.dump
```

Raccourcis

Généralement, préciser `-d` n'est pas nécessaire quand la base de données est le premier argument non nommé de la ligne de commande. Par exemple :

```
$ psql -U jeanpierre comptabilite
```

```
$ sudo -iu postgres  vacuumdb nomdelabase
```

Il faut faire attention à quelques différences entre les outils : pour `pgbench`, `-d` désigne le mode debug et le nom de la base est un argument non nommé ; alors que pour `pg_restore`, le `-d` est nécessaire pour restaurer vers une base.

Variable d'environnement PGOPTIONS

La variable d'environnement `$PGOPTIONS` permet de positionner la plupart des paramètres de sessions disponibles, qui surchargent les valeurs par défaut.

Par exemple, pour exécuter une requête avec un paramétrage différent de `work_mem` (mémoire de tri) :

```
$ PGOPTIONS="-c work_mem=100MB" psql -p 5433 -h serveur nombase  < grosse_requete.sql
```

ou pour importer une sauvegarde sans être freiné par un serveur secondaire synchrone :

```
$ PGOPTIONS="-c synchronous_commit=local" pg_restore -d nombase sauvegarde.dump
```

4.3.2 Autres variables d'environnement

- $PGAPPNAME
- $PGSSLMODE
- $PGPASSWORD
- …

Il existe aussi :

- `$PGSSLMODE` pour définir le niveau de chiffrement SSL de la connexion avec les valeurs `disable`, `prefer` (le défaut), `require`… (il existe d'autres variables d'environnement pour une gestion fine du SSL) ;
- `$PGAPPNAME` permet de définir une chaîne de caractère arbitraire pour identifier la connexion dans les outils et vues d'administration (paramètre de session `application_name`) : mettez-y par exemple le nom du script ;
- `$PGPASSWORD` peut stocker le mot de passe pour ne pas avoir à l'entrer à la connexion (voir plus loin).

Toutes ces variables d'environnement, ainsi que de nombreuses autres, et leurs diverses valeurs possibles, sont documentées[8].

4.3.3 Chaînes libpq clés/valeur

```
psql "host=serveur1  user=jeanpierre  dbname=comptabilite"

psql -d "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
     sslmode=require  application_name='chargement'
     options='-c work_mem=30MB' "
```

En lieu du nom de la base, une chaîne peut inclure tous les paramètres nécessaires. Cette syntaxe permet de fournir plusieurs paramètres d'un coup.

[8] https://docs.postgresql.fr/current/libpq-envars.html

Les paramètres disponibles sont listés dans la documentation[9]. On retrouvera de nombreux paramètres équivalents aux variables d'environnement[10], ainsi que d'autres.

Dans cet exemple, on se connecte en exigeant le SSL, en positionnant `application_name` pour mieux repérer la session, et en modifiant la mémoire de tri, ainsi que le paramétrage de la synchronisation sur disque pour accélérer les choses :

```
$ psql "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
       sslmode=require  application_name='chargement'
       options='-c work_mem=99MB' -c synchronous_commit=off' " \
       <  script_chargement.sql
```

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

4.3.4 Chaînes URI

```
psql -d "postgresql://jeanpierre@serveur1:5432/comptabilite"

psql \
"postgres://jeanpierre@serveur1/comptabilite?sslmode=require\
&options=-c%20synchronous_commit%3Doff"

psql -d postgresql://serveur1/comptabilite?user=jeanpierre\&port=5432
```

Une autre possibilité existe : des chaînes sous forme URI comme il en existe pour beaucoup de pilotes et d'outils. La syntaxe est de la forme :

```
postgresql://[user[:motdepasse]@][lieu][:port][/dbname][?param1=valeur21&param2=valeur2…]
```

Là encore cette chaîne remplace le nom de la base dans les outils habituels. `postgres://` et `postgresql://` sont tous deux acceptés.

Cette syntaxe est très souple. Une difficulté réside dans la gestion des caractères spéciaux, signes `=` et des espaces :

```
# Ces deux appels sont équivalents
$ psql -d postgresql:///var/lib/postgresql?dbname=pgbench\&user=pgbench\&port=5436
$ psql -h /var/lib/postgresql -d pgbench -U pgbench -p 5436

# Ces deux appels sont équivalents
$ psql "postgresql://bob@vm1/proddb?&options=-c%20synchronous_commit%3Doff"
$ psql -U bob -h vm1 -d proddb -c 'synchronous_commit=off'
```

[9] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-PARAMKEYWORDS
[10] https://docs.postgresql.fr/current/libpq-envars.html

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

4.3.5 Connexion avec choix automatique du serveur

```
psql "host=serveur1,serveur2,serveur3
      port=5432,5433,5434
      user=jeanpierre  dbname=comptabilite
      target_session_attrs=read-write
      load_balance_hosts=random"   # v16
```

Il est possible d'indiquer plusieurs hôtes et ports. Jusque PostgreSQL 15 inclus, l'hôte sélectionné est le premier qui répond avec les conditions demandées. À partir de PostgreSQL 16, on peut ajouter `load_balance_hosts=random` pour que la libpq choisisse un serveur au hasard, pour une répartition de charge basique, mais très simple à mettre en place.

Si l'authentification ne passe pas, la connexion tombera en erreur. Il est possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule grâce au paramètre `target_session_attrs`.

Par exemple, on se connectera ainsi au premier serveur de la liste ouvert en écriture et disponible parmi les trois précisés :

```
psql postgresql://jeanpierre@serveur1:5432,serveur2:5433,serveur3:5434/\
comptabilite?target_session_attrs=read-write
```

qui équivaut à :

```
psql "host=serveur1,serveur2,serveur3 port=5432,5433,5434
      target_session_attrs=read-write  user=jeanpierre  dbname=comptabilite"
```

Depuis la version 14, dans le but de faciliter la connexion aux différents serveurs d'une grappe, `target_session_attrs` possède d'autres options[11] que `read-write`, à savoir : `any`, `read-only`, `primary`, `standby`, `prefer-standby`. Par exemple, la commande suivante permet de se connecter à un secondaire au hasard parmi ceux qui répondent. `serveur3` est doublé pour prendre une charge double des autres. S'il ne reste plus que le primaire, la connexion se fera à celui-ci.

```
psql "host=serveur1,serveur2,serveur3,serveur3 port=5432,5433,5434,5434
      target_session_attrs=prefer-standby
      load_balance_hosts=random
      user=jeanpierre  dbname=comptabilite"
```

[11] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-CONNECT-TARGET-SESSION-ATTRS

4.3.6 Authentification d'un client (outils console)

- En interactif (`psql`)
  - `-W | --password`
  - `-w | --no-password`
- Variable `$PGPASSWORD`
- À préférer : fichier `.pgpass`
  - `chmod 600 .pgpass`
  - `nom_hote:port:database:nomutilisateur:motdepasse`

Options -W et -w de psql

L'option `-W` oblige à saisir le mot de passe de l'utilisateur. C'est le comportement par défaut si le serveur demande un mot de passe. Si les accès aux serveurs sont configurés sans mot de passe et que cette option est utilisée, le mot de passe sera demandé et fourni à PostgreSQL lors de la demande de connexion. Mais PostgreSQL ne vérifiera pas s'il est bon si la méthode d'authentification ne le réclame pas.

L'option `-w` empêche la saisie d'un mot de passe. Si le serveur a une méthode d'authentification qui nécessite un mot de passe, ce dernier devra être fourni par le fichier `.pgpass` ou par la variable d'environnement `$PGPASSWORD`. Dans tous les autres cas, la connexion échoue.

Variable $PGPASSWORD

Si `psql` détecte une variable `$PGPASSWORD` initialisée, il se servira de son contenu comme mot de passe qui sera soumis pour l'authentification du client.

Fichier .pgpass

Le fichier `.pgpass`, situé dans le répertoire personnel de l'utilisateur ou celui référencé par `$PGPASSFILE`, est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\pgpass.conf` (où `%APPDATA%` fait référence au sous-répertoire `Application Data` du profil de l'utilisateur).

Ce fichier devra être composé de lignes du format :

```
nom_hote:port:nom_base:nom_utilisateur:mot_de_passe
```

Chacun des quatre premiers champs pourraient être une valeur littérale ou `*` (qui correspond à tout). La première ligne réalisant une correspondance pour les paramètres de connexion sera utilisée (du

coup, placez les entrées plus spécifiques en premier lorsque vous utilisez des jokers). Si une entrée a besoin de contenir `*` ou `\`, échappez ce caractère avec `\`. Un nom d'hôte `localhost` correspond à la fois aux connexions `host` (`TCP`) et aux connexions `local` (`socket` de domaine *Unix*) provenant de la machine locale.

Les droits sur `.pgpass` doivent interdire l'accès aux autres et au groupe ; réalisez ceci avec la commande :

```
chmod 0600 ~/.pgpass
```

Attention : si les droits du fichier sont moins stricts, le fichier sera ignoré !

Les droits du fichier ne sont actuellement pas vérifiés sur Microsoft Windows.

4.4 LA CONSOLE PSQL

- Un outil simple pour
  - les opérations courantes
  - les tâches de maintenance
  - l'exécution des scripts
  - les tests

```
postgres$ psql
  base=#
```

La console `psql` permet d'effectuer l'ensemble des tâches courantes d'un utilisateur de bases de données. Si ces tâches peuvent souvent être effectuées à l'aide d'un outil graphique, la console présente l'avantage de pouvoir être utilisée en l'absence d'environnement graphique ou de scripter les opérations à effectuer sur la base de données. Elle a la garantie d'être également toujours disponible.

Nous verrons également qu'il est possible d'administrer la base de données depuis cette console.

Enfin, elle permet de tester l'activité du serveur, l'existence d'une base, la présence d'un langage de programmation…

4.4.1 Obtenir de l'aide et quitter

- Obtenir de l'aide sur les commandes internes `psql`
  - `\?`
- Obtenir de l'aide sur les ordres SQL
  - `\h <COMMANDE>`
- Quitter
  - `\q` ou `ctrl-D`
  - `quit` ou `exit` (v11)

`\?` liste les commandes propres à `psql` (par exemple `\d` ou `\du` pour voir les tables ou les utilisateurs), trop nombreuses pour pouvoir être mémorisées.

`\h <COMMANDE>` affiche l'aide en ligne des commandes SQL. Sans argument, la liste des commandes disponibles est affichée. La version 12 ajoute en plus l'URL vers la page web documentant cette commande.

Exemple :

```
postgres=# \h ALTER TA

Commande :    ALTER TABLE
Description : modifier la définition d'une table
Syntaxe :
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
…
URL: https://www.postgresql.org/docs/15/sql-altertable.html
```

`\q` ou `Ctrl-D` permettent de quitter `psql`. Depuis la version 11, il est aussi possible d'utiliser `quit` ou `exit`.

4.4.2 Gestion de la connexion

- Modifier le mot de passe d'un utilisateur
  - `\password  nomutilisateur`
- Quelle est la connexion courante ?
  - `\conninfo`
  - `SELECT current_user,session_user,system_user;`
- Se connecter à une autre base :
  - `\c ma base`
  - `\c mabase  utilisateur serveur 5432`

`\c` permet de changer d'utilisateur et/ou de base de données sans quitter le client. `\conninfo` permet de savoir où l'on est connecté.

Exemple :

```
CREATE DATABASE formation;
CREATE DATABASE prod;
CREATE USER stagiaire1;
CREATE USER stagiaire2;
CREATE USER admin;
```

```
postgres=# \c formation stagiaire1

You are now connected to database "formation" as user "stagiaire1".

formation=> \c - stagiaire2

You are now connected to database "formation" as user "stagiaire2".

formation=> \c prod admin

You are now connected to database "prod" as user "admin".

prod=> \conninfo

You are connected to database "prod" as user "admin"
                  on host "localhost" (address "::1") at port "5412".
```

Au niveau SQL, un équivalent partiel de `\conninfo` serait :

```
SELECT  current_user,
        current_catalog,                          -- base
        inet_server_addr(), inet_server_port(),   -- serveur
        system_user                               -- depuis PG16
        \gx

-[ RECORD 1 ]----+-----------------------
current_user     | dalibo
current_catalog  | pgbench
inet_server_addr | 192.168.74.5
inet_server_port | 5433
system_user      | scram-sha-256:postgres
```

Un superutilisateur pourra affecter un mot de passe à un autre utilisateur grâce à la commande `\password`, qui en fait encapsule un `ALTER USER … PASSWORD …`. Le gros intérêt de `\password` est d'envoyer le mot de passe chiffré au serveur. Ainsi, même si les traces contiennent toutes les requêtes SQL exécutées, il est impossible de retrouver les mots de passe via le fichier de traces. Ce n'est pas le cas avec un `CREATE ROLE` ou un `ALTER ROLE` manuel (à moins de chiffrer soi-même le mot de passe).

4.4.3 Catalogue système : objets utilisateurs

Lister :

- les bases de données
  - `\l` , `\l+`
- les tables
  - `\d` , `\d+` , `\dt` , `\dt+`
- les index
  - `\di` , `\di+`
- les schémas
  - `\dn`
- les fonctions & procédures
  - `\df[+]`
- etc…

Ces commandes permettent d'obtenir des informations sur les objets utilisateurs de tout type stockés dans la base de données. Pour les commandes qui acceptent un motif, celui-ci permet de restreindre les résultats retournés à ceux dont le nom d'opérateur correspond au motif précisé.

Le client psql en version 15 est compatible avec toutes les versions du serveur depuis la 9.2 incluse.

`\l` dresse la liste des bases de données sur le serveur. Avec `\l+`, les commentaires, les tablespaces par défaut et les tailles des bases sont également affichés, ce qui peut être très pratique.

`\dt` affiche les tables, `\di` les index, `\dn` les schémas, `\ds` les séquences, `\dv` les vues, etc. Là encore, on peut ajouter `+` pour d'autres informations comme la taille, et même `S` pour inclure les objets système normalement masqués.

Exemple :

Si l'on crée une simple base grâce à `pgbench` avec un utilisateur dédié :

```
$ psql
```

```
=# CREATE ROLE testeur LOGIN ;
=# \password testeur
Saisir le nouveau mot de passe :
Saisir le mot de passe à nouveau :
```

Utilisateurs en place :

```
=# \du
                             Liste des rôles
         Nom du rôle         |        Attributs        |       Membre de
-----------------------------+-------------------------+------------------
 dalibo                      |                         | {}
 nagios                      | Superutilisateur        | {}
 postgres                    | Superutilisateur, (...) | {}
 testeur                     |                         | {}
```

Création de la base :

```
CREATE DATABASE pgbench OWNER testeur ;
```

Bases de données en place :

```
=# \l
                          Liste des bases de données
      Nom       | Propriétaire | Encodage | Collationnement | Type caract. | …
----------------+--------------+----------+-----------------+--------------+----
 pgbench        | testeur      | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 postgres       | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
```

Création des tables de cette nouvelle base :

```
$ pgbench --initialize --scale=10 -U testeur -h localhost pgbench
```

Connexion à la nouvelle base :

```
$ psql -h localhost -U testeur -d pgbench
```

Les tables :

```
=# \d
               Liste des relations
 Schéma |       Nom        | Type  | Propriétaire
--------+------------------+-------+--------------
 public | pgbench_accounts | table | testeur
 public | pgbench_branches | table | testeur
 public | pgbench_history  | table | testeur
 public | pgbench_tellers  | table | testeur
(4 lignes)

=# \dt+ pgbench_*s
                          Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Persistence |  M…  | Taille |
--------+------------------+-------+--------------+-------------+------+--------+--
 public | pgbench_accounts | table | testeur      | permanent   | heap | 128 MB |
 public | pgbench_branches | table | testeur      | permanent   | heap | 40 kB  |
 public | pgbench_tellers  | table | testeur      | permanent   | heap | 40 kB  |
(3 lignes)
```

Une table (affichage réduit pour des raisons de mise en page) :

```
=# \d+ pgbench_accounts
                          Table « public.pgbench_accounts »
 Colonne  |     Type      | Col..nt | NULL-able | … | Stockage | Compr. | …
----------+---------------+---------+-----------+---+----------+--------+--
```

```
 aid      | integer       |         | not null  |   | plain    |        |
 bid      | integer       |         |           |   | plain    |        |
 abalance | integer       |         |           |   | plain    |        |
 filler   | character(84) |         |           |   | extended |        |
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Méthode d'accès : heap
Options: fillfactor=100
```

Les index :

```
=> \di
                          Liste des relations
 Schéma |          Nom          | Type  | Propriétaire |      Table
--------+-----------------------+-------+--------------+------------------
 public | pgbench_accounts_pkey | index | testeur      | pgbench_accounts
 public | pgbench_branches_pkey | index | testeur      | pgbench_branches
 public | pgbench_tellers_pkey  | index | testeur      | pgbench_tellers
(3 lignes)
```

Les schémas, utilisateur ou système :

```
=> \dn+
                          Liste des schémas
  Nom   | Propriétaire |    Droits d´accès    |      Description
--------+--------------+----------------------+------------------------
 public | postgres     | postgres=UC/postgres+| standard public schema
        |              | =UC/postgres         |
(1 ligne)

=> \dnS
         Liste des schémas
        Nom         | Propriétaire
--------------------+--------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | postgres
(4 lignes)
```

Les tablespaces (ici ceux par défaut) :

```
=> \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
(2 lignes)
```

4.4.4 Catalogue système : rôles et accès

- Lister les rôles (utilisateurs et groupes)
  - `\du[+]`
- Lister les droits d'accès
  - `\dp`
- Lister les droits d'accès par défaut
  - `\ddp`
  - `ALTER DEFAULT PRIVILEGES`
- Lister les configurations par rôle et par base
  - `\drds`

On a vu que `\du` (`u` pour *user*) affiche les rôles existants. Rappelons qu'un rôle peut être aussi bien un utilisateur (si le rôle a le droit de `LOGIN`) qu'un groupe d'utilisateurs, voire les deux à la fois.

Dans les versions antérieures à la 8.1, il n'y avait pas de rôles, et les groupes et les utilisateurs étaient deux notions distinctes. Certaines commandes ont conservé le terme de *user*, mais il s'agit bien de rôles dans tous les cas.

Les droits sont accessibles par les commandes `\dp` (`p` pour « permissions ») ou `\z`.

Dans cet exemple, le rôle **admin** devient membre du rôle système **pg_signal_backend** :

```
=# GRANT pg_signal_backend TO admin;

=# \du
                                 List of roles
 Nom du rôle |                      Attributs                      |      Membre de
-------------+-----------------------------------------------------+---------------------
 admin       |                                                     | {pg_signal_backend}
 postgres    | Superuser, Create role, Create DB,                  |
             | Replication, Bypass RLS                             | {}
 …
```

Toujours dans la base **pgbench** :

```
=# GRANT SELECT ON TABLE pgbench_accounts TO dalibo ;
```

```
=# GRANT ALL ON TABLE pgbench_history TO dalibo ;
=# \z
```

```
=> \z
                                  Droits d'accès
 Schéma |       Nom        | Type  |     Droits d'accès      | Droits … | …
--------+------------------+-------+-------------------------+----------+---
 public | pgbench_accounts | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=r/testeur        |          |
 public | pgbench_branches | table |                         |          |
 public | pgbench_history  | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=arwdDxt/testeur  |          |
 public | pgbench_tellers  | table |                         |          |
(4 lignes)
```

La commande `\ddp` permet de connaître les droits accordés par défaut à un utilisateur sur les nouveaux objets avec l'ordre `ALTER DEFAULT PRIVILEGES`.

```
=# ALTER DEFAULT PRIVILEGES GRANT ALL ON TABLES TO dalibo ;
```

```
=# \ddp
                Droits d´accès par défaut
 Propriétaire | Schéma | Type  |     Droits d´accès
--------------+--------+-------+------------------------
 testeur      |        | table | dalibo=arwdDxt/testeur +
              |        |       | testeur=arwdDxt/testeur
(1 ligne)
```

Enfin, la commande `\drds` permet d'obtenir la liste des paramètres appliqués spécifiquement à un utilisateur ou une base de données.

```
=# ALTER DATABASE pgbench SET work_mem TO '15MB';
=# ALTER ROLE testeur SET log_min_duration_statement TO '0';
```

```
=# \drds
              Liste des paramètres
  Rôle   | Base de données |         Réglages
---------+-----------------+--------------------------
 testeur |                 | log_min_duration_statement=0
         | pgbench         | work_mem=15MB
…
```

4.4.5 Visualiser le code des objets

- Voir les vues ou les fonctions & procédures
  - `\dv`, `\df`
- Code d'une vue
  - `\sv`
- Code d'une procédure stockée
  - `\sf`

Ceci permet de visualiser le code de certains objets sans avoir besoin de l'éditer. Par exemple avec cette vue système :

```
=# \dv pg_tables
             Liste des relations
   Schéma   |    Nom    | Type | Propriétaire
------------+-----------+------+--------------
 pg_catalog | pg_tables | vue  | postgres
(1 ligne)


=# \sv+ pg_tables
1       CREATE OR REPLACE VIEW pg_catalog.pg_tables AS
2        SELECT n.nspname AS schemaname,
3           c.relname AS tablename,
4           pg_get_userbyid(c.relowner) AS tableowner,
5           t.spcname AS tablespace,
6           c.relhasindex AS hasindexes,
7           c.relhasrules AS hasrules,
8           c.relhastriggers AS hastriggers,
9           c.relrowsecurity AS rowsecurity
10         FROM pg_class c
11           LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
12           LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
13        WHERE c.relkind = ANY (ARRAY['r'::"char", 'p'::"char"])
```

Ou cette fonction :

```
=# CREATE FUNCTION nb_sessions_actives () RETURNS int
LANGUAGE sql
AS $$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$$ ;
```

```
=# \df nb_sessions_actives
                                        Liste des fonctions
 Schéma |         Nom          | Type … du résultat | Type …des paramètres | Type
--------+----------------------+--------------------+----------------------+------
 public | nb_sessions_actives | integer            |                      | func
(1 ligne)
```

```
=# \sf nb_sessions_actives
CREATE OR REPLACE FUNCTION public.nb_sessions_actives()
 RETURNS integer
 LANGUAGE sql
AS $function$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$function$
```

Il est même possible de lancer un éditeur depuis `psql` pour modifier directement la vue ou la fonction avec respectivement `\ev` ou `\ef`.

4.4.6 Configuration

- Lister les paramètres correspondants au motif indiqué
  - `\dconfig` (v15+)

La méta-commande `\dconfig` permet de récupérer la configuration d'un paramètre. Par exemple :

```
b1=# \dconfig log_r*
  List of configuration parameters
         Parameter          | Value
----------------------------+-------
 log_recovery_conflict_waits | off
 log_replication_commands    | off
 log_rotation_age            | 1d
 log_rotation_size           | 0
(4 rows)
```

En ajoutant le signe `+`, il est possible d'obtenir plus d'informations :

```
b1=# \dconfig+ log_r*
                      List of configuration parameters
         Parameter          | Value |  Type   |  Context  | Access privileges
----------------------------+-------+---------+-----------+-------------------
 log_recovery_conflict_waits | off   | bool    | sighup    |
 log_replication_commands    | off   | bool    | superuser |
 log_rotation_age            | 1d    | integer | sighup    |
 log_rotation_size           | 0     | integer | sighup    |
(4 rows)
```

4.4.7 Exécuter des requêtes

- Exécuter une requête :

```
SELECT * FROM pg_tables ;
SELECT * FROM pg_tables \g
SELECT * FROM pg_tables \gx   -- une ligne par champ
INSERT INTO … VALUES (1) \; INSERT INTO … VALUES (2) ; -- 1 transaction
```

- Rappel des requêtes:
  - flèche vers le haut
  - `\g`
  - `Ctrl-R` suivi d'un extrait de texte représentatif

Les requêtes SQL doivent se terminer par `;` ou, pour marquer la parenté de PostgreSQL avec Ingres, `\g`. Cette dernière commande permet de relancer un ordre.

Depuis version 10, il est aussi possible d'utiliser `\gx` pour bénéficier de l'affichage étendu sans avoir à jouer avec `\x on` et `\x off`.

La console `psql`, lorsqu'elle est compilée avec la bibliothèque `libreadline` ou la bibliothèque `libedit` (cas des distributions Linux courantes), dispose des mêmes possibilités de rappel de commande que le shell bash.

Exemple :

```
postgres=# SELECT * FROM pg_tablespace LIMIT 5;

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# SELECT * FROM pg_tablespace LIMIT 5\g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# \g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
```

```
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |


postgres=# \gx


-[ RECORD 1 ]----------
oid        | 1663
spcname    | pg_default
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 2 ]----------
oid        | 1664
spcname    | pg_global
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 3 ]----------
oid        | 16502
spcname    | ts1
spcowner   | 10
spcacl     |
spcoptions |
```

Plusieurs ordres sur la même ligne séparés par des `;` seront exécutés et affichés à la suite. Par contre, s'ils sont séparés par `\;`, ils seront envoyés ensemble et implicitement dans une même transaction (sauf utilisation explicite de `BEGIN` / `COMMIT` / `ROLLBACK`). Avant la version 15, ne sera affiché que le résultat du dernier ordre.

Dans cet exemple, la division par zéro fait tomber en erreur et annule l'insertion de la valeur 2 car les deux sont dans la même transaction :

```
=# CREATE TABLE demo_insertion (i float);
CREATE TABLE
=# INSERT INTO demo_insertion VALUES(1.0); INSERT INTO demo_insertion VALUES(1.0/0);
INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion \; INSERT INTO demo_insertion VALUES (2.0) \;
   INSERT INTO demo_insertion VALUES (2.0/0) ;
 i
 1
(1 ligne)

INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion ;
 i
 1
(1 ligne)
```

4.4.8 Afficher le résultat d'une requête

- `\x` pour afficher un champ par ligne
- Affichage par paginateur si l'écran ne suffit pas
- Préférer `less` :
  - `set PAGER='less -S'`
- Ou outil dédié : pspg[12]
  - `\setenv PAGER 'pspg'`

En mode interactif, `psql` cherche d'abord à afficher directement le résultat :

```
postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

      relname      | reltype | relchecks |  oid  |  oid
-------------------+---------+-----------+-------+-------
 pg_statistic      |   11319 |         0 |  2619 |  2619
 t3                |   16421 |         0 | 16419 | 16419
 capitaines_id_seq |   16403 |         0 | 16402 | 16402
 t1                |   16415 |         0 | 16413 | 16413
 t2                |   16418 |         0 | 16416 | 16416
```

S'il y a trop de colonnes, on peut préférer n'avoir qu'un champ par ligne grâce au commutateur `\x` :

```
postgres=# \x on

Expanded display is on.

postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

-[ RECORD 1 ]----------------
relname   | pg_statistic
reltype   | 11319
relchecks | 0
oid       | 2619
oid       | 2619
-[ RECORD 2 ]----------------
relname   | t3
reltype   | 16421
relchecks | 0
oid       | 16419
oid       | 16419
-[ RECORD 3 ]----------------
relname   | capitaines_id_seq
reltype   | 16403
relchecks | 0
```

```
oid       | 16402
oid       | 16402
```

`\x on` et `\x off` sont alternativement appelés si l'on tape plusieurs fois `\x`. `\x auto` délègue à `psql` la décision du meilleur affichage, en général à bon escient. `\gx` à la place de `;` bascule l'affichage pour une seule requête.

S'il n'y a pas la place à l'écran, `psql` appelle le paginateur par défaut du système. Il s'agit souvent de `more`, parfois de `less`. Ce dernier est bien plus puissant, permet de parcourir le résultat en avant et en arrière, avec la souris, de chercher une chaîne de caractères, de tronquer les lignes trop longues (avec l'option `-S`) pour naviguer latéralement.

Le paramétrage du paginateur s'effectue par des variables d'environnement :

```
export PAGER='less -S'
psql
```

ou :

```
PAGER='less -S'  psql
```

ou dans `psql` directement, ou `.psqlrc` :

```
\setenv PAGER 'less -S'
```

Mais `less` est généraliste. Un paginateur dédié à l'affichage de résultats de requêtes a été récemment développé par Pavel Stěhule[13] et son paquet figure dans les principales distributions.

Pour les gens qui consultent souvent des données dans les tables depuis la console, `pspg` permet de naviguer dans les lignes avec la souris en figeant les entêtes ou quelques colonnes ; de poser des signets sur des lignes ; de sauvegarder les lignes. (Pour plus de détail, voir cette présentation[14] et la page Github du projet[15]). La mise en place est similaire :

```
\setenv PAGER 'pspg'
```

À l'inverse, la pagination se désactive complètement avec :

```
\pset pager
```

(et bien sûr en mode non interactif, par exemple quand la sortie de `psql` est redirigée vers un fichier).

[13] http://okbob.blogspot.fr/2017/07/i-hope-so-every-who-uses-psql-uses-less.html
[14] http://blog-postgresql.verite.pro/2017/11/21/test-pspg.html
[15] https://github.com/okbob/pspg

4.4.9 Afficher les détails d'une requête

- `\gdesc`
- Afficher la liste des colonnes correspondant au résultat d'exécution d'une requête
  - noms
  - type de données

Après avoir exécuté une requête, ou même à la place de l'exécution, il est possible de connaître le nom des colonnes en résultat ainsi que leur type de données.

Requête :

```
SELECT nspname, relname
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

Description des colonnes :

```
postgres=# \gdesc

 Column  | Type
---------+------
 nspname | name
 relname | name
```

Ou sans exécution :

```
postgres=# SELECT * FROM generate_series (1, 1000) \gdesc

     Column      |  Type
-----------------+---------
 generate_series | integer
```

4.4.10 Exécuter le résultat d'une requête

- Exécuter le résultat d'une requête
  - `\gexec`

Parfois, une requête permet de créer des requêtes sur certains objets. Par exemple, si nous souhaitons exécuter un `VACUUM` sur toutes les tables du schéma `public`, nous allons récupérer la liste des tables avec cette requête :

```
SELECT nspname, relname FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
 nspname |     relname
---------+------------------
 public  | pgbench_branches
 public  | pgbench_tellers
 public  | pgbench_accounts
 public  | pgbench_history
(4 lignes)
```

Plutôt que d'éditer manuellement cette liste de tables pour créer les ordres SQL nécessaires, autant modifier la requête pour qu'elle prépare elle-même les ordres SQL :

```
SELECT 'VACUUM ' || quote_ident(nspname) || '.' || quote_ident(relname)
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
           ?column?
--------------------------------
 VACUUM public.pgbench_branches
 VACUUM public.pgbench_tellers
 VACUUM public.pgbench_accounts
 VACUUM public.pgbench_history
(4 lignes)
```

Une fois que nous avons vérifié la validité des requêtes SQL, il ne reste plus qu'à les exécuter. C'est ce que permet la commande `\gexec` :

```
=# \gexec
```

```
VACUUM
VACUUM
VACUUM
VACUUM
```

4.4.11 Manipuler le tampon de requêtes

- Éditer
  - dernière requête : `\e`
  - vue : `\ev nom_vue`
  - fonction PL/pgSQL : `\ef nom_fonction`
- Historique :
  - `\s`

`\e nomfichier.sql` édite le tampon de requête courant ou un fichier existant indiqué à l'aide d'un éditeur externe.

L'éditeur désigné avec les variables d'environnement `$EDITOR` ou `$PSQL_EDITOR` notamment. Sur Unix, c'est généralement par défaut une variante de `vi` mais n'importe quel éditeur fait l'affaire :

```
PSQL_EDITOR=nano  psql
```

ou dans `psql` ou dans le `.psqlrc` :

```
\setenv PSQL_EDITOR 'gedit'
```

`\p` affiche le contenu du tampon de requête.

`\r` efface la requête qui est en cours d'écriture. La précédente requête reste accessible avec `\p`.

`\w nomfichier.sql` provoque l'écriture du tampon de requête dans le fichier indiqué (à modifier par la suite avec `\e` par exemple).

`\ev v_mavue` édite la vue indiquée. Sans argument, cette commande affiche le squelette de création d'une nouvelle vue.

`\ef f_mafonction` est l'équivalent pour une fonction.

`\s` affiche l'historique des commandes effectuées dans la session. `\s historique.log` sauvegarde cet historique dans le fichier indiqué.

4.4.12 Entrées/sorties

- Charger et exécuter un script SQL
  - `\i fichier.sql`
- Rediriger la sortie dans un fichier
  - `\o resultat.out`
- Écrire un texte sur la sortie standard
  - `\echo "Texte…"`

`\i fichier.sql` lance l'exécution des commandes placées dans le fichier passé en argument. `\ir` fait la même chose sauf que le chemin est relatif au chemin courant.

`\o resultat.out` envoie les résultats de la requête vers le fichier indiqué (voire vers une commande UNIX via un *pipe*).

Exemple :

```
=# \o tables.txt
=# SELECT * FROM pg_tables ;
```

(Si l'on intercale `\H`, on peut avoir un formatage en HTML.)

Attention : cela va concerner toutes les commandes suivantes. Entrer `\o` pour revenir au mode normal.

`\echo "Texte"` affiche le texte passé en argument sur la sortie standard. Ce peut être utile entre les étapes d'un script.

4.4.13 Gestion de l'environnement système

- Chronométrer les requêtes
  - `\timing on`
- Exécuter une commande OS
  - `\! ls -l` (sur le client !)
- Changer de répertoire courant
  - `\cd /tmp`
- Affecter la valeur d'une variable d'environnement (v15+)
  - `\getenv toto PATH`

`\timing on` active le chronométrage de toutes les commandes. C'est très utile mais alourdit l'affichage. Sans argument, la valeur actuelle est basculée de `on` à `off` et vice-versa.

```
=# \timing on
Chronométrage activé.

=# VACUUM ;
VACUUM
Temps : 26,263 ms
```

`\! <commande>` ouvre un shell interactif en l'absence d'argument ou exécute la commande indiquée **sur le client** (pas le serveur !) :

`\cd` (et non `\! cd` !) permet de changer de répertoire courant, là encore **sur le client**. Cela peut servir pour définir le chemin d'un script à exécuter ou d'un futur export.

`\getenv` permet de récupérer la valeur d'une valeur d'environnement système et de l'affecter à une variable.

Exemple :

```
\! cat nomfichier.out
\! ls -l /tmp
\! mkdir /home/dalibo/travail
\cd /home/dalibo/travail
\! pwd
/home/dalibo/travail
```

4.4.14 Variables internes psql

- Positionner des variables internes
  - `\set NOMVAR nouvelle_valeur`
- Variables internes usuelles
  - `ON_ERROR_STOP` : `on` / `off`
  - `ON_ERROR_ROLLBACK` : `on` / `off` / `interactive`
  - `ROW_COUNT` : nombre de lignes renvoyées par la dernière requête (v11)
  - `ERROR` : `true` si dernière requête en erreur (v11)
- Ne pas confondre avec `SET` (au niveau du serveur) !

Les variables déclarées avec `\set` sont positionnées au niveau de `psql`, outil client. Elles ne sont pas connues du serveur et n'existent pas dans les autres outils clients (pgAdmin, DBeaver…).
Il ne faut pas les confondre avec les paramètres définis sur le serveur au niveau de la session avec `SET`. Ceux-ci sont transmis directement au serveur quand ils sont entrés dans un outil client, quel qu'il soit.

`\set` affiche les variables internes et utilisateur.

`\set NOMVAR nouvelle_valeur` permet d'affecter une valeur.

La liste des variables prédéfinies est disponible dans la documentation de psql[16]. Beaucoup modifient le comportement de `psql`.

Exemple :

```
postgres=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'b1'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
FETCH_COUNT = '0'
HIDE_TABLEAM = 'off'
HIDE_TOAST_COMPRESSION = 'off'
```

[16] https://docs.postgresql.fr/current/app-psql.html#app-psql-variables

```
HISTCONTROL = 'none'
HISTSIZE = '500'
HOST = '/var/run/postgresql'
IGNOREEOF = '0'
LAST_ERROR_MESSAGE = ''
LAST_ERROR_SQLSTATE = '00000'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5432'
PROMPT1 = '%/%R%x%# '
PROMPT2 = '%/%R%x%# '
PROMPT3 = '>> '
QUIET = 'off'
SERVER_VERSION_NAME = '15.1'
SERVER_VERSION_NUM = '150001'
SHOW_ALL_RESULTS = 'on'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 15.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
↪  20210514 (Red Hat 8.5.0-15), 64-bit'
VERSION_NAME = '15.1'
VERSION_NUM = '150001'
```

Les variables `ON_ERROR_ROLLBACK` et `ON_ERROR_STOP` sont discutées dans la partie relative à la gestion des erreurs.

La version 11 ajoute quelques variables internes. `ROW_COUNT` indique le nombre de lignes de résultat de la dernière requête exécutée :

```
=# SELECT * FROM pg_namespace;
      nspname       | nspowner |               nspacl
--------------------+----------+-------------------------------------
 pg_toast           |       10 |
 pg_temp_1          |       10 |
 pg_toast_temp_1    |       10 |
 pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
 public             |       10 | {postgres=UC/postgres,=UC/postgres}
 information_schema |       10 | {postgres=UC/postgres,=U/postgres}

=# \echo :ROW_COUNT
6

=# SELECT :ROW_COUNT ;
6
```

alors que `ERROR` est un booléen indiquant si la dernière requête était en erreur ou pas :

```
=# CREATE TABLE t1(id integer);
CREATE TABLE

=# CREATE TABLE t1(id integer);

ERROR:  relation "t1" already exists
```

```
postgres=# \echo :ERROR
true
postgres=# CREATE TABLE t2(id integer);
CREATE TABLE
postgres=# \echo :ERROR
false
```

4.4.15 Variables utilisateur psql

- Définir une variable utilisateur
  - `\set NOMVAR nouvelle_valeur`
- Invalider une variable
  - `\unset NOMVAR`
- Stockage du résultat d'une requête :
  - si résultat est une valeur unique
  - Exemple :

```
SELECT now() AS maintenant \gset
SELECT :'maintenant' ;
```

En dehors des variables internes évoquées dans le chapitre précédent, il est possible à un utilisateur de `psql` de définir ses propres variables.

```
-- initialiser avec une constante
\set active 'y'
\echo :active
y
-- initialiser avec le résultat d'une commande système
\set uptime `uptime`
\echo :uptime

 09:38:58 up 53 min,  1 user,  load average: 0,12, 0,07, 0,07
```

Et pour supprimer la variable :

```
\unset uptime
```

Il est possible de stocker le résultat d'une requête dans une variable pour sa réutilisation dans un script avec la commande `\gset`. Le nom de la variable est celui de la colonne ou de son alias. La valeur retournée par la requête doit être unique sous peine d'erreur.

```
SELECT now() AS "curdate" \gset
\echo :curdate
```

```
2020-08-16 10:53:51.795582+02
```

Il est possible aussi de donner un préfixe au nom de la variable :

```
SELECT now() AS "curdate" \gset v_
\echo :v_curdate

2020-08-16 10:54:20.484344+02
```

4.4.16 Tests conditionnels

- `\if`
- `\elif`
- `\else`
- `\endif`

Ces quatre instructions permettent de tester la valeur de variables `psql`, ce qui permet d'aller bien plus loin dans l'écriture de scripts SQL. Le client doit être en version 10 au moins (pas forcément le serveur).

Par exemple, si on souhaite savoir si on se trouve sur un serveur standby ou sur un serveur primaire, il suffit de configurer la variable `PROMPT1` à partir du résultat de l'interrogation de la fonction `pg_is_in_recovery()`. Pour cela, il faut enregistrer ce code dans le fichier `.psqlrc` :

```
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
    \set PROMPT1 'standby %x$ '
\else
    \set PROMPT1 'primaire %x$ '
\endif
```

Puis, en lançant `psql` sur un serveur primaire, on obtient :

```
psql (15.1)
Type "help" for help.

primaire $
```

alors qu'on obtient sur un serveur secondaire :

```
psql (15.1)
Type "help" for help.

standby $
```

4.4.17 Personnaliser psql

- Fichier de configuration `~/.psqlrc`
  - voire `~/.psqlrc-X.Y` ou `~/.psqlrc-X`
  - ignoré avec `-X`
- Exemple de `.psqlrc` :

```
\set ON_ERROR_ROLLBACK interactive  -- paramétrage de session
\timing on
\set PROMPT1 '%M:%> %n@%/%R%#%x'    -- invite
\set cfg 'SHOW ALL ;'               -- requête utilisable avec :cfg
\set cls '\\! clear;'               -- nettoyer l'écran avec :cls
```

`psql` est personnalisable par le biais de plusieurs variables internes. Il est possible de pérenniser ces personnalisations par le biais d'un fichier `~/.psqlrc` (ou `%APPDATA%\postgresql\psqlrc.conf` sous Windows, ou dans un répertoire désigné par `$PSQLRC`). Il peut exister des fichiers par version (par exemple `~/.psqlrc-15` ou `~/.psqlrc-15.0`), voire un fichier global[17].

Modifier l'invite est utile pour toujours savoir où et comment l'on est connecté. Tous les détails sont dans la documentation[18]. Par exemple, ajouter `%>` dans l'invite affiche le port. On peut aussi afficher toutes les variables listées par `\set` (par exemple ainsi : `%:PORT:` ou `%:USER:`). En cas de reconnexion, psql en régénère certaines, mais pas toutes.

Exemple de fichier `.psqlrc` :

```
\set QUIET 1
\timing on
\pset pager on
\setenv pager
\pset null 'ø'
-- Mots clés autocomplétés en majuscule
\set COMP_KEYWORD_CASE upper
-- Affichage
\x auto
\pset linestyle 'unicode'
\pset border 2
\pset unicode_border_linestyle double
\pset unicode_column_linestyle double
\pset unicode_header_linestyle double
-- Prompt dynamique
-- %> indique le port, %m le serveur, %n l'utilisateur, %/ la base…
```

[17] https://docs.postgresql.fr/current/app-psql.html#id-1.9.4.20.10
[18] https://docs.postgresql.fr/current/app-psql.html#app-psql-prompting

```
\set PROMPT1 '%m:%> %[%033[1;33;40m%]%n@%/%R%[%033[0m%]%#%x '
-- serveur secondaire ? (NB : non mis à jour lors d'une reconnexion !)
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
   \set PROMPT1 :PROMPT1 '(standby) '
\else
   \set PROMPT1 :PROMPT1
\endif
\set QUIET 0
```

```
$ psql -h serveur -p5435 -U jeanpierre -d mabase
```

```
[serveur]:5435 jeanpierre@postgres=# (standby) SELECT pi(), now(), null ;
```

pi	now	?column?
3.141592653589793	2022-01-07 16:08:39.925262+01	ø

```
(1 ligne)
```

Il est aussi possible d'y rajouter du paramétrage de session avec `SET` pour adapter le fuseau horaire, par exemple.

Des requêtes très courantes peuvent être ajoutées dans le `.psqlrc`, par exemple celles-ci :

```
-- Paramètres en cours avec leur source
-- Ceci impérativement sur une seule ligne !
\set config 'SELECT name, current_setting(name), CASE source WHEN $$configuration
 ↪  file$$ THEN
regexp_replace(sourcefile, $$^/.*/$$, $$$$)||$$:$$||sourceline ELSE source END FROM
 ↪  pg_settings
WHERE source <> $$default$$ OR name LIKE $$%.%$$;'
```

(Requête inspirée de Christoph Berg[19]).

```
\set extensions 'SELECT * FROM pg_available_extensions ORDER BY name ;'
```

…utilisables ainsi dans `psql` :

```
=# :config
```

```
=# :extensions
```

Attention : le `.psqlrc` n'est exécuté qu'au démarrage de `psql`, mais pas lors d'une reconnexion avec `\c` ! Les prompts dynamiques à base de variables utilisateur sont donc susceptibles d'être alors faux ! Pour relancer le script, utiliser :

```
\i ~/.psqlrc
```

[19] https://www.postgresql.org/message-id/YlFQLzlPi4QD0wSi%40msg.df7cb.de

Dans un script, il vaut mieux ignorer ce fichier de configuration grâce à `--no-psqlrc` (`-X`) pour revenir à l'environnement par défaut et éviter de polluer l'affichage :

```
$ psql -X -f script.sql
$ psql -X -At -c 'SELECT name, setting FROM pg_settings ;'
```

4.5 ÉCRITURE DE SCRIPTS SHELL

- Script SQL
- Script Shell
- Exemple sauvegarde

4.5.1 Exécuter un script SQL avec psql

- Avec `-c` :

```
psql -c 'SELECT * FROM matable' -c 'SELECT fonction(123)' ;
```

- Avec un script :

```
psql -f nom_fichier.sql
```

```
psql < nom_fichier.sql
```

- Depuis `psql` :
  - `\i nom_fichier.sql`

L'option `-c` permet de spécifier du SQL en ligne de commande sans avoir besoin de faire un script. Plusieurs ordres seront enchaînés dans la même session.

Il est généralement préférable d'enregistrer les ordres dans des fichiers si on veut les exécuter plusieurs fois sans se tromper. L'option `-f` est très utile dans ce cas. La redirection avec `<` est une alternative répandue.

4.5.2 Gestion des transactions

- `psql` est en mode auto-commit par défaut
  - variable `AUTOCOMMIT`
- Ouvrir une transaction explicitement
  - `BEGIN;`
- Terminer une transaction
  - `COMMIT;` ou `ROLLBACK;`
- Ouvrir une transaction implicitement
  - option `-1` (`--single-transaction`)

Par défaut, `psql` est en mode « autocommit », c'est-à-dire que tous les ordres SQL sont automatiquement validés après leur exécution.

Pour exécuter une suite d'ordres SQL dans une seule et même transaction, il faut soit ouvrir explicitement une transaction avec `BEGIN;` et la valider avec `COMMIT;` ou l'annuler avec `ROLLBACK;`. Les autres outils clients sont généralement dans ce même cas.

L'ordre

```
=# \set AUTOCOMMIT off
```

a pour effet d'insérer systématiquement un `BEGIN` avant chaque ordre s'il n'y a pas déjà une transaction ouverte ; il faudra valider ensuite avec `COMMIT` **avant** de déconnecter. Il est déconseillé de changer le comportement par défaut (`on`), même s'il peut désorienter au premier abord des personnes ayant connu une base de données concurrente.

Une autre possibilité est d'utiliser `psql -1` ou `psql --single-transation` : les ordres sont automatiquement encadrés d'un `BEGIN` et d'un `COMMIT`. La présence d'ordres `BEGIN`, `COMMIT` ou `ROLLBACK` explicites modifiera ce comportement en conséquence.

4.5.3 Écrire un script SQL

- Attention à l'encodage des caractères
  - `\encoding`
  - `SET client_encoding`
- Écriture des requêtes

L'encodage a moins d'importance depuis qu'UTF-8 s'est généralisé, mais il y a encore parfois des problèmes dans de vieilles bases ou certains vieux outils.

Rappelons que les bases modernes devraient toutes utiliser l'encodage UTF-8 (c'est le défaut).

`\encoding [ENCODAGE]` permet, en l'absence d'argument, d'afficher l'encodage du client. En présence d'un argument, il permet de préciser l'encodage du client.

Exemple :

```
postgres=# \encoding
UTF8
postgres=# \encoding LATIN9
postgres=# \encoding
LATIN9
```

Cela a le même effet que d'utiliser l'ordre SQL `SET client_encoding TO LATIN9;`.

En terme de présentation, il est commun d'écrire les mots clés SQL en majuscules et d'écrire les noms des objets et fonctions manipulés dans les requêtes en minuscule. Le langage SQL est un langage au même titre que Java ou PHP, la présentation est importante pour la lisibilité des requêtes, même si les variations personnelles sont nombreuses.

4.5.4 Les blocs anonymes

- Bloc procédural anonyme en PL/pgSQL :

```
DO $$
DECLARE r record;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        RAISE NOTICE 'Analyze %.%', r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END$$;
```

Les blocs anonymes sont utiles pour des petits scripts ponctuels qui nécessitent des boucles ou du conditionnel, voire du transactionnel, sans avoir à créer une fonction ou une procédure. Ils ne renvoient rien. Ils sont habituellement en PL/pgSQL mais tout langage procédural installé est possible.

L'exemple ci-dessus lance un `ANALYZE` sur toutes les tables où les statistiques n'ont pas été calculées d'après la vue système, et donne aussi un exemple de SQL dynamique. Le résultat est par exemple :

```
NOTICE:  Analyze public.pgbench_history
NOTICE:  Analyze public.pgbench_tellers
NOTICE:  Analyze public.pgbench_accounts
NOTICE:  Analyze public.pgbench_branches
DO
Temps : 141,208 ms
```

(Pour ce genre de SQL dynamique, si l'on est sous `psql` , il est souvent plus pratique d'utiliser `\gexec` [20].)

Noter que les ordres constituent une transaction unique, à moins de rajouter des `COMMIT` ou `ROLLBACK` explicitement (ce n'est autorisé qu'à partir de la version 11).

[20] https://docs.postgresql.fr/current/app-psql.html#R2-APP-PSQL-4

4.5.5 Utiliser des variables

```
\set nom_table 'ma_table'
SELECT * FROM :"nom_table";

\set valeur_col1 'test'
SELECT * FROM :"nom_table" WHERE col1 = :'valeur_col1';

\prompt 'invite' nom_variable
\unset variable

psql -v VARIABLE=valeur
```

psql permet de manipuler des variables internes personnalisées dans les scripts. Ces variables peuvent être particulièrement utiles pour passer des noms d'objets ou des termes à utiliser dans une requête par le biais des options de ligne de commande (`-v variable=valeur`).

Noter la position des guillemets quand la variable est une chaîne de caractères !

Exemple :

Une fois connecté à la base **pgbench**, on déclare deux variables propres au client :

```
pgbench=# \set nomtable pgbench_accounts
pgbench=# \set taillemini 1000000
```

Elles apparaissent bien dans la liste des variables :

```
pgbench=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'pgbench'
…
nomtable = 'pgbench_accounts'
taillemini = '1000000'
```

Elles s'utilisent ainsi :

```
SELECT pg_relation_size (:'nomtable') ;

 pg_relation_size
------------------
        134299648

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class
WHERE relkind = 'r' AND pg_relation_size (oid) > :taillemini ;
```

```
     relname      | pg_size_pretty
------------------+----------------
 pgbench_accounts | 128 MB
```

La substitution s'effectue bien au niveau du client. Si l'on trace tout au niveau du serveur, ces requêtes apparaissent :

```
SELECT pg_relation_size ('pgbench_accounts') ;

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class WHERE relkind = 'r'
AND pg_relation_size (oid) > 1000000 ;
```

4.5.6 Gestion des erreurs

- Ignorer les erreurs dans une transaction
  - `ON_ERROR_ROLLBACK`
- Gérer des erreurs SQL en shell
  - `ON_ERROR_STOP`

La variable interne `ON_ERROR_ROLLBACK` n'a de sens que si elle est utilisée dans une transaction. Elle peut prendre trois valeurs :

- `off` (défaut) ;
- `on` ;
- `interactive`.

Lorsque `ON_ERROR_ROLLBACK` est à `on`, psql crée un `SAVEPOINT` systématiquement avant d'exécuter une requête SQL. Ainsi, si la requête SQL échoue, psql effectue un `ROLLBACK TO SAVEPOINT` pour annuler cette requête. Sinon il relâche le `SAVEPOINT`.

Lorsque `ON_ERROR_ROLLBACK` est à `interactive`, le comportement de psql est le même seulement si il est utilisé en interactif. Si psql exécute un script, ce comportement est désactivé. Cette valeur permet de se protéger d'éventuelles fautes de frappe.

Utiliser cette option n'est donc pas neutre, non seulement en terme de performances, mais également en terme d'intégrité des données. Il ne faut donc pas utiliser cette option à la légère.

Enfin, la variable interne `ON_ERROR_STOP` a deux objectifs : arrêter l'exécution d'un script lorsque psql rencontre une erreur et retourner un code retour shell différent de 0. Si cette variable reste à `off`, psql retournera toujours la valeur 0 même s'il a rencontré une erreur dans l'exécution d'une requête.

Une fois activée, psql retournera un code d'erreur 3 pour signifier qu'il a rencontré une erreur dans l'exécution du script.

L'exécution d'un script qui comporte une erreur retourne le code 0, signifiant que psql a pu se connecter à la base de données et exécuté le script :

```
$ psql -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
0
```

Lorsque la variable `ON_ERROR_STOP` est activée, psql retourne un code erreur 3, signifiant qu'il a rencontré une erreur :

```
$ psql -v ON_ERROR_STOP=on -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
3
```

psql retourne les codes d'erreurs suivant au shell :

- 0 au shell s'il se termine normalement ;
- 1 s'il y a eu une erreur fatale de son fait (pas assez de mémoire, fichier introuvable) ;
- 2 si la connexion au serveur s'est interrompue ou arrêtée ;
- 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` a été initialisée.

4.5.7 Formatage des résultats

- Sortie simplifiée pour exploitation automatisée : `-XAt`
  - `-t` (`--tuples-only`)
  - `-A` (`--no-align`)
  - `-X` (`--no-psqlrc`)
  - séparateurs : `-F` (`--field-separator`) et `-R` (`--record-separator`)
- Formats HTML ou CSV
  - `-H | --html`
  - `--csv` (à partir de la version 12)

`psql` peut servir à afficher des rapports basiques et possède quelques options de formatage.

L'option `--csv` suffit à répondre à ce besoin, à partir d'un client en version 12 au moins.

S'il faut définir plus finement le format, il existe des options. `-A` impose une sortie non alignée des données. En ajoutant `-t`, qui supprime l'entête, et `-X`, qui demande à ignorer le `.psqlrc`, la sortie peut être facilement exploitée par des outils classiques comme `awk` ou `sed` :

```
$ psql -XAt -c 'select datname, pg_database_size(datname) from pg_database'
postgres|87311139
powa|765977379
template1|9028387
template0|8593923
pgbench|166134563
```

Le séparateur `|` peut être remplacé par un autre avec `-F`, ou un octet nul avec `-z`, et le retour chariot de fin de ligne par une chaîne définie avec `-R`, ou un octet nul avec `-0`.

`-H` permet une sortie en HTML pour une meilleure lisibilité par un humain.

4.5.8 Résultats en pivot (tableau croisé)

- `\crosstabview [colV [colH [colD [colonnedetriH]]]]`
- Exécute la requête en tampon
  - au moins 3 colonnes

Par exemple, pour voir les différents types de clients connectés aux bases (clients système inclus), le résultat n'est pas très lisible :

```
=# \pset null ø

=# SELECT datname, backend_type, COUNT(*) as nb  FROM pg_stat_activity
   GROUP BY 1,2
   ORDER BY datname NULLS LAST, backend_type ;

 datname  |         backend_type          | nb
----------+-------------------------------+----
 pgbench  | client backend                |  2
 postgres | client backend                |  1
 powa     | powa                          |  1
 ø        | archiver                      |  1
 ø        | autovacuum launcher           |  1
 ø        | background writer             |  1
 ø        | checkpointer                  |  1
 ø        | logical replication launcher  |  1
 ø        | pg_wait_sampling collector    |  1
```

```
 ø        | walwriter                   |  1
(10 lignes)
```

On peut le reformater ainsi :

```
=# \crosstabview backend_type datname

         backend_type          | postgres | ø | powa | pgbench
-------------------------------+----------+---+------+---------
 client backend                |        2 |   |      |       1
 walwriter                     |          | 1 |      |
 autovacuum launcher           |          | 1 |      |
 logical replication launcher  |          | 1 |      |
 powa                          |          |   |    1 |
 background writer             |          | 1 |      |
 archiver                      |          | 1 |      |
 checkpointer                  |          | 1 |      |
(9 lignes)
```

4.5.9 Formatage dans les scripts SQL

- Donner un titre au résultat de la requête
  - `\pset title 'Résultat de la requête'`
- Formater le résultat
  - `\pset format html` (ou `csv`...)
- Diverses options peu utilisées

Il est possible de réaliser des modifications sur le format de sortie des résultats de requête directement dans le script SQL ou en mode interactif dans psql.

Afficher `\pset` permet de voir ces différentes options. La complétion automatique après `\pset` affiche les paramètres et valeurs possibles.

Par exemple, l'option `format` est par défaut à `aligned` mais possède d'autres valeurs :

```
=# \pset format <TAB>

aligned          csv              latex            troff-ms         wrapped
asciidoc         html             latex-longtable  unaligned
```

D'autres options existent, peu utilisées. La liste complète des options de formatage et leur description est disponible dans la documentation de la commande psql[21].

[21] https://docs.postgresql.fr/current/app-psql.html

4.5.10 Scripts & Crontab

- `cron`
  - Attention aux variables d'environnement !
- Ou tout ordonnanceur

La planification d'un script périodique s'effectue de préférence avec les outils du système, donc sous Unix avec `cron` ou une de ses variantes, même si n'importe quel ordonnanceur peut convenir.

Avec `cron`, il faut se rappeler qu'à l'exécution d'un script, l'environnement de l'utilisateur n'est pas initialisé, ou plus simplement, les fichiers de personnalisation (par ex. `.bashrc`) de l'environnement ne sont pas lus. Seule la valeur `$HOME` est initialisée. Un script fonctionnant parfaitement dans une session peut échouer une fois planifié. Il faut donc prévoir ce cas, initialiser les variables d'environnement requises de façon adéquate, et bien tester.

Par exemple, pour charger l'environnement de l'utilisateur :

```
#!/bin/bash

. ${HOME}/.bashrc

…
```

Rappelons que chaque utilisateur du système peut avoir ses propres crontab. L'utilisateur peut les visualiser avec la commande `crontab -l` et les éditer avec la commande `crontab -e`.

4.5.11 Exemple de script de sauvegarde

- Sauvegarder une base et classer l'archive (squelette) :

```
#!/bin/bash
# Paramètre : la base
t=$(mktemp)                       # fichier temporaire
pg_dump -Fc "$1" > $t             # sauvegarde
d=$(eval date +%d%m%y-%H%M%S)     # date
mv $t /backup/"${1}_${d}.dump" # déplacement
exit 0
```

- …et ajouter la gestion des erreurs !
- …et les surveiller

Il est délicat d'écrire un script fiable. Ce script d'exemple possède plusieurs problèmes potentiels si le paramètre (la base) manque, si la sauvegarde échoue, si l'espace disque manque dans `/tmp`, si le déplacement échoue, si la partition cible n'est pas montée…

Parmi les outils existants, nous évoquerons notamment pg_back lors des sauvegardes[22].

Par convention, un script doit renvoyer 0 s'il s'est déroulé correctement.

[22] https://dali.bo/i1_html

4.6 OUTILS GRAPHIQUES

- Outils graphiques d'administration
  - temBoard
  - pgAdminIII et pgAdmin 4
  - pgmodeler

Il existe de nombreux outils graphiques permettant d'administrer des bases de données PostgreSQL. Certains sont libres, d'autres propriétaires. Certains sont payants, d'autres gratuits. Ils ont généralement les mêmes fonctionnalités de base, mais vont se distinguer sur certaines fonctionnalités un peu plus avancées comme l'import et l'export de données.

Nous allons étudier ici plusieurs outils proposés par la communauté, temBoard, pgAdmin.

4.6.1 temBoard

- Adresse: https://labs.dalibo.com/temboard
- Licence: PostgreSQL
- Notes: Serveur sur Linux, client web

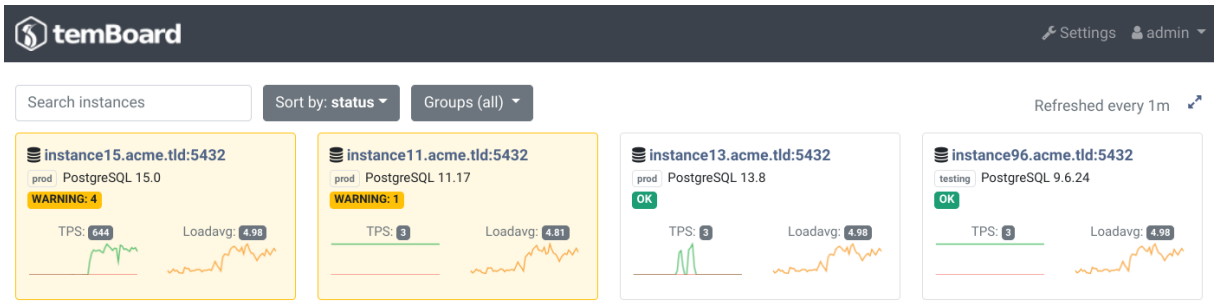
4.6.2 temBoard - PostgreSQL Remote Control

- Multi-instances
- Surveillance OS / PostgreSQL
- Suivi de l'activité
- Gestion des performances de PostgreSQL
- Configuration de chaque instance

temBoard est un outil permettant à un DBA de mener à bien la plupart de ses tâches courantes.

Le serveur web est installé de façon centralisée et un agent est déployé pour chaque instance.

4.6.3 temBoard - Vue parc



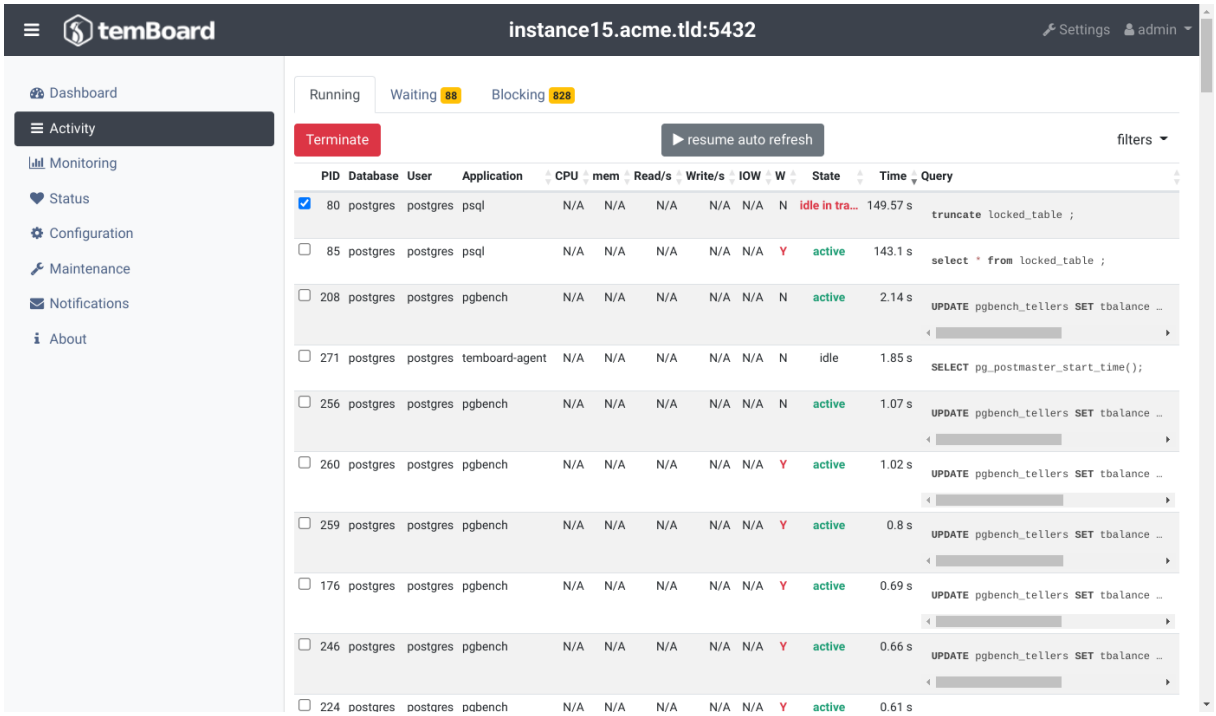
temboard fournit un tableau de bord global.

temBoard peut administrer plusieurs centaines d'instances.

4.6.4 temBoard - Tableau de bord

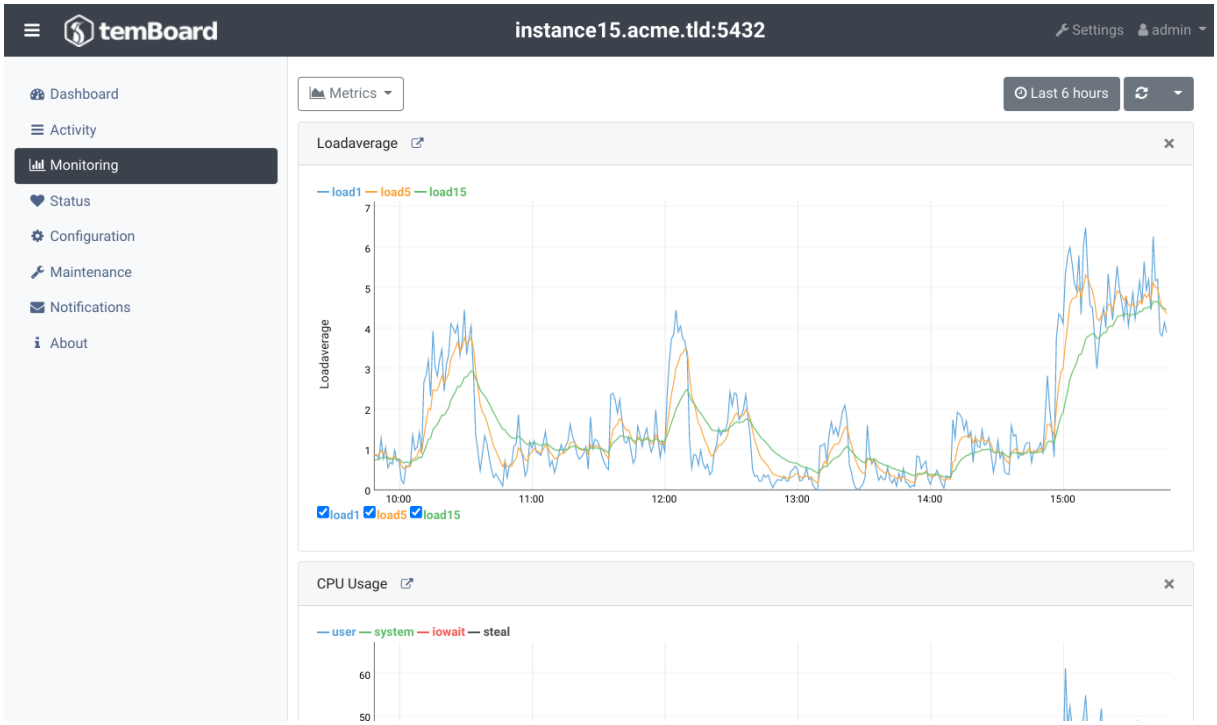
temboard présente un tableau de bord par instance.

4.6.5 temBoard - Activity



La section **Activity** permet de lister toutes les requêtes courantes (**Running**), les requêtes bloquées (**Waiting**) ou bloquantes (**Blocking**). Il est possible à partir de cette vue de terminer une session.

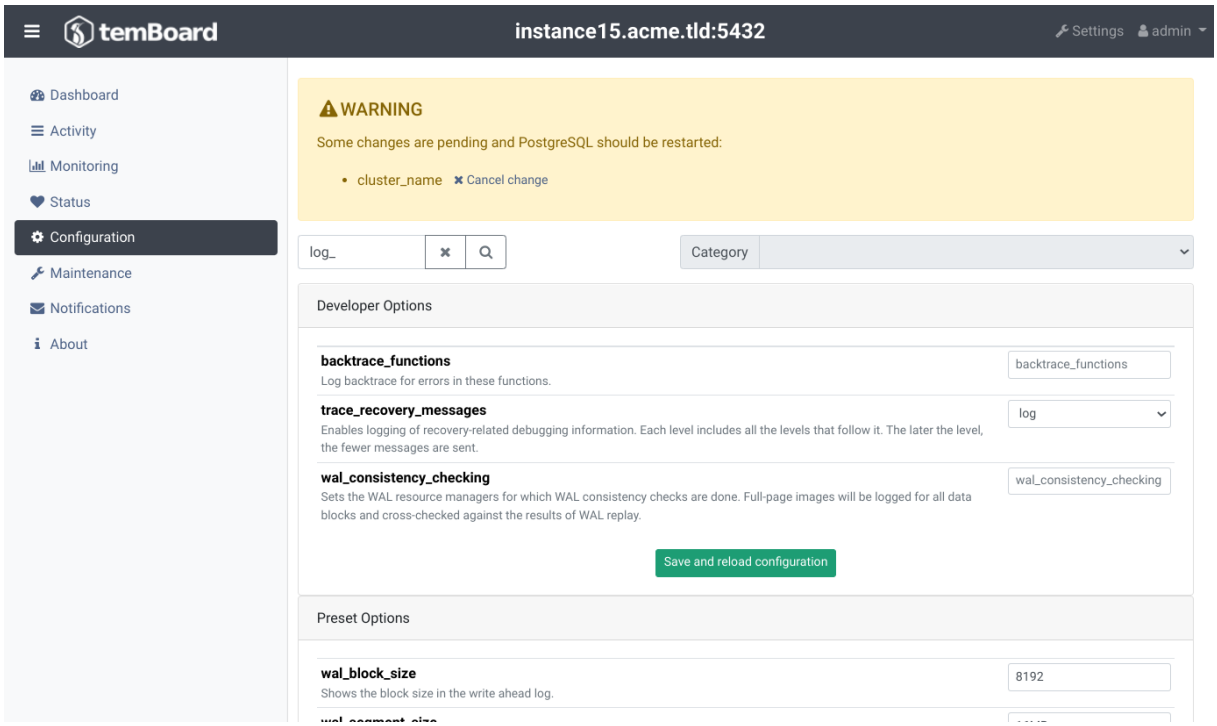
4.6.6 temBoard - Supervision



La section **Monitoring** permet de visualiser les graphiques historisés au niveau du système d'exploitation (CPU, mémoire, …) ainsi qu'au niveau de l'instance PostgreSQL.

temBoard inclut un système d'alerte par courriel et SMS lorsqu'une métrique dépasse un seuil.

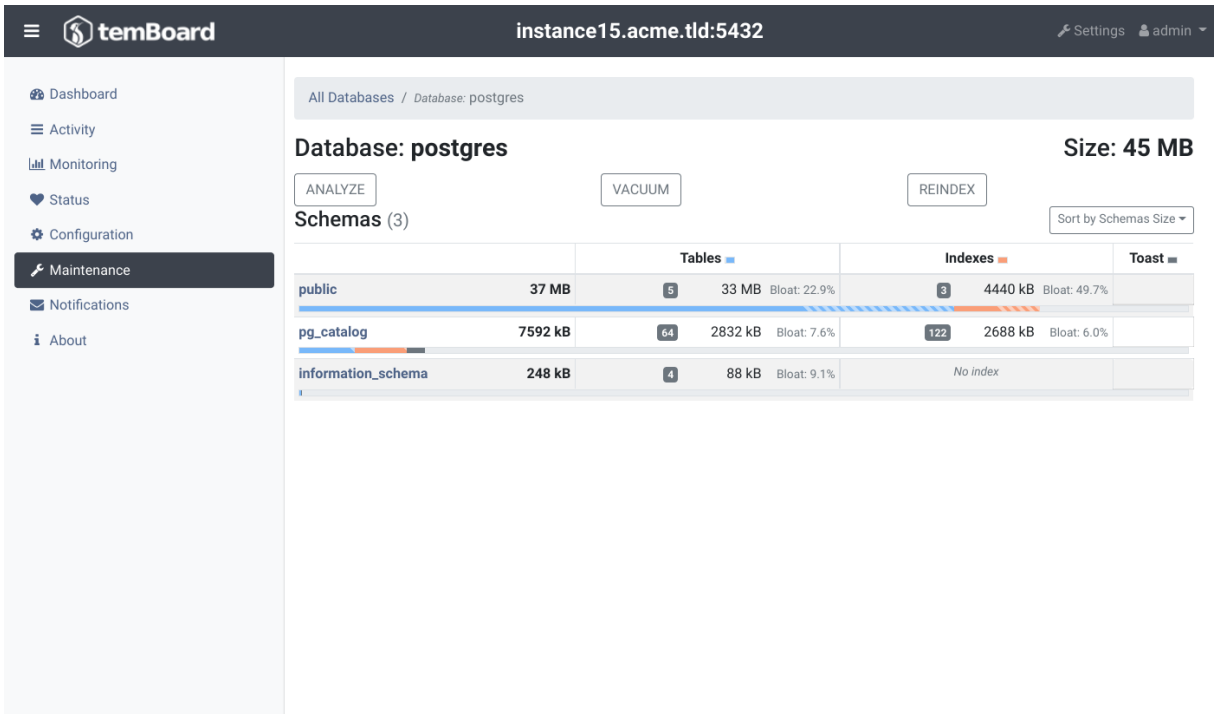
4.6.7 temBoard - Configuration



La section *Configuration* permet naviguer dans les paramètres de PostgreSQL de modifier ces paramètres

Suivant les cas, il sera proposé de recharger la configuration ou de redémarrer l'instance pour appliquer ces changements.

4.6.8 temBoard - Maintenance



temBoard estime la fragmentation du stockage des tables et des index.

Une interface permet de visualiser la fragmentation et d'agir dessus avec une granularité fine.

4.6.9 pgAdmin 4

- https://www.pgadmin.org
- Application web
- Licence : PostgreSQL
- Multiplateforme, multilangue

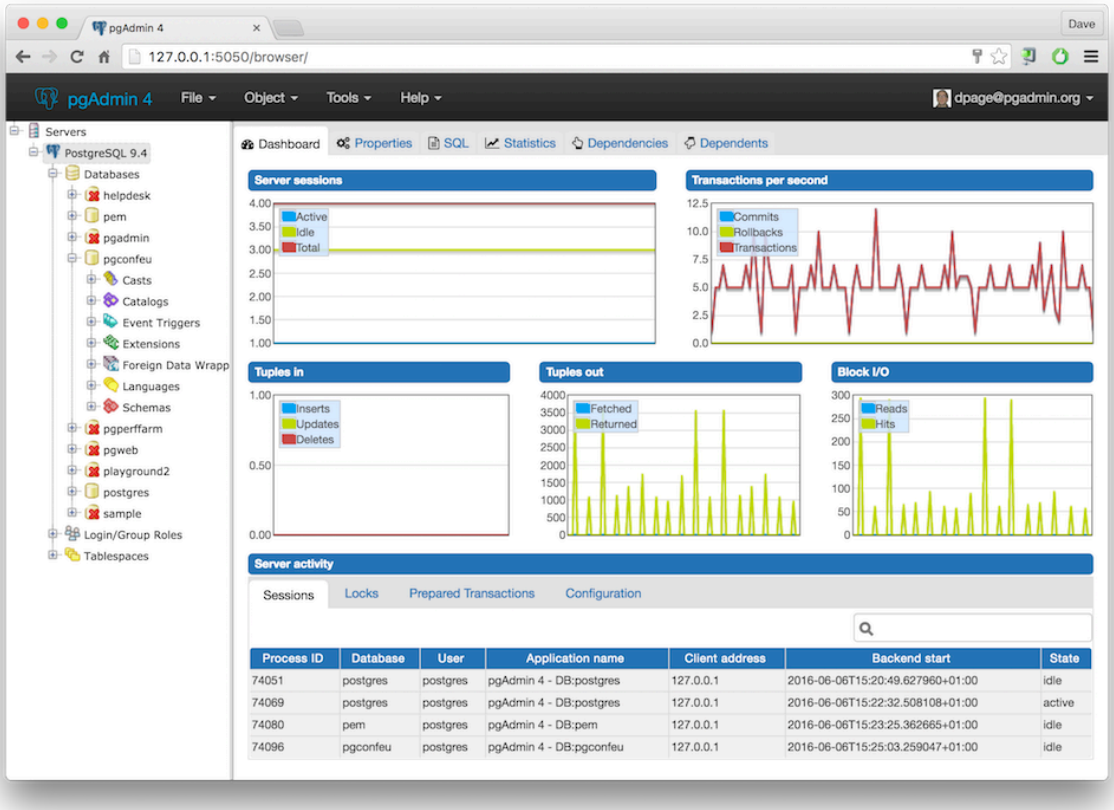
pgAdmin 4 est une application web, même si une version émulant un client lourd existe. Après un début difficile, le produit est à présent mature. Il reprend l'essentiel des fonctionnalités de pgAdmin III. Il est bien entendu compatible avec les dernières versions de PostgreSQL.

Il peut être déployé sur Windows et macOS X et bien sûr Linux, où il faudra utiliser les dépôts fournis par le projet[23], ou l'image docker.

Il est disponible sous licence PostgreSQL.

[23] https://www.pgadmin.org/download/

4.6.10 pgAdmin 4 : tableau de bord



Une des nouvelles fonctionnalités de pgAdmin 4 est l'apparition d'un tableau de bord remontant quelques métriques intéressantes. Il permet aussi la visualisation des géométries PostGIS, ce qui est parfois très utile.

4.6.11 DBeaver

- https://dbeaver.io
- Version Community sous Licence Apache 2.0
- Application Java Multiplateforme
- Version web *CloudBeaver* aussi disponible
- Supporte PostgreSQL (et ~ 80 autres SBGD)
- Version Pro payante et plus complète

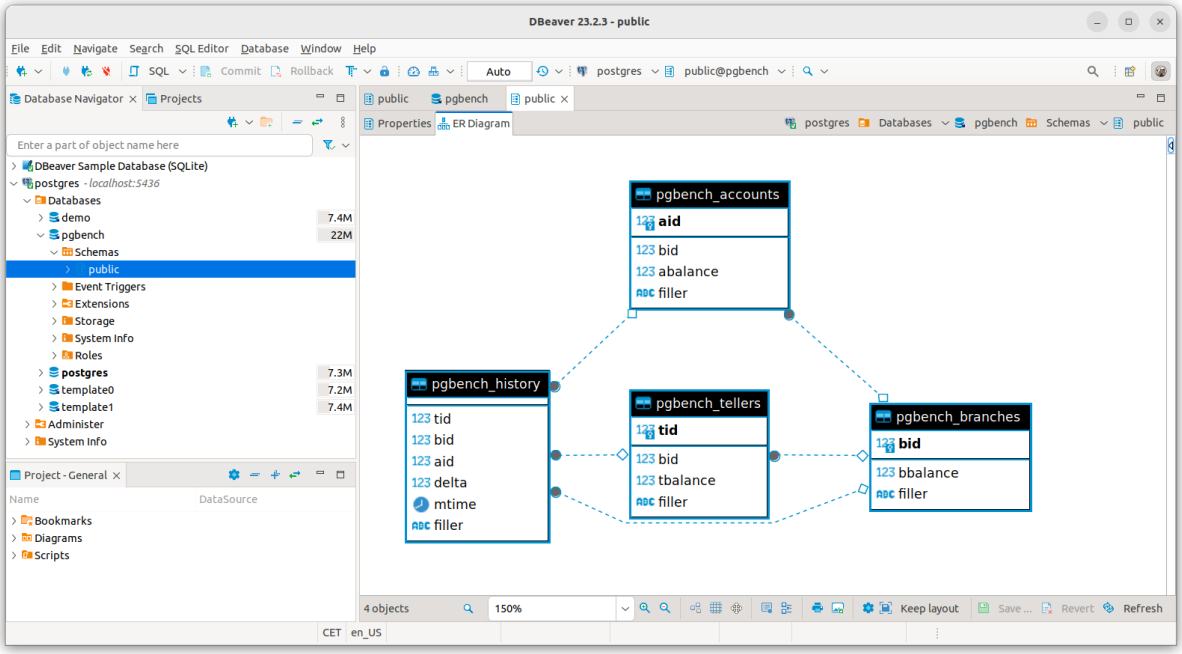
DBeaver est un outil graphique basé sur Eclipse. Il permet l'administration de plus de 80 SGBDs différents, y compris PostgreSQL. Le projet a sa propre entreprise depuis 2017 ainsi qu'une version payante plus complète. La version open-source *Community Edition* est disponible sous licence Apache 2.0. Une version web de DBeaver est disponible sous le nom de *CloudBeaver* (avec une page de démonstration publique[24]).

La version Pro, disponible sous différentes déclinaisons, propose des connecteurs propriétaires supplémentaires vers des systèmes de bases de données non relationnels (MongoDB, Redis, Cassandra, etc.), une gestion native des bases de données Cloud (AWS, Azure, etc.), des fonctionnalités dédiées aux entreprises ainsi qu'un support de la part de l'éditeur.

Il existe des paquets d'installation pour les systèmes d'exploitation les plus courants[25].

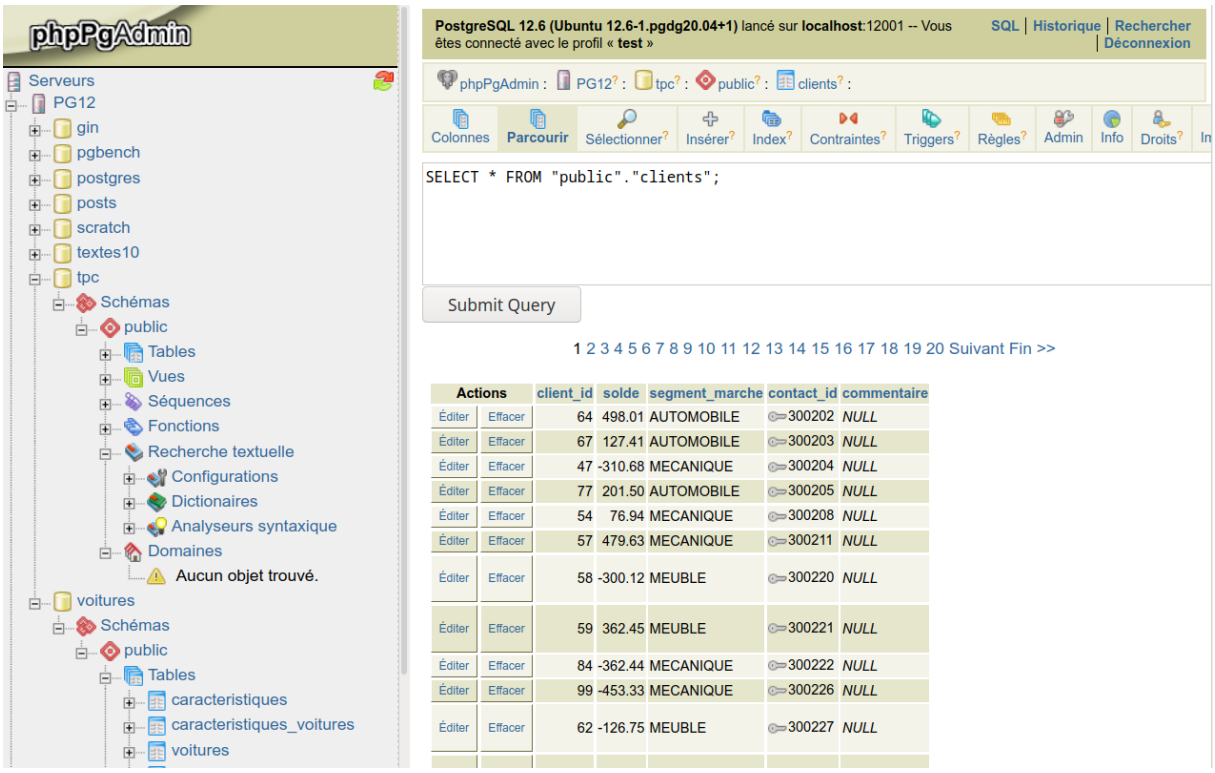
[24] https://demo.cloudbeaver.io/
[25] https://dbeaver.io/download/

4.6.12 DBeaver : fenêtre principale



La vue principale est divisée en deux sections, le côté gauche référence les différentes instances et le côté droit permet de consulter les différentes informations d'une instance en particulier. Ici, nous avons une vue entité-association des tables crées avec l'outil `pgbench` quand l'option `--foreign-keys` est spécifiée.

4.6.13 phpPgAdmin



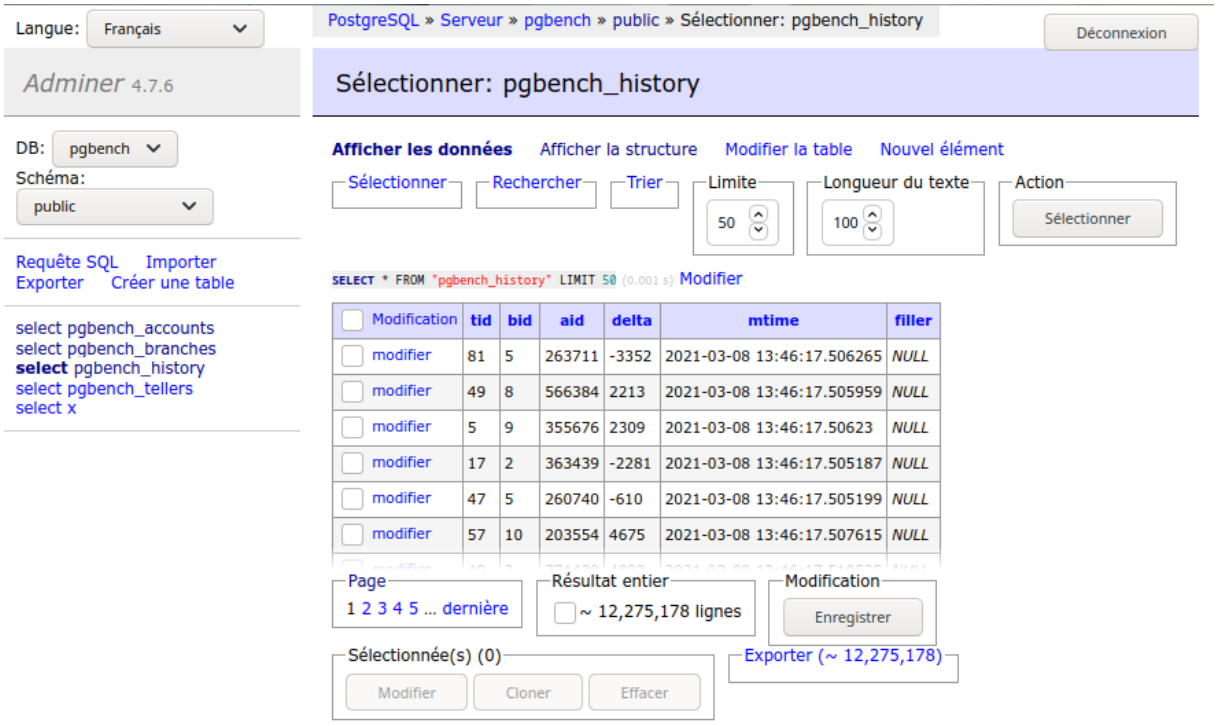
4.6.14 phpPgAdmin : fonctionnalités

- https://github.com/phppgadmin
- Licence: GNU Public License
- Application web, simple
  - consultation, édition
  - sauvegarde, export
- Pérennité ?

PhpPgAdmin est une application web en PHP, légère et simple d'emploi, que l'on peut éventuellement ouvrir à un simple utilisateur pour modifier des données.

Le projet PhpPgAdmin n'était plus maintenu pendant des années mais son principal développeur a décidé de reprendre la maintenance du projet. La version 7.13 se dit compatible jusqu'à la version 13 de PostgreSQL mais le partitionnement, par exemple, n'est pas géré. Notre conseil reste néanmoins de préférer la version web de pgAdmin 4 qui est plus lourd mais plus puissant et plus pérenne.

4.6.15 adminer



4.6.16 adminer : fonctionnalités

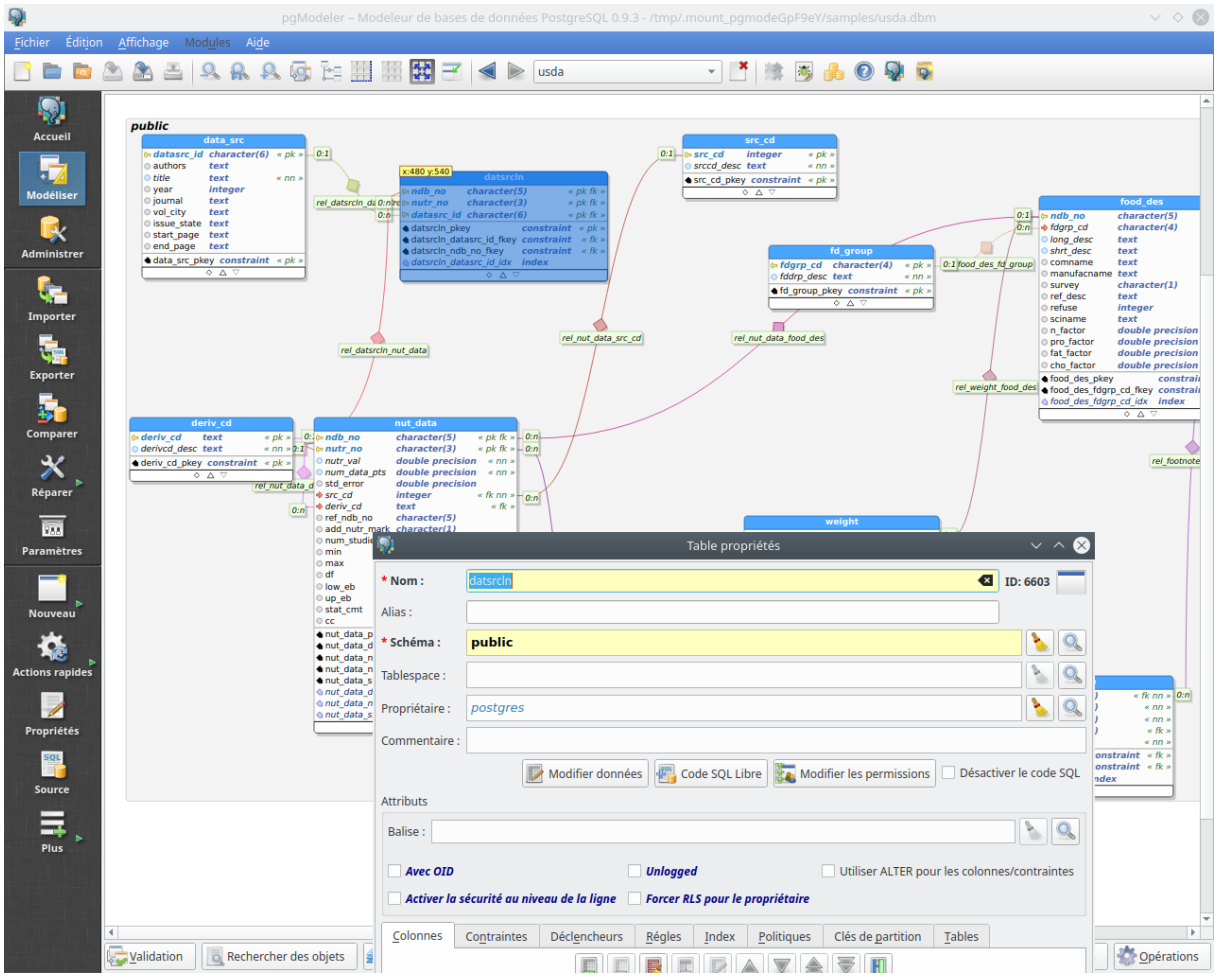
- https://www.adminer.org/
- Application web pour utilisateurs
- Basique mais simple & efficace
- Et simple : 1 fichier PHP
- Multibases, multilangues
- Licence : Apache License ou GPL 2

Adminer est une application web à destination des utilisateurs, pouvant gérer plusieurs types de bases, dont PostgreSQL.

Il consiste en un unique fichier PHP (éventuellement personnalisable par CSS).

Son interface peut sembler datée, voire primitive, mais elle est très simple et regroupe efficacement l'essentiel des fonctionnalités. C'est un candidat au remplacement de phpPgAdmin.

4.6.17 pgModeler



4.6.18 pgModeler

- Site officiel : https://pgmodeler.io/
- Licence : GPLv3
- Modélisation de base de données
- Fonctionnalité d'import export
- Comparaison de base

pgModeler permet de modéliser une base de données. Son intérêt par rapport à d'autres produits concurrents est qu'il est spécialisé pour PostgreSQL. Il en supporte donc toutes les spécificités,

comme l'héritage de tables, les types composites, les types tableaux… C'est une excellente solution pour modéliser une base en partant de zéro, ou pour extraire une visualisation graphique d'une base existante.

Il est à noter que, bien que le projet soit libre, son installation par les sources peut être laborieuse, et les paquets ne sont pas forcément disponibles. L'équipe de développement propose des paquets binaires à prix modique.

4.7 CONCLUSION

- Les outils en ligne de commande sont « rustiques » mais puissants
- Ils peuvent être remplacés par des outils graphiques
- En cas de problème, il est essentiel de les maîtriser.

4.7.1 Questions

N'hésitez pas, c'est le moment !

4.8 QUIZ

https://dali.bo/de_quiz

4.9 INTRODUCTION À PGBENCH

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

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[26].

4.9.1 Installation

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

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

```
/usr/pgsql-16/bin/pgbench
```

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

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebienc0mplexe';
CREATE DATABASE pgbench OWNER pgbench ;
```

Le `pg_hba.conf` doit éventuellement être adapté.

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

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

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

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

4.9.2 Générer de l'activité

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

[26] https://youtu.be/aTwh_CgRaE0

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

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

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

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

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

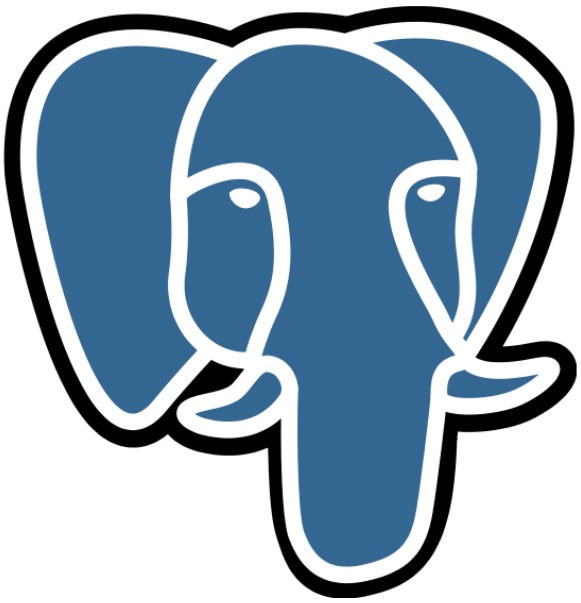
Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

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

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

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

5/ Tâches courantes



5.1 INTRODUCTION

- Gestion des bases
- Gestion des rôles
- Gestion des droits
- Tâches du DBA
- Sécurité

5.2 BASES

- Liste des bases
- Modèle (Template)
- Création
- Suppression
- Modification / configuration

Pour gérer des bases, il faut savoir les créer, les configurer et les supprimer. Il faut surtout comprendre qui a le droit de faire quoi, et comment. Ce chapitre détaille chacune des opérations possibles concernant les bases sur une instance.

5.2.1 Liste des bases

- Catalogue système : `pg_database`
- Commande `psql` :
  - `\l`
  - `\l+`

La liste des bases de données est disponible grâce à un catalogue système appelé `pg_database`. Il suffit d'un `SELECT` pour récupérer les méta-données sur chaque base :

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM pg_database \gx

-[ RECORD 1 ]--+----------------------------------
oid            | 5
datname        | postgres
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | f
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 8885214
datminmxid     | 1
```

```
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         |
-[ RECORD 2 ]--+-------------------------------------
oid            | 18770
datname        | cave
datdba         | 18769
encoding       | 6
datlocprovider | c
datistemplate  | f
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         |
-[ RECORD 3 ]--+-------------------------------------
oid            | 1
datname        | template1
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | t
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         | {=c/postgres,postgres=CTc/postgres}
-[ RECORD 4 ]--+-------------------------------------
oid            | 4
datname        | template0
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | t
datallowconn   | f
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
```

```
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion |
datacl         | {=c/postgres,postgres=CTc/postgres}
…
```

Voici la signification des principales colonnes :

- `oid` : identifiant système de la base ;
- `datname` : nom de la base ;
- `datdba` : l'identifiant de l'utilisateur propriétaire de cette base (voir l'OID correspondant à cet identifiant dans le catalogue système `pg_roles`) ;
- `encoding` : avec la fonction `pg_encoding_to_char()`, on peut voir que les valeurs 6 correspondent à UTF8, ce qui est généralement recommandé ;
- `datlocprovider` : `c` indique que les collations sont fournies par la biliothèque `libc` du système, un `i` indiquerait l'utilisation de la bibliothèque ICU (indépendante du système), et dans ce dernier cas les champs `daticulocale` et `daticurules` sont remplies ;
- `datistemplate` : à *true* si cette base est utilisable comme modèle ;
- `datallowconn` : à *true* s'il est autorisé de se connecter à cette base ;
- `datconnlimit` : limite du nombre de connexions simultanées pour les utilisateurs standards sur cette base (`0` interdit toute connexion, et `-1` ne pose pas de limite, jusque `max_connections` du moins) ;
- `datfrozenxid` : plus ancien identifiant de transaction géré par cette base (cela a une importance dans le recyclage des numéros de transaction) ;
- `dattablespace` : identifiant du tablespace par défaut de cette base (1663 indique généralement le tablespace `pg_default`, de fait le répertoire PGDATA ; les emplacements des tablespaces peuvent se trouver dans la table `pg_tablespace` ou par la fonction `pg_tablespace_location()`) ;
- `datacl` : droits pour cette base (un champ vide indique qu'il n'y a pas de droits spécifiques).

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\l` dans `psql` (vue raccourcie pour la mise en page) :

```
postgres=# \l
                                             List of databases
   Name     |   Owner   | Enc. | Locale P. |   Collate   |…|   Access privileges
------------+-----------+------+-----------+-------------+-+-----------------------
 cave       | caviste   | UTF8 | libc      | fr_FR.UTF-8 | |
 pgbench    | pgbench   | UTF8 | libc      | fr_FR.UTF-8 | |
 postgres   | postgres  | UTF8 | libc      | fr_FR.UTF-8 | |
 template0  | postgres  | UTF8 | libc      | fr_FR.UTF-8 | | =c/postgres          +
            |           |      |           |             | | postgres=CTc/postgres
 template1  | postgres  | UTF8 | libc      | fr_FR.UTF-8 | | =c/postgres          +
            |           |      |           |             | | postgres=CTc/postgres
 tpc        | tpc_owner | UTF8 | libc      | fr_FR.UTF-8 | |
…
```

Avec `\l+`, il est possible d'avoir plus d'informations (notamment la taille de la base ou le commentaire).

Noter que si l'on veut savoir où psql va chercher ces informations, il est possible de lui demander d'afficher la requête qu'il envoie au serveur :

```
$ psql --echo-hidden -c '\l+'

********* QUERY **********
SELECT
  d.datname as "Name",
  pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
  pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
  CASE d.datlocprovider WHEN 'c' THEN 'libc' WHEN 'i' THEN 'icu' END AS "Locale
 ↪  Provider",
  d.datcollate as "Collate",
  d.datctype as "Ctype",
  d.daticulocale as "ICU Locale",
  d.daticurules as "ICU Rules",
  pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges",
  CASE WHEN pg_catalog.has_database_privilege(d.datname, 'CONNECT')
       THEN pg_catalog.pg_size_pretty(pg_catalog.pg_database_size(d.datname))
       ELSE 'No Access'
  END as "Size",
  t.spcname as "Tablespace",
  pg_catalog.shobj_description(d.oid, 'pg_database') as "Description"
FROM pg_catalog.pg_database d
  JOIN pg_catalog.pg_tablespace t on d.dattablespace = t.oid
ORDER BY 1;
**************************
…
```

(suivent les résultats.)

La requête affichée montre bien que `psql` accède au catalogue `pg_database`, ainsi qu'à des fonctions système permettant d'éviter d'avoir à faire soi-même les jointures.

5.2.2 Modèle (template)

- Toute création de base se fait à partir d'un modèle
  - `template1`
- Permet de personnaliser sa création de base
- Mais il est aussi possible d'utiliser une autre base

Toute création de base se fait à partir d'un modèle. Par défaut, PostgreSQL utilise le modèle `template1`.

Tout objet ajouté dans le modèle est copié dans la nouvelle base. Cela concerne le schéma (la structure) comme les données. Il est donc intéressant d'ajouter des objets directement dans `template1` pour que ces derniers soient copiés dans les prochaines bases qui seront créées. Pour éviter malgré tout que cette base soit trop modifiée, il est possible de créer des bases qui seront ensuite utilisées comme modèle.

5.2.3 Création d'une base

- **SQL** : `CREATE DATABASE`
  - droit nécessaire: `SUPERUSER` ou `CREATEDB`
  - prérequis : base inexistante
- **Outil système** : `createdb`

L'ordre `CREATE DATABASE` est le seul moyen avec PostgreSQL de créer une base de données. Il suffit d'y ajouter le nom de la base à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `OWNER`, pour préciser le propriétaire de la base de données (si cette option n'est pas utilisée, le propriétaire est celui qui exécute la commande) ;
- `TEMPLATE`, pour indiquer le modèle à copier (par défaut `template1`) ;
- `ENCODING`, pour forcer un autre encodage que celui du serveur (à noter qu'il faudra utiliser le modèle `template0` dans ce cas) ;
- `LC_COLLATE` et `LC_CTYPE`, pour préciser respectivement l'ordre de tri des données textes et le jeu de caractères (par défaut, il s'agit de la locale utilisée lors de l'initialisation de l'instance) ;
- `STRATEGY` (depuis la version 15), pour indiquer la stratégie employée pour créer la base de donnée. Deux choix sont disponibles :
  - `FILE_COPY` : C'est la méthode historique, seule possible jusqu'en version 14 incluse. Le contenu des répertoires d'une base de données est intégralement copié pour initialiser la nouvelle base, avec juste une trace dans les journaux de transaction. Elle implique deux *checkpoints* parfois gênants, mais peut être intéressante pour copier de grosses bases en générant moins de journaux ;
  - `WAL_LOG` : C'est la méthode par défaut à partir de la version 15. L'opération est entièrement journalisée, et la liste des objets à copier et générée via le catalogue de PostgreSQL. Cette méthode évite les checkpoints et sécurise la copie en garantissant que toutes les opérations sont tracées, tout en évitant la copie accidentelle de fichier orphelins de la base modèle vers la base cible. Cette opération peut écrire beaucoup de journaux dans le cas où la base modèle est grosse, mais elle est idéale pour les créations de nouvelles bases presque vides ;

- `TABLESPACE`, pour stocker la base dans un autre tablespace que le répertoire des données ;
- `ALLOW_CONNECTIONS`, pour autoriser ou non les connexions à la base ;
- `CONNECTION LIMIT`, pour limiter le nombre de connexions d'utilisateurs standards simultanées sur cette base (illimité par défaut, tout en respectant le paramètre `max_connections`) ;
- `IS_TEMPLATE`, pour configurer ou non le mode template.

La copie se fait par clonage de la base de données modèle sélectionnée. Tous les objets et toutes les données faisant partie du modèle seront copiés sans exception. Par exemple, avant la 9.0, on ajoutait le langage PL/pgSQL dans la base de données `template1` pour que toutes les bases créées à partir de `template1` disposent directement de ce langage. Ce n'est plus nécessaire à partir de la 9.0 car le langage PL/pgSQL est activé dès la création de l'instance. Mais il est possible d'envisager d'autres usages de ce comportement (par exemple installer une extension ou une surcouche comme PostGIS dans chaque base).

À noter qu'il peut être nécessaire de sélectionner le modèle `template0` en cas de sélection d'un autre encodage que celui par défaut (comme la connexion est interdite sur `template0`, il y a peu de chances que des données textes avec un certain encodage aient été enregistrées dans cette base).

Voici l'exemple le plus simple de création d'une base :

```
CREATE DATABASE b1 ;
```

Cet ordre crée la base de données **b1**. Elle aura toutes les options par défaut. Autre exemple :

```
CREATE DATABASE b2 OWNER u1;
```

Cette commande SQL crée la base **b2** et s'assure que le propriétaire de cette base soit l'utilisateur **u1** (il faut que ce dernier existe au préalable).

Tous les utilisateurs n'ont pas le droit de créer une base de données. L'utilisateur qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEDB`. S'il utilise un autre modèle que celui par défaut, il doit être propriétaire de ce modèle ou le modèle doit être marqué comme étant un modèle officiel (autrement dit la colonne `datistemplate` doit être à `true`).

Voici un exemple complet :

```
postgres=# CREATE DATABASE b1;
CREATE DATABASE
postgres=# CREATE USER u1;
CREATE ROLE
postgres=# CREATE DATABASE b2 OWNER u1;
CREATE DATABASE
postgres=# CREATE USER u2 CREATEDB;
CREATE ROLE
```

NB : pour que la connexion qui suit fonctionne, et sans mot de passe, il faut paramétrer `pg_hba.conf` pour autoriser la connexion de cet utilisateur. Ce sera traité plus bas.

```
postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> CREATE DATABASE b3;

CREATE DATABASE

postgres=> CREATE DATABASE b4 TEMPLATE b2;
ERROR:  permission denied to copy database "b2"
postgres=> CREATE DATABASE b4 TEMPLATE b3;

CREATE DATABASE

postgres=> \c postgres postgres

You are now connected to database "postgres" as user "postgres".

postgres=# ALTER DATABASE b2 IS_TEMPLATE=true;

ALTER DATABASE

postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> CREATE DATABASE b5 TEMPLATE b2;

CREATE DATABASE

postgres=> \c postgres postgres
postgres=# \l

postgres=# \l
                                                List of databases
  Name    |  Owner   | Enc… |   Collate   |    Ctype    | … |   Access privileges
----------+----------+------+-------------+-------------+---+-----------------------
b1        | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b2        | u1       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b3        | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b4        | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b5        | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
postgres  | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
template0 | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   | =c/postgres          +
          |          |      |             |             |   | postgres=CTc/postgres
template1 | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   | =c/postgres          +
          |          |      |             |             |   | postgres=CTc/postgres
```

L'outil système `createdb` se connecte à la base de données `postgres` et exécute la commande `CREATE DATABASE`, exactement comme ci-dessus. Appelée sans aucun argument, `createdb` crée une base de donnée portant le nom de l'utilisateur connecté (si cette dernière n'existe pas). L'option `--echo` de cette commande permet de voir exactement ce que `createdb` exécute :

```
$ createdb --echo --owner u1 b6

SELECT pg_catalog.set_config('search_path', '', false)
CREATE DATABASE b6 OWNER u1;
```

Avec une configuration judicieuse des traces de PostgreSQL (`log_min_duration_statement = 0`, `log_connections = on`, `log_disconnections = on`), il est possible de voir cela complètement du point de vue du serveur :

```
[unknown] - LOG:  connection received: host=[local]
[unknown] - LOG:  connection authorized: user=postgres database=postgres
createdb - LOG:  duration: 1.018 ms
                 statement: SELECT pg_catalog.set_config('search_path', ''Z, false)
createdb - CREATE DATABASE b6 OWNER u1;
createdb - LOG:  disconnection: session time: 0:00:00.277 user=postgres
                                                          database=postgres
                                                          host=[local]
```

5.2.4 Suppression d'une base

- **SQL** : `DROP DATABASE`
  - droit nécessaire : `SUPERUSER` ou propriétaire de la base
  - prérequis : aucun utilisateur connecté sur la base
  - ou déconnexion forcée (v13)
- **Outil système** : `dropdb`

Supprimer une base de données supprime tous les objets et toutes les données contenues dans la base. La destruction d'une base de données ne peut pas être annulée.

La suppression se fait uniquement avec l'ordre `DROP DATABASE`. Seuls les superutilisateurs et le propriétaire d'une base peuvent supprimer cette base. Cependant, pour que cela fonctionne, il faut qu'aucun utilisateur ne soit connecté à cette base. Si quelqu'un est connecté, un message d'erreur apparaîtra :

```
postgres=# DROP DATABASE b6;

ERROR:  database "b6" is being accessed by other users
DETAIL:  There are 1 other session(s) using the database.
```

Il faut donc attendre que les utilisateurs se déconnectent, ou leur demander de le faire, voire les déconnecter autoritairement :

```
postgres=# SELECT pg_terminate_backend(pid)
       FROM pg_stat_activity
       WHERE datname='b6';

 pg_terminate_backend
----------------------
 t
```

```
postgres=# DROP DATABASE b6;

DROP DATABASE
```

Là-aussi, PostgreSQL propose un outil système appelé `dropdb` pour faciliter la suppression des bases. Cet outil se comporte comme `createdb`. Il se connecte à la base `postgres` et exécute l'ordre SQL correspondant à la suppression de la base :

```
$ dropdb --echo b5

SELECT pg_catalog.set_config('search_path', '', false)
DROP DATABASE b5;
```

Contrairement à `createdb`, sans nom de base, `dropdb` ne fait rien.

À partir de la version 13, il est possible d'utiliser la clause `WITH (FORCE)` de l'ordre `DROP DATABASE` ou l'option en ligne de commande `--force` de l'outil `dropdb` pour forcer la déconnexion des utilisateurs.

5.2.5 Modification / configuration

- `ALTER DATABASE`
  - pour modifier quelques méta-données
  - pour ajouter, modifier ou supprimer une configuration
- Catalogue système `pg_db_role_setting`

Avec la commande `ALTER DATABASE`, il est possible de modifier quelques méta-données :

- le nom de la base ;
- son propriétaire ;
- la limite de connexions ;
- le tablespace de la base.

Dans le cas d'un changement de nom ou de tablespace, aucun utilisateur ne doit être connecté à la base pendant l'opération.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour une base de données en utilisant la syntaxe suivante :

```
ALTER DATABASE base SET paramètre TO valeur;
```

La configuration spécifique de chaque base de données surcharge toute configuration reçue sur la ligne de commande du processus `postgres` père ou du fichier de configuration `postgresql.conf`.

L'ajout d'une configuration avec `ALTER DATABASE` sauvegarde le paramétrage mais ne l'applique pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
b1=# SHOW work_mem;

 work_mem
----------
 4MB

b1=# ALTER DATABASE b1 SET work_mem TO '2MB';

ALTER DATABASE

b1=# SHOW work_mem;

 work_mem
----------
 4MB

b1=# \c b1

You are now connected to database "b1" as user "postgres".

b1=# SHOW work_mem;

 work_mem
----------
 2MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf`, mais dans un catalogue système appelé `pg_db_role_setting` :

```
b1=# ALTER DATABASE b2 SET work_mem TO '32MB';

ALTER DATABASE

b1=# ALTER USER u1 SET maintenance_work_mem TO '256MB';

ALTER ROLE

b1=# SELECT * FROM pg_db_role_setting;

 setdatabase | setrole |          setconfig
-------------+---------+-----------------------------
       16384 |       0 | {work_mem=2MB}
       16386 |       0 | {work_mem=32MB}
           0 |   16385 | {maintenance_work_mem=256MB}

b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |        Configuration
------+-------------+-----------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
      | u1          | {maintenance_work_mem=256MB}
```

```
b1=# ALTER DATABASE b3 SET work_mem to '10MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET maintenance_work_mem to '128MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET random_page_cost to 3;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |                         Configuration
------+-------------+-------------------------------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB,random_page_cost=3}
      | u1          | {maintenance_work_mem=256MB}
```

Pour annuler la configuration d'un paramètre, utilisez :

```
ALTER DATABASE base RESET paramètre;
```

Par exemple :

```
b1=# ALTER DATABASE b3 RESET random_page_cost;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |              Configuration
------+-------------+-----------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
```

Si vous copiez avec `CREATE DATABASE … TEMPLATE` une base dont certains paramètres ont été configurés spécifiquement pour elle, ces paramètres ne sont pas appliqués à la nouvelle base de données.

5.3 RÔLES

- Utilisateur/groupe
- Liste des rôles
- Création
- Suppression
- Modification
- Gestion des mots de passe

Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est conçu et configuré. Les rôles peuvent être propriétaires d'objets de la base de données (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler l'accès à ces objets. De plus, il est possible de donner l'appartenance d'un rôle à un autre rôle, l'autorisant ainsi à utiliser les droits affectés au rôle dont il est membre.

Nous allons voir dans cette partie comment gérer les rôles, en allant de leur création à leur suppression, en passant par leur configuration.

5.3.1 Utilisateurs et groupes

- « Rôle » = utilisateurs et groupes
- Ordres SQL
  - `CREATE/DROP/ALTER USER`
  - `CREATE/DROP/ALTER GROUP`

Les rôles sont disponibles depuis la version 8.1. Auparavant, PostgreSQL avait la notion d'utilisateur et de groupe. Pour conserver la compatibilité avec les anciennes applications, les ordres SQL pour les utilisateurs et les groupes ont été conservés. Il est donc toujours possible de les utiliser mais il est actuellement conseillé de passer par les ordres SQL pour les rôles.

5.3.2 Liste des rôles

- Catalogue système : `pg_roles`
- Dans `psql` : `\du`

La liste des rôles est disponible grâce à un catalogue système appelé `pg_roles`. Il suffit d'un SELECT pour récupérer les méta-données sur chaque rôle :

```
postgres=# \x

Expanded display is on.

postgres=# SELECT * FROM pg_roles LIMIT 3;

-[ RECORD 1 ]--+----------------------
rolname        | postgres
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | t
rolconfig      |
oid            | 10
-[ RECORD 2 ]--+----------------------
rolname        | pg_monitor
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      |
oid            | 3373
-[ RECORD 3 ]--+----------------------
rolname        | pg_read_all_settings
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
```

```
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      |
oid            | 3374
```

Voici la signification des différentes colonnes :

- `rolname`, le nom du rôle ;
- `rolsuper`, le rôle a-t-il l'attribut `SUPERUSER` ? ;
- `rolinherit`, le rôle hérite-t-il automatiquement des droits des rôles dont il est membre ? ;
- `rolcreaterole`, le rôle a-t-il le droit de créer des rôles ? ;
- `rolcreatedb`, le rôle a-t-il le droit de créer des bases ? ;
- `rolcanlogin`, le rôle a-t-il le droit de se connecter ? ;
- `rolreplication`, le rôle peut-il être utilisé dans une connexion de réplication ? ;
- `rolconnlimit`, limite du nombre de connexions simultanées pour ce rôle (`0` indiquant « pas de connexions possibles », `-1` permet d'indiquer qu'il n'y a pas de limite en dehors de la valeur du paramètre `max_connections`) ;
- `rolpassword`, mot de passe du rôle (non affiché) ;
- `rolvaliduntil`, date limite de validité du mot de passe ;
- `rolbypassrls`, le rôle court-circuite-t-il les droits sur les lignes ? ;
- `rolconfig`, configuration spécifique du rôle ;
- `oid`, identifiant système du rôle.

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\du` dans `psql` :

```
postgres=# \du

List of roles
-[ RECORD 1 ]-------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 2 ]-------------------------------------------------------
Role name  | u1
Attributes |
Member of  | {}
-[ RECORD 3 ]-------------------------------------------------------
Role name  | u2
Attributes | Create DB
Member of  | {}
```

Il est à noter que les rôles systèmes ne sont pas affichés. Les rôles systèmes sont tous ceux commençant par `pg_`.

La métacommande `\du` ne fait qu'accéder aux tables systèmes. Par exemple :

```
$ psql -E postgres
```

```
psql (13.0)
Type "help" for help.

postgres=# \du
********* QUERY **********
SELECT r.rolname, r.rolsuper, r.rolinherit,
  r.rolcreaterole, r.rolcreatedb, r.rolcanlogin,
  r.rolconnlimit, r.rolvaliduntil,
  ARRAY(SELECT b.rolname
        FROM pg_catalog.pg_auth_members m
        JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid)
        WHERE m.member = r.oid) as memberof
, r.rolreplication
, r.rolbypassrls
FROM pg_catalog.pg_roles r
WHERE r.rolname !~ '^pg_'
ORDER BY 1;
**************************

List of roles
-[ RECORD 1 ]-------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
[...]
```

La requête affichée montre bien que `psql` accède aux catalogues `pg_roles` et `pg_auth_members`.

5.3.3 Création d'un rôle

- **SQL** : `CREATE ROLE`
  - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
  - prérequis : utilisateur inexistant
- **Outil système** : `createuser`
  - attribut `LOGIN` par défaut

L'ordre `CREATE ROLE` est le seul moyen avec PostgreSQL de créer un rôle. Il suffit d'y ajouter le nom du rôle à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `SUPERUSER`, pour que le nouveau rôle soit superutilisateur (autrement dit, ce rôle a le droit de tout faire une fois connecté à une base de données) ;
- `CREATEDB`, pour que le nouveau rôle ait le droit de créer des bases de données ;

- `CREATEROLE`, pour que le nouveau rôle ait le droit de créer un rôle ;
- `INHERIT`, pour que le nouveau rôle hérite automatiquement des droits des rôles dont il est membre ;
- `LOGIN`, pour que le nouveau rôle ait le droit de se connecter ;
- `REPLICATION`, pour que le nouveau rôle puisse se connecter en mode réplication ;
- `BYPASSRLS`, pour que le nouveau rôle puisse ne pas être vérifié pour les sécurités au niveau ligne ;
- `CONNECTION LIMIT`, pour limiter le nombre de connexions simultanées pour ce rôle ;
- `PASSWORD`, pour préciser le mot de passe de ce rôle ;
- `VALID UNTIL`, pour indiquer la date limite de validité du mot de passe ;
- `IN ROLE`, pour indiquer à quel rôle ce rôle appartient ;
- `IN GROUP`, pour indiquer à quel groupe ce rôle appartient ;
- `ROLE`, pour indiquer les membres de ce rôle ;
- `ADMIN`, pour indiquer les membres de ce rôles (les nouveaux membres ayant en plus la possibilité d'ajouter d'autres membres à ce rôle) ;
- `USER`, pour indiquer les membres de ce rôle ;
- `SYSID`, pour préciser l'identifiant système, mais est ignoré.

Par défaut, un rôle n'a aucun attribut (ni superutilisateur, ni le droit de créer des rôles ou des bases, ni la possibilité de se connecter en mode réplication, ni la possibilité de se connecter).

Voici quelques exemples simples :

```
postgres=# CREATE ROLE u3;

CREATE ROLE

postgres=# CREATE ROLE u4 CREATEROLE;

CREATE ROLE

postgres=# CREATE ROLE u5 LOGIN IN ROLE u2;

CREATE ROLE

postgres=# CREATE ROLE u6 ROLE u5;

CREATE ROLE

postgres=# \du

List of roles
-[ RECORD 1 ]-------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 2 ]-------------------------------------------------------
Role name  | u1
Attributes |
Member of  | {}
-[ RECORD 3 ]-------------------------------------------------------
Role name  | u2
```

```
Attributes | Create DB
Member of  | {}
-[ RECORD 4 ]-----------------------------------------------------------
Role name  | u3
Attributes | Cannot login
Member of  | {}
-[ RECORD 5 ]-----------------------------------------------------------
Role name  | u4
Attributes | Create role, Cannot login
Member of  | {}
-[ RECORD 6 ]-----------------------------------------------------------
Role name  | u5
Attributes |
Member of  | {u2,u6}
-[ RECORD 7 ]-----------------------------------------------------------
Role name  | u6
Attributes | Cannot login
Member of  | {}
```

Tous les rôles n'ont pas le droit de créer un rôle. Le rôle qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEROLE`. Un utilisateur qui a l'attribut `CREATEROLE` pourra créer tout type de rôles sauf des superutilisateurs.

Voici un exemple complet :

```
postgres=# CREATE ROLE u7 LOGIN CREATEROLE;

CREATE ROLE

postgres=# \c postgres u7

You are now connected to database "postgres" as user "u7".

postgres=> CREATE ROLE u8 LOGIN;

CREATE ROLE

postgres=> CREATE ROLE u9 LOGIN CREATEDB;

CREATE ROLE

postgres=> CREATE ROLE u10 LOGIN SUPERUSER;

ERROR:  must be superuser to create superusers

postgres=> \du

                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 u1        |                                                            | {}
 u2        | Create DB                                                  | {}
 u3        | Cannot login                                               | {}
 u4        | Create role, Cannot login                                  | {}
 u5        |                                                            | {u2,u6}
 u6        | Cannot login                                               | {}
 u7        | Create role                                                | {}
```

```
 u8        |                                                    | {}
 u9        | Create DB                                          | {}
```

Il est toujours possible d'utiliser les ordres SQL `CREATE USER` et `CREATE GROUP`. PostgreSQL les comprend comme étant l'ordre `CREATE ROLE`. Dans le premier cas (`CREATE USER`), il ajoute automatiquement l'option `LOGIN`.

Il est possible de créer un utilisateur (dans le sens, rôle avec l'attribut `LOGIN`) sans avoir à se rappeler de la commande SQL. Le plus simple est certainement l'outil `createuser`, livré avec PostgreSQL, mais c'est aussi possible avec n'importe quel autre outil d'administration de bases de données PostgreSQL.

L'outil système `createuser` se connecte à la base de données `postgres` et exécute la commande `CREATE ROLE`, exactement comme ci-dessus, avec par défaut l'option `LOGIN`. L'option `--echo` de cette commande nous permet de voir exactement ce que `createuser` exécute :

```
$ createuser --echo u10 --superuser
```

```
SELECT pg_catalog.set_config('search_path', '', false)
CREATE ROLE u10 SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

Il est à noter que `createuser` est un programme interactif. Avant la version 9.2, si le nom du rôle n'est pas indiqué, l'outil demandera le nom du rôle à créer. De même, si au moins un attribut n'est pas explicitement indiqué, il demandera les attributs à associer à ce rôle :

```
$ createuser u11
```

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
```

Depuis la version 9.2, il crée un utilisateur avec les valeurs par défaut (équivalent à une réponse `n` à toutes les questions). Pour retrouver le mode interactif, il faut utiliser l'option `--interactive`.

5.3.4 Suppression d'un rôle

- **SQL** : `DROP ROLE`
  - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
  - prérequis : rôle existant, rôle ne possédant pas d'objet
- **Outil système** : `dropuser`

La suppression d'un rôle se fait uniquement avec l'ordre `DROP ROLE`. Seuls les utilisateurs disposant des attributs `SUPERUSER` et `CREATEROLE` peuvent supprimer des rôles. Cependant, pour que cela

fonctionne, il faut que le rôle à supprimer ne soit pas propriétaire d'objets dans l'instance. S'il est propriétaire, un message d'erreur apparaîtra :

```
postgres=> DROP ROLE u1;

ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  owner of database b2
```

Il faut donc changer le propriétaire des objets en question ou supprimer les objets. Vous pouvez utiliser respectivement les ordres `REASSIGN OWNED` et `DROP OWNED` pour cela.

Un rôle qui n'a pas l'attribut `SUPERUSER` ne peut pas supprimer un rôle qui a cet attribut :

```
postgres=> DROP ROLE u10;

ERROR:  must be superuser to drop superusers
```

Par contre, il est possible de supprimer un rôle qui est connecté. Le rôle connecté aura des possibilités limitées après sa suppression. Par exemple, il peut toujours lire quelques tables systèmes mais il ne peut plus créer d'objets.

Là-aussi, PostgreSQL propose un outil système appelé `dropuser` pour faciliter la suppression des rôles. Cet outil se comporte comme `createrole` : il se connecte à la base PostgreSQL et exécute l'ordre SQL correspondant à la suppression du rôle :

```
$ dropuser --echo u10

SELECT pg_catalog.set_config('search_path', '', false)
DROP ROLE u10;
```

Sans spécifier le nom de rôle sur la ligne de commande, `dropuser` demande le nom du rôle à supprimer.

5.3.5 Modification d'un rôle

- `ALTER ROLE`
  - pour modifier quelques méta-données
  - pour ajouter, modifier ou supprimer une configuration
- Catalogue système : `pg_db_role_setting`

Avec la commande `ALTER ROLE`, il est possible de modifier quelques méta-données du rôle :

- son nom ;
- son mot de passe ;
- sa limite de validité ;

- ses attributs :
  - `SUPERUSER` ;
  - `CREATEDB` ;
  - `CREATEROLE` ;
  - `CREATEUSER` ;
  - `INHERIT` ;
  - `LOGIN` ;
  - `REPLICATION` ;
  - `BYPASSRLS` .

Toutes ces opérations peuvent s'effectuer alors que le rôle est connecté à la base.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour un rôle en utilisant la syntaxe suivante :

```
ALTER ROLE rôle SET paramètre TO valeur;
```

La configuration spécifique de chaque rôle surcharge toute configuration reçue sur la ligne de commande du processus postgres père ou du fichier de configuration `postgresql.conf`, mais aussi la configuration spécifique de la base de données où le rôle est connecté. L'ajout d'une configuration avec `ALTER ROLE` sauvegarde le paramétrage mais ne l'applique pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les rôles peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
$ psql -U u2 postgres

psql (13.0)
Type "help" for help.

postgres=> SHOW work_mem;

 work_mem
----------
 4MB

postgres=> ALTER ROLE u2 SET work_mem TO '20MB';

ALTER ROLE

postgres=> SHOW work_mem;

 work_mem
----------
 4MB

postgres=> \c - u2

You are now connected to database "postgres" as user "u2".

postgres=> SHOW work_mem;
```

```
 work_mem
----------
 20MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf` mais dans un catalogue système appelé `pg_db_role_setting` :

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;

 Base | Utilisateur |               Configuration
------+-------------+--------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
```

Il est aussi possible de configurer un paramétrage spécifique pour un utilisateur et une base donnés :

```
postgres=# ALTER ROLE u2 IN DATABASE b1 SET work_mem to '10MB';

ALTER ROLE

postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> SHOW work_mem;

 work_mem
----------
 20MB

postgres=> \c b1 u2

You are now connected to database "b1" as user "u2".

b1=> SHOW work_mem;

 work_mem
----------
 10MB

b1=> \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SHOW work_mem;

 work_mem
----------
 2MB
```

```
b1=> \c postgres u1

You are now connected to database "postgres" as user "u1".

postgres=> SHOW work_mem;

 work_mem
----------
 4MB
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;

 Base | Utilisateur |                Configuration
------+-------------+-----------------------------------------
 b1   | u2          | {work_mem=10MB}
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
```

Pour annuler la configuration d'un paramètre pour un rôle, utilisez :

```
ALTER ROLE rôle RESET paramètre;
```

Attention : la prise en compte de ces options dans les sauvegardes est un point délicat. Il est détaillé dans notre module de formation sur les sauvegardes logiques.

Après sa création, il est toujours possible d'ajouter et de supprimer un rôle dans un autre rôle. Pour cela, il est possible d'utiliser les ordres `GRANT` et `REVOKE` :

```
GRANT role_groupe TO role_utilisateur;
```

Il est aussi possible de passer par la commande `ALTER GROUP` de cette façon :

```
ALTER GROUP role_groupe ADD USER role_utilisateur;
```

5.3.6 Mot de passe

Le mot de passe ne concerne pas toutes les méthodes d'authentification

- Par défaut : l'utilisateur n'a pas de mot de passe
  - donc pas de connexion possible
- Modification :

```
ALTER ROLE u1 PASSWORD 'supersecret';  -- dangereux
```

- Attention à ne pas l'afficher dans les traces
  - fournir un mot de passe déjà chiffré
  - utiliser `\password` ou `createuser`

```
createuser --login --echo --pwprompt u1
```

Certaines méthodes d'authentification n'ont pas besoin de mot de passe (comme `peer` pour les connexions depuis le serveur même), ou délèguent l'authentification à un système extérieur (comme `ldap`). Par défaut, les utilisateurs n'ont pas de mot de passe. Si la méthode en exige un, ils ne pourront pas se connecter.

Il est très fortement conseillé d'utiliser une méthode d'authentification avec saisie du mot de passe.

On peut le créer ainsi :

```
ALTER ROLE u1 PASSWORD 'supersecret';
```

À partir de là, avec une méthode d'authentification bien configurée, le mot de passe sera demandé. Il faudra, dans cet exemple, saisir « supersecret » pour que la connexion se fasse.

ATTENTION ! Le mot de passe peut apparaître en clair dans les traces ! Notamment si `log_min_duration_statement` vaut 0.

```
$ grep PASSWORD $PGDATA/log/traces.log

psql - LOG: duration: 1.865 ms
       statement: ALTER ROLE u1 PASSWORD 'supersecret';
```

La vue système `pg_stat_activity` ou l'extension `pg_stat_statements`, voire d'autres outils, sont aussi susceptibles d'afficher la commande et donc le mot de passe en clair.

Il est donc essentiel de s'arranger pour que seules des personnes de confiance aient accès aux traces et vues systèmes. Il est certes possible de demander temporairement la désactivation des traces pendant le changement de mot de passe (si l'on est superutilisateur) :

```
SET log_min_duration_statement TO -1;
SET log_statement TO none;
ALTER ROLE u1 PASSWORD 'supersecret';

$ grep PASSWORD $PGDATA/log/postgresql-Mon.log
[rien]
```

Cependant, cela ne règle pas le cas de `pg_stat_statements` et `pg_stat_activity`.

De manière générale, il est donc chaudement conseillé de ne renseigner que des mots de passe chiffrés. C'est très simple en mode interactif avec `psql`, la métacommande `\password` opère le chiffrement :

```
\password u1

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

L'ordre effectivement envoyé au serveur et éventuellement visible dans les traces sera :

```
ALTER USER u1 PASSWORD 'SCRAM-SHA-256$4096:KcHoLSZE…Hj81r3w='
```

De même si on crée le rôle depuis le shell avec `createuser` :

```
createuser --login --echo --pwprompt u1

Saisir le mot de passe pour le nouveau rôle :
Le saisir de nouveau :
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE u1 PASSWORD
 ↪  'SCRAM-SHA-256$4096:/LVaGESxmDwyF92urT…hS0kOxIpwWAbTpW1i9peGIfg='
NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN NOREPLICATION NOBYPASSRLS;
```

Lors des changements de mots de passe, ne pas oublier qu'il reste un risque de fuite aussi au niveau des outils système, par exemple l'historique du shell, l'affichage de la liste des processus ou les traces système !

5.3.7 Sécurité des mots de passe

```
password_encryption = "scram-sha-256"
```

- SCRAM-SHA-256 : défaut (≥ v14)
- MD5 : dépassé, mais défaut jusque v13 !
- Sécurité :
  - date limite sur le mot de passe (pas le rôle)
  - pas de vérification de la force du mot de passe
  - pas de limite de tentatives échouées (tester `fail2ban`)
  - itérations SCRAM (v16)

Le mot de passe est chiffré en interne, et visible dans les sauvegardes avec `pg_dumpall -g`, ou dans la vue système `pg_authid`.

Il existe deux méthodes de chiffrement : SCRAM-SHA-256 (fortement conseillée) et l'historique MD5. Elles sont éventuellement utilisables simultanément pour des utilisateurs différents.

L'exemple suivant montre que la méthode de chiffrement peut différer selon les rôles, en fonction de la valeur du paramètre `password_encryption` au moment de la mise en place du mot de passe :

```
SET password_encryption TO "scram-sha-256";

CREATE ROLE u12 LOGIN PASSWORD 'supersecret';

SELECT * FROM pg_authid  WHERE rolname IN ('u1', 'u12') ORDER BY 1;

-[ RECORD 1 ]--+------------------------------------------------------------
rolname        | u1
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
```

```
rolpassword    | md5fb75f17111cea61e62b54ab950dd1268
rolvaliduntil  |
-[ RECORD 2 ]--+------------------------------------------------------------
rolname        | u12
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    | SCRAM-SHA-256$4096:0/uC6oDNuQWO8H9pMaVg8g==$nDUpGSefHOZMd
                 TcbWR13NPELJubGg7PduiJjX/Hyt/M=:PSUzE+rP5g4f6mb5sFDRq/Hds
                 OrLvfYew9ZIdzO/GDw=
rolvaliduntil  |
```

Chiffrement SCRAM-SHA-256 :

SCRAM-SHA-256 n'est la méthode par défaut que depuis PostgreSQL 14, bien que disponible depuis PostgreSQL 10. Elle est plus complexe et plus sûre que l'ancienne méthode MD5. Notamment, le même mot de passe entré plusieurs fois pour un même utilisateur, même sur la même instance, donnera des versions chiffrées à chaque fois différentes, mais interchangeables.

Un chiffrement SCRAM-SHA-256 est de la forme :

```
SCRAM-SHA-256$<sel>:<nombre d'itérations>$<hash>
```

Pour quelques détails d'implémentation et une comparaison avec MD5, voir par exemple cette présentation de Jonathan Katz[1].

Générer soi-même des mots de passe chiffrés en SCRAM-SHA-256 en-dehors de `psql` est plus compliqué qu'avec MD5, et les outils comme `\password` s'appuient souvent sur les fonctions fournies par la bibliothèque `libpq`. Cette dernière sert aussi de base à la bibliothèque python psycopg3[2] et sa fonction PGconn.encrypt_password()[3]. Indépendamment de la `libpq`, il existe aussi un script python de Jonathan Katz[4] (version 3.6 minimum).

Depuis la version 16 il est possible d'ajuster le nombre d'itérations avec le paramètre `scram_iterations`. Le défaut de 4096 est un compromis. Monter plus haut permet de lutter contre les attaques par force brute. Réduire ce paramètre permet de parer aux problèmes de performances depuis certains terminaux peu puissants, ou en cas de connexions très fréquentes (le début de la discussion entre les développeurs[5] illustre bien ces deux contraintes).

Chiffrement MD5 :

Le chiffrement `md5` est celui par défaut jusque PostgreSQL 13 inclus. Il consiste à calculer la somme MD5 du mot de passe concaténé au nom du rôle (pour que deux utilisateurs de même mot de

[1] https://fr.slideshare.net/jkatz05/safely-protect-postgresql-passwords-tell-others-to-scram
[2] https://github.com/psycopg/psycopg
[3] https://www.psycopg.org/psycopg3/docs/api/pq.html#psycopg.pq.PGconn.encrypt_password
[4] https://gist.github.com/jkatz/e0a1f52f66fa03b732945f6eb94d9c21#file-encypt_password-py-L20
[5] https://www.postgresql.org/message-id/F72E7BC7-189F-4B17-BF47-9735EB72C364@yesql.se

passe n'ait pas la même version chiffrée) ; puis « md5 » est ajouté devant. De manière plus générale, l'algorithme MD5 est considéré comme trop faible de nos jours.

Un autre problème de sécurité est que la version chiffrée d'un mot de passe est identique sur deux instances différentes pour un même nom d'utilisateur, ce qui ouvre la possibilité d'attaques par *rainbow tables*[6].

Un inconvénient plus mineur du chiffrement MD5 est qu'il utilise le nom de l'utilisateur : en cas de changement de ce nom, il faudra de nouveau rentrer le mot de passe pour que son stockage chiffré soit correct.

Pour éviter de fournir un mot de passe en clair à PostgreSQL, il est facile de le chiffrer en MD5 avant de le fournir à PostgreSQL :

```
$ echo -n "supersecretu1" | md5sum
fb75f17111cea61e62b54ab950dd1268  -

ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Dans les traces on ne trouvera que la version chiffrée :

```
psql - LOG:  duration: 2.100 ms  statement: ALTER ROLE u1
                                 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Ceci fait la même chose sous forme de script :

```
set +o history    # suspend l'historique du shell
MDP=supersecret
U=u1
psql -X --echo-all -c \
"$(echo ALTER ROLE ${U} PASSWORD \'md5$(echo -n ${MDP}${U}|md5sum|cut -f1 -d' '))';"

…
ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
…
```

Cohabitation de mots de passe SCRAM-SHA-256 et MD5 :

Lors de la mise à jour d'une ancienne instance, il n'est pas forcément possible de ré-entrer immédiatement tous les mots de passe existants chiffrés en MD5. De plus, certains outils clients un peu anciens pourraient ne pas supporter SCRAM-SHA-256 (voir le wiki[7]). Changer la méthode d'authentification selon l'utilisateur est donc utile.

La méthode d'authentification SCRAM-SHA-256 doit rester le défaut, et tout mot de passe réentré sera chiffré en SCRAM-SHA-256 :

```
password_encryption = "scram-sha-256"
```

Si le mot de passe est stocké au format SCRAM-SHA-256, une authentification paramétrée sur `md5` ou `password` dans le fichier `pg_hba.conf` fonctionnera aussi. Cela facilite la migration progressive des utilisateurs de `md5` à `scram-sha-256`, qui peuvent réentrer leur mot de passe quand ils veulent.

[6] https://fr.wikipedia.org/wiki/Rainbow_table
[7] https://wiki.postgresql.org/wiki/List_of_drivers

Par contre, indiquer la méthode `scram-sha-256` dans `pg_hba.conf` impose un chiffrage en SCRAM-SHA-256 et interdit d'utiliser MD5.

On peut mixer les deux méthodes dans `pg_hba.conf` si le besoin se fait sentir, par exemple pour n'utiliser `md5` que pour une seule application (avec un compte dédié) avec un ancien client :

```
# vieille application (avant le cas général)
host    compta  mathusalem   192.168.66.66/32    md5
# authentification habituelle
host    all     all          192.168.66.0/24     scram-sha-256
```

Dates de validité des mots de passe :

Les mots de passe ont une date de validité mais pas les rôles eux-mêmes. Par exemple, il reste possible de se connecter en local par la méthode `peer` même si le mot de passe a expiré.

Sécurité des mots de passe :

Enfin, il est à noter que PostgreSQL ne vérifie pas la faiblesse d'un mot de passe. Il est certes possible d'installer une extension appelée `passwordcheck` (voir sa documentation[8]).

```
postgres=# ALTER ROLE u1 PASSWORD 'supersecret';

ERROR:  password must contain both letters and nonletters
```

Il est possible de modifier le code source de cette extension pour y ajouter les règles convenant à votre cas, ou d'utiliser la bibliothèque `Cracklib`. Des extensions de ce genre, extérieures au projet, existent. Cependant, ces outils exigent que le mot de passe soit fourni en clair, et donc sujet à une fuite (dans les traces par exemple), ce qui, répétons-le, est fortement déconseillé !

Un rôle peut tenter de se connecter autant de fois qu'il le souhaite, ce qui expose à des attaques de type force brute. Il est possible d'interdire toute connexion à partir d'un certain nombre de connexions échouées si vous utilisez une méthode d'authentification externe qui le gère (comme PAM, LDAP ou Active Directory). Vous pouvez aussi obtenir cette fonctionnalité en utilisant un outil comme `fail2ban`. Sa configuration est détaillée dans notre base de connaissances[9].

[8] https://docs.postgresql.fr/current/passwordcheck.html
[9] https://kb.dalibo.com/fail2ban

5.4 DROITS SUR LES OBJETS

- Droits sur les objets
- Droits sur les méta-données
- Héritage des droits
- Changement de rôle

Pour bien comprendre l'intérêt des utilisateurs, il faut bien comprendre la gestion des droits. Les droits sur les objets vont permettre aux utilisateurs de créer des objets ou de les utiliser. Les commandes `GRANT` et `REVOKE` sont essentielles pour cela. Modifier la définition d'un objet demande un autre type de droit, que les commandes précédentes ne permettent pas d'obtenir.

Donner des droits à chaque utilisateur peut paraître long et difficile. C'est pour cela qu'il est généralement préférable de donner des droits à une entité spécifique dont certains utilisateurs hériteront.

5.4.1 Droits sur les objets

- Donner un droit :

```
GRANT USAGE ON SCHEMA unschema TO utilisateur ;
GRANT SELECT,DELETE,INSERT ON TABLE matable TO utilisateur ;
```

- Retirer un droit :

```
REVOKE UPDATE ON TABLE matable FROM utilisateur ;
```

- Droits spécifiques pour chaque type d'objets :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA …
ALTER DEFAULT PRIVILEGES FOR ROLE …
```

- Avoir le droit de donner le droit :

```
WITH GRANT OPTION
```

- Groupe implicite : `public`
- Schéma par défaut : `public` lisible par tous (≤ 14) !

```
REVOKE ALL ON SCHEMA public FROM public ;
```

Par défaut, seul le propriétaire a des droits sur son objet. Les superutilisateurs n'ont pas de droit spécifique sur les objets mais étant donné leur statut de superutilisateur, ils peuvent tout faire sur tous les objets.

Le propriétaire d'un objet peut décider de donner certains droits sur cet objet à certains rôles. Il le fera avec la commande `GRANT` :

```
GRANT droits ON type_objet nom_objet TO role
```

Les droits disponibles dépendent du type d'objet visé. Par exemple, il est possible de donner le droit `SELECT` sur une table mais pas sur une fonction. Une fonction ne se lit pas, elle s'exécute. Il est donc possible de donner le droit `EXECUTE` sur une fonction.

La liste complète des droits figure dans la documentation officielle[10].

Il faut donner les droits aux différents objets séparément. De plus, donner le droit `ALL` sur une base de données donne tous les droits sur la base de données, autrement dit l'objet base de donnée, pas sur les objets à l'intérieur de la base de données. `GRANT` n'est pas une commande récursive. Prenons un exemple :

```
b1=# CREATE ROLE u20 LOGIN;

CREATE ROLE

b1=# CREATE ROLE u21 LOGIN;

CREATE ROLE

b1=# \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> CREATE SCHEMA s1;

ERROR:  permission denied for database b1

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT CREATE ON DATABASE b1 TO u20;

GRANT

b1=# \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> CREATE SCHEMA s1;

CREATE SCHEMA

b1=> CREATE TABLE s1.t1 (c1 integer);

CREATE TABLE
```

[10] https://docs.postgresql.fr/current/sql-grant.html

```
b1=> INSERT INTO s1.t1 VALUES (1), (2);

INSERT 0 2

b1=> SELECT * FROM s1.t1;

 c1
----
  1
  2

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> GRANT SELECT ON TABLE s1.t1 TO u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> GRANT USAGE ON SCHEMA s1 TO u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

 c1
----
  1
  2

b1=> INSERT INTO s1.t1 VALUES (3);

ERROR:  permission denied for relation t1
```

Le problème de ce fonctionnement est qu'il faut indiquer les droits pour chaque utilisateur, ce qui peut devenir difficile et long. Imaginez avoir à donner le droit `SELECT` sur les 400 tables d'un schéma… Il est néanmoins possible de donner les droits sur tous les objets d'un certain type dans un schéma. Voici un exemple :

```
GRANT SELECT ON ALL TABLES IN SCHEMA s1 to u21;
```

Notez aussi que, lors de la création d'une base, PostgreSQL ajoute automatiquement un schéma nommé `public`. Avant la version 15, tous les droits sont donnés sur ce schéma à un pseudo-rôle, lui aussi appelé `public`, et dont tous les rôles existants et à venir sont membres d'office. À partir de la version 15, le schéma `public` appartient au propriétaire de la base et aucun droit par défaut n'est donné aux autres utilisateurs.

Avec une version antérieure à la version 15, n'importe quel utilisateur peut donc, par défaut, créer des tables dans le schéma `public` de toute base où il peut se connecter (mais ne peut lire les tables créées là par d'autres, sans droit supplémentaire) !

Dans une logique de sécurisation, avant la version 15, il faut donc penser à enlever les droits à `public`. Une fausse bonne idée est de tout simplement supprimer le schéma `public`, ou de le récréer (par défaut, sans droits pour le groupe `public`). Cependant, une sauvegarde logique serait restaurée dans une base qui, par défaut, aurait à nouveau un schéma `public` ouvert à tous. Une révocation explicite des droits se retrouvera par contre dans une sauvegarde :

```
REVOKE ALL ON SCHEMA public FROM public ;
```

(Noter la subtilité de syntaxe : `GRANT... TO...` et `REVOKE... FROM...`)

Nombre de scripts et outils peuvent tomber en erreur sans ces droits. Il faudra remonter cela aux auteurs en tant que bugs.

Cette modification peut être faite aussi dans la base `template1` (qui sert de modèle à toute nouvelle base), sur toute nouvelle instance.

Enfin il est possible d'ajouter des droits pour des objets qui n'ont pas encore été créés. En fait, la commande `ALTER DEFAULT PRIVILEGES` permet de donner des droits par défaut à certains rôles. De cette façon, sur un schéma qui a tendance à changer fréquemment, il n'est plus nécessaire de se préoccuper des droits sur les objets.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO public ;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT INSERT ON TABLES TO utilisateur ;
```

Lorsqu'un droit est donné à un rôle, par défaut, ce rôle ne peut pas le donner à un autre. Pour lui donner en plus le droit de donner ce droit à un autre rôle, il faut utiliser la clause `WITH GRANT OPTION` comme le montre cet exemple :

```
b1=# CREATE TABLE t2 (id integer);

CREATE TABLE

b1=# INSERT INTO t2 VALUES (1);

INSERT 0 1

b1=# SELECT * FROM t2;

 id
----
  1

b1=# \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SELECT * FROM t2;

ERROR:  permission denied for relation t2

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT SELECT ON TABLE t2 TO u1;

GRANT

b1=# \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SELECT * FROM t2;

 id
----
  1

b1=> \c b1 u2

You are now connected to database "b1" as user "u2".

b1=> SELECT * FROM t2;

ERROR:  permission denied for relation t2

b1=> \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> GRANT SELECT ON TABLE t2 TO u2;

WARNING:  no privileges were granted for "t2"
GRANT

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT SELECT ON TABLE t2 TO u1 WITH GRANT OPTION;
```

```
GRANT

b1=# \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> GRANT SELECT ON TABLE t2 TO u2;

GRANT

b1=> \c b1 u2

You are now connected to database "b1" as user "u2".

b1=> SELECT * FROM t2;

 id
----
  1
```

5.4.2 Afficher les droits

- Colonne `*acl` sur les tables systèmes
  - `datacl` pour `pg_database`
  - `relacl` pour `pg_class`
- Codage `role1=xxxx/role2` (format `aclitem`)
  - `role1` : rôle concerné par les droits
  - `xxxx` : droits parmi `xxxx`
  - `role2` : rôle qui a donné les droits
- Sous `psql` :
  - `\dp` et `\z`
  - `df`, `\dconfig+` etc..

Les colonnes `*acl` des catalogues systèmes indiquent les droits sur un objet. Leur contenu est un codage au format `aclitem` indiquant le rôle concerné, ses droits, et qui lui a fourni ces droits (ou le propriétaire de l'objet, si celui qui a fourni les droits est un superutilisateur).

Les droits sont codés avec des lettres. Les voici avec leur signification :

- `r` pour la lecture (`SELECT`) ;

- `w` pour les modifications (`UPDATE`) ;
- `a` pour les insertions (`INSERT`) ;
- `d` pour les suppressions (`DELETE`) ;
- `D` pour la troncation (`TRUNCATE`) ;
- `x` pour l'ajout de clés étrangères ;
- `t` pour l'ajout de triggers ;
- `X` pour l'exécution de routines ;
- `U` pour l'utilisation (d'un schéma par exemple) ;
- `C` pour la création d'objets permanents (tables ou vues par exemple) ;
- `c` pour la connexion (spécifique aux bases de données) ;
- `T` pour la création d'objets temporaires (tables ou index temporaires) ;
- `s` pour la modification d'un paramètre superutilisateur avec `SET` ;
- `A` pour la modification d'un paramètre avec `ALTER SYSTEM`.

5.4.3 Droits sur les métadonnées

- Seul le propriétaire peut changer la structure d'un objet
  - le renommer
  - le changer de schéma ou de tablespace
  - lui ajouter/retirer des colonnes
- Un seul propriétaire
  - peut être un utilisateur ou un groupe
- `REASSIGN OWNER` / `DROP OWNED`

Les droits sur les objets ne concernent pas le changement des méta-données et de la structure de l'objet. Seul le propriétaire (et les superutilisateurs) peut le faire. S'il est nécessaire que plusieurs personnes puissent utiliser la commande `ALTER` sur l'objet, il faut que ces différentes personnes aient un rôle qui soit membre du rôle propriétaire de l'objet. Prenons un exemple :

```
b1=# \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;

ERROR:  must be owner of relation t1

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".
```

```
b1=> GRANT u20 TO u21;

GRANT ROLE

b1=> \du

                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 u1        |                                                            | {}
 u11       | Create DB                                                  | {}
 u12       |                                                            | {}
 u2        | Create DB                                                  | {}
 u20       |                                                            | {}
 u21       |                                                            | {u20}
 u3        | Cannot login                                               | {}
 u4        | Create role, Cannot login                                  | {}
 u5        |                                                            | {u2,u6}
 u6        | Cannot login                                               | {}
 u7        | Create role                                                | {}
 u8        |                                                            | {}
 u9        | Create DB                                                  | {}

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;

ALTER TABLE
```

Pour assigner un propriétaire différent aux objets ayant un certain propriétaire, il est possible de faire appel à l'ordre `REASSIGN OWNED`. De même, il est possible de supprimer tous les objets appartenant à un utilisateur avec l'ordre `DROP OWNED`. Voici un exemple de ces deux commandes :

```
b1=# \d

       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
 public | t2   | table | u21

b1=# REASSIGN OWNED BY u21 TO u1;

REASSIGN OWNED

b1=# \d

       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
 public | t2   | table | u1

b1=# DROP OWNED BY u1;
```

```
DROP OWNED
```

```
b1=# \d
```

```
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
```

5.4.4 Droits plus globaux 1/2

- Rôles systèmes d'administration
  - `pg_signal_backend`
  - `pg_database_owner` (14+)
  - `pg_checkpoint` (15+)
  - `pg_reserved_connections` (16+)
- Rôles systèmes de supervision
  - `pg_read_all_stats`
  - `pg_read_all_settings`
  - `pg_stat_scan_tables`
  - `pg_monitor`

Certaines fonctionnalités nécessitent l'attribut `SUPERUSER` alors qu'il serait bon de pouvoir les effectuer sans avoir ce droit suprême. Cela concerne principalement l'administration, la sauvegarde et la supervision.

Après beaucoup de discussions, les développeurs de PostgreSQL ont décidé de créer des rôles systèmes permettant d'avoir plus de droits. Le premier rôle de ce type est `pg_signal_backend` qui donne le droit d'exécuter les fonctions `pg_cancel_backend()` et `pg_terminate_backend()`, même en simple utilisateur sur des requêtes autres que les siennes :

```
postgres=# \c - u1
```

```
You are now connected to database "postgres" as user "u1".
```

```
postgres=> SELECT usename, pid FROM pg_stat_activity WHERE usename IS NOT NULL;
```

```
 usename |  pid
---------+-------
 u2      | 23194
 u1      | 23195
```

```
postgres=> SELECT pg_terminate_backend(23194);

ERROR:  must be a member of the role whose process is being terminated
        or member of pg_signal_backend

postgres=> \c - postgres

You are now connected to database "postgres" as user "postgres".

postgres=# GRANT pg_signal_backend TO u1;

GRANT ROLE

postgres=# \c - u1

You are now connected to database "postgres" as user "u1".

postgres=> SELECT pg_terminate_backend(23194);

 pg_terminate_backend
----------------------
 t

postgres=> SELECT usename, pid FROM pg_stat_activity WHERE usename IS NOT NULL;

 usename |  pid
---------+-------
 u1      | 23212
```

Par contre, les connexions des superutilisateurs ne sont pas concernées.

Parmi ces rôles, `pg_read_all_stats` permet de lire les tables de statistiques d'activité. `pg_read_all_settings` permet de lire la configuration de tous les paramètres. `pg_stat_scan_tables` permet d'exécuter les procédures stockées de lecture des statistiques. `pg_monitor` est le rôle typique pour de la supervision : il combine les trois rôles précédents. Leur utilisation est identique à `pg_signal_backend`.

En version 15 arrive le rôle `pg_checkpoint`. Ce dernier permet à un rôle non `SUPERUSER` de lancer un `CHECKPOINT`.

À partir de la version 16 apparaît un rôle `pg_use_reserved_connections`, qui donne accès à un pool de connexions réservées défini par le paramètre `reserved connections` (0 par défaut). Cela peut servir à garantir un accès à la supervision ou à divers utilitaires sans en faire des superutilisateurs.

5.4.5 Droits plus globaux 2/2

- Rôles d'accès aux fichiers
  - `pg_read_server_files`
  - `pg_write_server_files`
  - `pg_execute_server_program`
- Rôles d'accès aux données (14+)
  - `pg_read_all_data`
  - `pg_write_all_data`
- Autres
  - `pg_create_subscription` (16+)

`pg_read_server_files` permet d'autoriser la lecture de fichiers auxquels le serveur peut accéder avec la commande SQL `COPY` et toutes autres fonctions d'accès de fichiers. `pg_write_server_files` permet la même chose en écriture. Cela sous-entend aussi les sauvegardes de fichiers côté serveur, par exemple avec `pg_basebackup` (à partir de la version 15) ou d'autres outils. `pg_execute_server_program` autorise les utilisateurs membres d'exécuter des programmes en tant que l'utilisateur qui exécute le serveur PostgreSQL au travers de la commande SQL `COPY` et de toute fonction permettant l'exécution d'un programme sur le serveur.

La version 14 ajoute trois nouveaux rôles. `pg_read_all_data` permet de lire toutes les données des tables, vues et séquences, alors que `pg_write_all_data` permet de les écrire. Quant à `pg_database_owner`, il permet de se comporter comme le propriétaire des bases de données.

En version 16 apparaît `pg_create_subscription` pour gérer des souscriptions en réplication logique.

5.4.6 Héritage des droits

- Créer un rôle sans droit de connexion
- Donner les droits à ce rôle
- Placer les utilisateurs concernés comme membre de ce rôle

Plutôt que d'avoir à donner les droits sur chaque objet à chaque ajout d'un rôle, il est beaucoup plus simple d'utiliser le système d'héritage des droits.

Supposons qu'une nouvelle personne arrive dans le service de facturation. Elle doit avoir accès à toutes les tables concernant ce service. Sans utiliser l'héritage, il faudra récupérer les droits d'une autre personne du service pour retrouver la liste des droits à donner à cette nouvelle personne. De plus, si un nouvel objet est créé et que chaque personne du service doit pouvoir y accéder, il faudra ajouter l'objet et ajouter les droits pour chaque personne du service sur cet objet. C'est long et sujet à erreur. Il est préférable de créer un rôle facturation, de donner les droits sur ce rôle, puis d'ajouter chaque rôle du service facturation comme membre du rôle facturation. L'ajout et la suppression d'un objet est très simple : il suffit d'ajouter ou de retirer le droit sur le rôle facturation, et cela impactera tous les rôles membres.

Voici un exemple complet :

```
b1=# CREATE ROLE facturation;

CREATE ROLE

b1=# CREATE TABLE factures(id integer, dcreation date, libelle text,
montant numeric);

CREATE TABLE

b1=# GRANT ALL ON TABLE factures TO facturation;

GRANT

b1=# CREATE TABLE clients (id integer, nom text);

CREATE TABLE

b1=# GRANT ALL ON TABLE clients TO facturation;

GRANT

b1=# CREATE ROLE r1 LOGIN;

CREATE ROLE

b1=# GRANT facturation TO r1;

GRANT ROLE

b1=# \c b1 r1

You are now connected to database "b1" as user "r1".

b1=> SELECT * FROM factures;

 id | dcreation | libelle | montant
----+-----------+---------+---------
(0 rows)

b1=# CREATE ROLE r2 LOGIN;

CREATE ROLE

b1=# \c b1 r2
```

```
You are now connected to database "b1" as user "r2".
b1=> SELECT * FROM factures;
ERROR:  permission denied for relation factures
```

5.4.7 Changement de rôle

- Rôle par défaut
  - celui de la connexion
- Rôle emprunté :
  - après un `SET ROLE`
  - pour tout rôle dont il est membre

Par défaut, un utilisateur se connecte avec un rôle de connexion. Il obtient les droits et la configuration spécifique de ce rôle. Dans le cas où il hérite automatiquement des droits des rôles dont il est membre, il obtient aussi ces droits qui s'ajoutent aux siens. Dans le cas où il n'hérite pas automatiquement de ces droits, il peut temporairement les obtenir en utilisant la commande `SET ROLE`. Il ne peut le faire qu'avec les rôles dont il est membre.

```
b1=# CREATE ROLE r31 LOGIN;
CREATE ROLE
b1=# CREATE ROLE r32 LOGIN NOINHERIT IN ROLE r31;
CREATE ROLE
b1=# \c b1 r31
You are now connected to database "b1" as user "r31".
b1=> CREATE TABLE t1(id integer);
CREATE TABLE
b1=> INSERT INTO t1 VALUES (1), (2);
INSERT 0 2
b1=> \c b1 r32
You are now connected to database "b1" as user "r32".
b1=> SELECT * FROM t1;
ERROR:  permission denied for relation t1
```

```
b1=> SET ROLE TO r31;

SET

b1=> SELECT * FROM t1;

 id
----
  1
  2

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# ALTER ROLE r32 INHERIT;

ALTER ROLE

b1=# \c b1 r32

You are now connected to database "b1" as user "r32".

b1=> SELECT * FROM t1;

 id
----
  1
  2

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# REVOKE r31 FROM r32;

REVOKE ROLE

b1=# \c b1 r32

You are now connected to database "b1" as user "r32".

b1=> SELECT * FROM t1;

ERROR:  permission denied for relation t1

b1=> SET ROLE TO r31;

ERROR:  permission denied to set role "r31"
```

Le changement de rôle peut se faire uniquement au niveau de la transaction. Pour cela, il faut utiliser la clause `LOCAL`. Il peut se faire aussi sur la session, auquel cas il faut passer par la clause `SESSION`.

5.5 DROITS DE CONNEXION

- Lors d'une connexion, indication :
  - de l'hôte (socket Unix ou alias/adresse IP)
  - du nom de la base de données
  - du nom du rôle
  - du mot de passe (parfois optionnel)
- Selon les 3 premières informations
  - impose une méthode d'authentification

Lors d'une connexion, l'utilisateur fournit, explicitement ou non, plusieurs informations. PostgreSQL va choisir une méthode d'authentification en se basant sur les informations fournies et sur la configuration d'un fichier appelé `pg_hba.conf`.

5.5.1 Informations de connexion

Pour se connecter à une base, il faut :

- 4 informations
  - socket Unix ou adresse/alias IP
  - n° de port
  - nom de la base
  - nom du rôle
- Fournies explicitement
  - paramètres
  - environnement
- ou implicitement
  - environnement
  - défauts

Tous les outils fournis avec la distribution PostgreSQL (par exemple `createuser`) acceptent des options en ligne de commande pour fournir les informations en question :

- `-h` pour la socket Unix ou l'adresse/alias IP ;
- `-p` pour le numéro de port ;
- `-d` pour le nom de la base ;
- `-U` pour le nom du rôle.

Si l'utilisateur ne passe pas ces informations, plusieurs variables d'environnement sont vérifiées :

- `PGHOST` pour la socket Unix ou l'adresse/alias IP ;
- `PGPORT` pour le numéro de port ;
- `PGDATABASE` pour le nom de la base ;
- `PGUSER` pour le nom du rôle.

Au cas où ces variables ne seraient pas configurées, des valeurs par défaut sont utilisées :

- la socket Unix (`/var/run/postgresql`, parfois `/tmp`) en lieu d'un nom de machine ;
- le port 5432 ;
- la base `postgres` ou le nom de l'utilisateur PostgreSQL demandé, (suivant l'outil) ;
- le nom de l'utilisateur au niveau du système d'exploitation pour le nom du rôle.

Autrement dit, quelle que soit la situation, PostgreSQL remplacera les informations non fournies explicitement par des informations provenant des variables d'environnement, voire par des informations par défaut.

5.5.2 Configuration de l'authentification : pg_hba.conf

PostgreSQL utilise les informations de connexion pour choisir la méthode de connexion

- Fichier de configuration : `pg_hba.conf`
- Exemple :

```
local      DATABASE  USER  METHOD  [OPTIONS]
host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
hostnossl  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

- Clauses `include`, `include_if_exists`, `include_dir` (v16)
- Contrôle avec la vue `pg_hba_file_rules`

Lorsque le serveur PostgreSQL récupère une demande de connexion, il connaît le type de connexion utilisé par le client (socket Unix, connexion TCP SSL, connexion TCP simple, etc.). Il connaît aussi l'adresse IP du client (dans le cas d'une connexion via une socket TCP), le nom de la base et celui de l'utilisateur.

PostgreSQL va donc chercher les lignes correspondantes dans le tableau enregistré dans le fichier `pg_hba.conf` (HBA est l'acronyme de *Host Based Authentication*). Ce fichier est présent à côté de `postgresql.conf`, donc par exemple dans `/var/lib/pgsql/16/data` (défaut du packaging Red Hat) ou `/etc/postgresql/16/main` (défaut du packaging Debian). L'accès doit bien sûr en être le plus restreint possible.

Le contenu du fichier se présente en principe ainsi :

- 4 colonnes d'informations ;
- 1 colonne indiquant la méthode à appliquer ;
- 1 colonne optionnelle d'options.

Depuis PostgreSQL 16, il est possible d'indiquer ou plusieurs fichiers externes :

```
include pg_hba.ansible.conf
include "/chemin/fichier 2.conf"
```

(Existent aussi `include_if_exists`, qui ne tombe pas en erreur si le fichier n'existe pas, et `include_dir`, pour inclure tous les fichiers `.conf` d'un répertoire dans l'ordre de leurs noms.)

`pg_hba.conf` ne peut pas être modifié depuis PostgreSQL même, uniquement depuis le système d'exploitation. Après une modification, il faut dire explicitement à PostgreSQL de le recharger. Selon l'installation et l'OS, cet ordre peut être :

```
pg_ctl reload -D /mnt/base_pg/
systemctl reload postgresql-16
pg_ctlcluster 16 main reload
```

ou depuis PostgreSQL même :

```
SELECT pg_reload_conf() ;
```

Il y a une exception : sous Windows, le fichier est relu dès modification.

Si le fichier contient une erreur de syntaxe, il sera ignoré (et la ligne fautive apparaîtra dans les traces), sauf au (re)démarrage, où un fichier fautif bloque le démarrage !·

Exemple de trace problématique :

```
…user=,db=,app=,client= LOG:  invalid connection type "hsot"
…user=,db=,app=,client= CONTEXT:  line 121 of configuration file
 ↪  "/etc/postgresql/16/main/pg_hba.conf"
…user=,db=,app=,client= FATAL:  could not load /etc/postgresql/16/main/pg_hba.conf
```

Depuis une session utilisateur, il est possible de consulter le fichier via la vue `pg_hba_file_rules`, tel qu'il est sur le disque, et tel qu'il serait compris par PostgreSQL après un rechargement. La colonne `error` est très pratique pour repérer immédiatement les erreurs de syntaxe ou de frappe. La colonne `file_name` indique le fichier source (à partir de PostgreSQL 16).

5.5.3 Exemple de pg_hba.conf

```
# TYPE  DATABASE        USER      ADDRESS           METHOD
# accès direct par la socket
local    all             all                         peer
# accès en local via réseau (localhost IPv6 et IPv4)
host      all            all      ::1/128            scram-sha-256
host      all            all      127.0.0.0/24       scram-sha-256
# accès distants
hostssl   erp            all      192.168.0.0/16     scram-sha-256
hostssl   logistique     all      192.168.30.0/24    scram-sha-256
hostnossl test           demo     192.168.74.150/32  scram-sha-256
# vieux client
hostssl   compta         durand   192.168.10.99/32   md5
# Connexions de réplication
local   replication      all                         peer
host    replication      all      192.168.74.5/32    scram-sha-256
host    replication      all      127.0.0.1/32       scram-sha-256
host    replication      all      ::1/128            scram-sha-256
```

PostgreSQL lit le fichier dans l'ordre. La première ligne correspondant à la connexion demandée lui précise la méthode d'authentification à utiliser. Nous verrons que certaines méthodes comme `scram-sha-256` ou `md5` (les plus courantes) exigent un mot de passe, d'autres non, comme `ldap` ou `cert`. Il ne reste plus qu'à appliquer cette méthode. Si elle fonctionne, la connexion est autorisée et se poursuit.

Si elle ne fonctionne pas, quelle qu'en soit la raison, la connexion est refusée. Aucune autre ligne du fichier ne sera lue ! La raison du refus est tracée sur le serveur de manière beaucoup plus complète que pour le client :

```
2023-11-03 10:41:24 CET [1805700]: [2-1] user=attaquant,db=erp,
app=[unknown],client=::1 DETAIL:  Role "mechant_hacker" does not exist.
Connection matched file "/etc/postgresql/16/defo/pg_hba.conf"
line 110: "host    all      all      ::1/128           scram-sha-256"
```

Il est donc essentiel de bien configurer ce fichier pour avoir une protection maximale, et de faire très attention à l'ordre des entrées.

5.5.4 pg_hba.conf : colonne type

- 4 valeurs possibles
  - `local`
  - `host`
  - `hostssl`
  - `hostnossl` (`ssl=on` prérequis)

La colonne type peut contenir quatre valeurs différentes. La valeur `local` concerne les connexions via la socket Unix, donc depuis un compte sur le serveur même. Elle est généralement couplée à la méthode `peer`. En pratique, on s'en sert surtout pour les connexions administratives (de l'utiliseur système **postgres** au rôle **postgres**), pour les sauvegardes depuis le serveur même, ou pour une connexion depuis une application qui tourne sur la même machine.

Toutes les autres valeurs concernent les connexions via le réseau (socket TCP). La différence réside dans l'utilisation forcée ou non du SSL :

- `host` : SSL au choix du client ;
- `hostssl` : impérativement avec chiffrage SSL ;
- `hostnossl` : impérativement sans SSL.

Il est à noter que l'option `hostssl` n'est utilisable que si le paramètre `ssl` du fichier `postgresql.conf` est à `on` (c'est le cas par défaut sous Debian, pas forcément ailleurs).

5.5.5 pg_hba.conf : colonne database

À quelle(s) base(s) autoriser la connexion ?

- `mabase`
- `base1,base2,base3`
- `sameuser` / `samerole`
- `all`
- `@fichier.txt` (fichier avec plusieurs bases)
- `/db_[1-6]`, `/erp.*` (regex, v16+)
- `replication` (pseudo-base pour réplication physique)

La seconde colonne de `pg_hba.conf` peut recueillir le nom d'une base, le nom de plusieurs bases en les séparant par des virgules, le nom d'un fichier contenant la liste des bases ou quelques valeurs en dur. La valeur `all` indique toutes les bases.

Enfin, la valeur `sameuser` spécifie une base de données de même nom que le rôle demandé (l'utilisateur **durand** peut se connecter à la base **durand**), alors que la valeur `samerole` spécifie que le rôle demandé doit être membre du rôle portant le même nom que la base de données demandée (par exemple, si l'utilisateur **durand** est membre du groupe **compta**, il peut se connecter à la base **compta**).

Il est possible d'indiquer un fichier (préfixé de `@`) qui contiendra une liste de rôles. Il peut être à côté de `pg_hba.conf` mais on peut fournir un chemin complet.

Depuis PostgreSQL 16, il est possible d'utiliser des expressions régulières, qui doivent être précédées de `/` (voir exemple plus bas).

La valeur `replication` est utilisée pour définir les connexions de réplication physique (mais pas logique). Il n'est pas nécessaire d'avoir une base nommée **replication**, toutes les bases sont concernées par une réplication physique.

5.5.6 pg_hba.conf : colonne user

À quel(s) utilisateur(s) permettre la connexion ?

- `unrole`
- `role1,role2,role3`
- `all`
- `+groupe` (membres d'un rôle-groupe)
- `@fichier.txt` (fichier avec plusieurs utilisateurs)
- `/user_.*` (regex, v16+)

La troisième colonne peut recueillir :

- le nom d'un rôle, ou plusieurs rôles séparés par des virgules ;
- le nom d'un groupe (en le précédant d'un signe `+`) : tous les membres de ce groupe pourront se connecter avec leur rôle propre ;
- le nom d'un fichier (précédé de `@`) contenant une liste de rôles ;
- la valeur `all` qui indique tous les rôles ;
- une regex (précédée de `/`) pour indiquer plusieurs rôles (nouveauté de la version 16).

Exemple :

```
# TYPE    DATABASE      USER              ADDRESS       METHOD
hostssl   erp_hr        u_virginie,u_sharon 10.1.17.67/32 scram-sha-256
hostssl   all           +grp_admin        10.0.0.0/8    scram-sha-256
hostssl   /erp_dev[1-5] @prestataires.txt 10.1.0.0/16   scram-sha-256
hostssl   /erp.*        /u_.*             10.0.0.0/8    scram-sha-256
```

Ici, ne peuvent se connecter à la base **erp_hr** que deux utilisatrices nommées, et uniquement depuis une IP précise. Les membres du groupe **grp_admin** peuvent se connecter à toutes les bases depuis tout le réseau privé. Aux bases **erp_dev1** à **erp_dev5** peuvent se connecter les prestataires dont les rôles sont dans le fichier `prestataires.txt`, depuis un certain sous-réseau. Et tous les utilisateurs dont le rôle commence par **u_** peuvent se connecter aux bases dont le nom commence par **erp_**.

5.5.7 pg_hba.conf : colonne adresse IP

- Uniquement pour `host` / `hostssl` / `hostnossl`
- Adresse IPv4 ou IPv6 : format CIDR ou avec masque
  - `192.168.1.0/24`
  - `192.168.1.1/32`
  - `192.168.1.0  255.255.255.0`
- Nom d'hôte possible
  - mais coût recherche DNS

La colonne de l'adresse IP permet d'indiquer une adresse IP ou un sous-réseau IP. Il est donc possible de filtrer les connexions par rapport aux adresses IP, ce qui est une excellente protection. Voici deux exemples d'adresses IP au format adresse et masque de sous-réseau :

```
192.168.168.1 255.255.255.255
192.168.168.0 255.255.255.0
```

Et voici quelques exemples d'adresses IP (ou plages d'adresses) au format CIDR :

```
192.168.0.0/8       # réseau 192.168.x.y
192.168.66.0/24     # réseau 192.168.66.x
192.168.168.1/32    # 1 IP
0.0.0.0/0           # toutes les IPv4
::0/0               # toutes les IPv6
::1/128             # localhost en IPv6
fe80::7a31:c1ff:0000:0000/96  # un réseau en IPv6
```

Dans le doute, la vue `pg_hba_file_rules` indique le masque calculé.

Les alias `samehost` et `samenet` désignent le serveur (avec toutes ses adresses) et tous les réseaux auquels il appartient.

Il est possible d'utiliser un nom d'hôte ou un domaine DNS au prix d'une recherche DNS pour chaque hostname présent, pour chaque nouvelle connexion.

Rappelons que le paramètre `listen_addresses` dans `postgresql.conf` restreint les IP sur lesquelles que PostgreSQL écoute. En général on le modifie dès l'installation.

5.5.8 pg_hba.conf : colonne méthode

Quelle méthode d'authentification utiliser ?

- Interne / externe
  - et options dans la dernière colonne
- Le client peut en exiger une (v16)

La dernière colonne est généralement la méthode d'authentification, mais certaines acceptent des options dans une colonne supplémentaire. Nous verrons plus bas les différentes méthodes d'authentification, internes ou externes.

Le serveur a toujours le dernier mot pour imposer une méthode d'authentification, mais pour des raisons de sécurité le client peut aussi exiger celle qu'il attend (depuis PostgreSQL 16). Dans l'exemple suivant, le client exige la méthode `scram-sha-256`, mais le serveur est encore configuré en `md5` et la connexion échoue.

```
psql 'host=s1 dbname=postgres user=postgres require_auth=scram-sha-256'
psql: error: connection to server at "s1" (192.168.74.65), port 54325 failed:
 authentication method requirement "scram-sha-256" failed:
 server requested a hashed password
```

D'autres variantes pour `require_auth` sont :

- `require_auth=scram-sha-256,md5`
- `require_auth=!password`
- `require_auth=cert`

5.5.9 pg_hba.conf : colonne options

- Dépend de la méthode d'authentification
- Méthode externe : option `map`

Les options disponibles dépendent de la méthode d'authentification sélectionnée. Cependant, toutes les méthodes externes permettent l'utilisation de l'option `map`. Cette option a notamment pour but d'indiquer la carte de correspondance à sélectionner dans le fichier `pg_ident.conf` (voir plus bas).

5.5.10 pg_hba.conf : méthodes internes

- `trust` : dangereux !
- `reject` : pour interdire
- `password` : en clair, à éviter !
- `md5` : déconseillée
- `scram-sha-256`

La méthode `trust` est certainement la pire. À partir du moment où le rôle est reconnu, aucun mot de passe n'est demandé. Si le mot de passe est fourni malgré tout, il n'est pas vérifié. Il est donc essentiel de proscrire cette méthode d'authentification.

La méthode `password` force la saisie d'un mot de passe. Cependant, ce dernier est envoyé en clair sur le réseau. Il n'est donc pas conseillé d'utiliser cette méthode, surtout sur un réseau non sécurisé. (Noter que PostgreSQL chiffrera toujours le mot de passe chiffré avec une des méthodes ci-dessous.)

La méthode `md5` est encore très utilisée mais obsolète. La saisie du mot de passe est forcée. Le mot de passe transite chiffré en `md5`. Cette méthode souffre néanmoins de certaines faiblesses décrites dans la section Mot de passe, on préférera la suivante.

La méthode `scram-sha-256` est la plus sécurisée, elle offre moins d'angles d'attaque que `md5`. Elle est à privilégier.

La méthode `reject` sert pour interdire définitivement un accès à un utilisateur, un réseau ou une base quelles que soient les lignes suivantes de `pg_hba.conf`. Par exemple, on veut que le rôle **u1** puisse se connecter à la base de données `b1` mais pas aux autres. Voici un moyen de le faire (pour une connexion via les sockets Unix) :

```
local  b1   u1  scram-sha-256
local  all  u1  reject
```

5.5.11 pg_hba.conf : méthodes externes

- Système externe : `ldap`, `radius`
- Kerberos : `gss`, `sspi`
- SSL : `cert`
- OS : `peer`, `pam`, `ident`, `bsd`

Certaines méthodes permettent d'utiliser des annuaires d'entreprise comme RADIUS, LDAP ou ActiveDirectory. Certaines méthodes sont spécifiques à un système d'exploitation.

La méthode `ldap` utilise un serveur LDAP pour authentifier l'utilisateur.

La méthode `radius` permet d'utiliser un serveur RADIUS pour authentifier l'utilisateur.

La méthode `gss` (GSSAPI) correspond au protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743. PostgreSQL supporte GSSAPI avec l'authentification Kerberos suivant la RFC 1964 ce qui permet de faire du *Single Sign-On*. C'est la méthode à utiliser avec Active Directory. `sspi` (uniquement dans le monde Windows) permet d'utiliser NTLM faute d'Active Directory.

La méthode `cert` utilise un certificat SSL sur le client[11], sans mot de passe.

La méthode `ident` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Un démon fournissant le service ident est nécessaire. (Les paquets RPM la mettaient en place jusque PostgreSQL 12 inclus pour une connexion depuis `localhost`, mais `peer` est maintenant recommandé.)

La méthode `peer` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Ceci n'est possible qu'avec une connexion locale. Par défaut, les paquets d'installation l'utilisent pour autoriser l'utilisateur système **postgres** à se connecter à l'instance locale, sans passer par le réseau, en tant que **postgres** sans mot de passe.

La méthode `pam` authentifie l'utilisateur en passant par les *Pluggable Authentication Modules* (PAM) fournis par Linux et d'autres Unix. De manière similaire, la méthode `bsd` utilise le système d'authentification BSD sur OpenBSD.

[11] https://docs.postgresql.fr/current/auth-cert.html

5.5.12 Un (mauvais) exemple de pg_hba.conf

Un exemple:

```
TYPE    DATABASE  USER        ADDRESS         METHOD
local   all       postgres                    peer
local   web       web                         md5
local   sameuser  all                         peer
host    all       postgres    127.0.0.1/32    peer
host    all       all         127.0.0.1/32    md5
host    all       all         89.192.0.3/8    md5
hostssl recherche recherche   89.192.0.4/32   scram-sha-256
```

à ne pas suivre…

Ce fichier comporte plusieurs erreurs. Tout d'abord,

```
host    all       all         127.0.0.1/32    md5
```

autorise tous les utilisateurs, en IP, en local (127.0.0.1) à se connecter à **toutes** les bases, ce qui est en contradiction avec :

```
local   sameuser  all                         peer
```

qui, sera appliqué d'abord et restreint à la base de même nom.

Dans cette règle :

```
host    all       all         89.192.0.3/8    md5
```

le masque CIDR `/8` est incorrect, et donc, au lieu d'autoriser seulement 89.192.0.3 à se connecter, on autorise tout le réseau 89.*.

L'entrée :

```
hostssl recherche recherche   89.192.0.4/32   scram-sha-256
```

est bonne, mais inutile, car masquée par la ligne précédente : toute ligne correspondant à cette entrée correspondra aussi à la ligne précédente. Le fichier étant lu séquentiellement, cette dernière entrée ne sert à rien.

Enfin, il aurait mieux valu que tous les mots de passe utilisent le chiffrage `scram-sha-256` et non `md5`.

5.5.13 Mapping : pg_ident.conf

- Correspondance entre :
  - utilisateurs authentifiés (serveur, LDAP)
  - rôle de la base
- Fichier `pg_ident.conf` :

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
pgadmins        joe                     postgres
pgadmins        zoe                     postgres
```

- Prise en compte des modifications : rechargement
- Includes possibles
- Contrôle : `pg_ident_file_mapping` (v15+)

En utilisant une méthode d'authentification externe, locale ou à distance, le nom de l'utilisateur authentifié ne correspond pas forcément au nom du rôle avec lequel on souhaite se connecter. On peut le préciser avec `-U` mais il y a plus simple et sécurisé.

Par exemple, deux DBA ont les comptes **joe** et **zoe** sur le système d'exploitation. Ils souhaitent pouvoir se connecter en tant que rôle **postgres** sans saisie de mot de passe depuis leur compte, sans devoir faire un `sudo -iu postgres` pour lesquels on ne va pas forcément leur donner les droits.

Ceci n'est pas possible avec la configuration par défaut :

```
joe$ psql -U postgres
psql: error: FATAL:  Peer authentication failed for user "postgres"
```

Mais cela devient possible en configurant correctement les fichiers `pg_hba.conf` et `pg_ident.conf`. Il est tout d'abord nécessaire d'indiquer trois informations dans le fichier `pg_ident.conf`, une « map », les utilisateurs authentifiés (ici **joe** et **zoe**) et le rôle de connexion (ici **postgres**) :

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
pgadmins        joe                     postgres
pgadmins        zoe                     postgres
```

À partir de la version 15, il est possible de vérifier que la configuration de ce fichier est bonne. Pour cela, il faut lire la vue `pg_ident_file_mappings` (et vérifier que le champ `error` est vide).

```
SELECT * FROM pg_ident_file_mappings;
```

```
 line_number | map_name | sys_name | pg_username | error
-------------+----------+----------+-------------+-------
          43 | pgadmins | joe      | postgres    |
          44 | pgadmins | zoe      | postgres    |
```

Il faut ensuite ajouter cette *map* à la ligne de configuration d'authentification en haut du fichier `pg_hba.conf` :

```
local    all   all     peer      map=pgadmins
```

Après rechargement de la conf, la connexion de **joe** ou **zoe** à **postgres** en local se fera. Noter que la sécurité de l'accès à PostgreSQL repose à présent sur la sécurisation des utilisateurs système **joe** et **zoe** !

Tout ceci n'est valable que pour les connexions depuis le système d'exploitation même. En pratique, `pg_ident.conf` est utilisé pour se connecter directement en tant que **postgres** depuis un compte utilisateur sur le serveur, ou avec une authentification LDAP.

Depuis PostgreSQL 16, `pg_ident.conf` supporte les mêmes directives d'inclusion que `pg_hba.conf` (`include` , `include_if_exists` , `include_dir`), et le fichier d'origine d'une ligne est visible dans `pg_ident_file_mappings` .

5.6 TÂCHES DE MAINTENANCE

- Trois opérations essentielles
  - `VACUUM`
  - `ANALYZE`
  - `REINDEX`
- En arrière-plan : démon autovacuum (pour les deux premiers)
- Optionnellement : automatisable par cron
- Manuellement : `VACUUM ANALYZE table` (batchs, gros imports…)

PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à conserver un système performant et agréable à utiliser.

La maintenance d'un serveur PostgreSQL revient à s'occuper de trois opérations :

- le `VACUUM`, pour éviter une fragmentation trop importante des tables ;
- l'`ANALYZE`, pour mettre à jour les statistiques sur les données contenues dans les tables ;
- le `REINDEX`, pour reconstruire les index.

Il s'agit donc de maintenir, voire d'améliorer, les performances du système. Il ne s'agit en aucun cas de s'assurer de la stabilité du système.

Généralement on se reposera sur le processus d'arrière-plan **autovacuum**, qui s'occupe des `VACUUM` et `ANALYZE` (mais pas `REINDEX`) en fonction de l'activité, et prend soin de ne pas la gêner. Il est possible de planifier des exécutions régulières avec cron (ou tout autre ordonnanceur) , notamment pour des `REINDEX`.

Un appel explicite est parfois nécessaire, notamment au sein de batchs ou de gros imports… L'autovacuum n'a pas forcément eu le temps de passer entre deux étapes, et les statistiques ne sont alors pas à jour : le planificateur pense que les tables sont encore vides et peut choisir un plan désastreux. On lancera donc systématiquement au moins un `ANALYZE` sur les tables modifiées après les modifications lourdes. Un `VACUUM ANALYZE` est parfois encore plus intéressant, notamment si des données ont été modifiées ou effacées, ou si les requêtes suivantes peuvent profiter d'un parcours d'index seul (`Index Only Scan`).

5.6.1 Maintenance : VACUUM

- `VACUUM nomtable ;`
  - cartographie les espaces libres pour une réutilisation (& autre maintenance)
  - utilisable en parallèle avec les autres opérations
  - et même automatisé
  - vue `pg_stat_progress_vacuum`
- `vacuumdb --echo`

PostgreSQL ne supprime pas des tables les versions périmées des lignes après un `UPDATE` ou un `DELETE`, elles deviennent juste invisibles. La commande `VACUUM` permet de récupérer l'espace utilisé par ces lignes afin d'éviter un accroissement continuel du volume occupé sur le disque.

Une table qui subit beaucoup de mises à jour et suppressions nécessitera des nettoyages plus fréquents que les tables rarement modifiées. Le `VACUUM` « simple » (`VACUUM nomdematable ;`) marque les données expirées dans les tables et les index pour une utilisation future. Il ne tente pas de rendre au système de fichiers l'espace utilisé par les données obsolètes, sauf si l'espace est à la fin de la table et qu'un verrou exclusif de table peut être facilement obtenu. L'espace inutilisé au début ou au milieu du fichier ne provoque pas un raccourcissement du fichier et ne redonne pas d'espace mémoire au système d'exploitation. De même, l'espace d'une colonne supprimée n'est pas rendu.

Cet espace libéré n'est pas perdu : il sera disponible pour les prochaines lignes insérées et mises à jour, et la table n'aura pas besoin de grandir.

Un `VACUUM` peut être lancé sans aucune gêne pour les utilisateurs. Il va juste générer des écritures supplémentaires. On verra plus loin que l'autovacuum s'occupe de tout cela en tâche de fond et de manière non intrusive, mais il arrive encore que l'on lance un `VACUUM` manuellement. Noter qu'un `VACUUM` s'occupe également de quelques autres opérations de maintenance qui ne seront pas détaillées ici.

L'option `VERBOSE` vous permet de suivre ce qui a été fait. Dans l'exemple suivant, 100 000 lignes sont nettoyées dans 541 blocs, mais 300 316 lignes ne peuvent être supprimées car une autre transaction reste susceptible de les voir.

```
VACUUM VERBOSE livraisons ;

INFO:  vacuuming "public.livraisons"
INFO:  "livraisons": removed 100000 row versions in 541 pages
INFO:  "livraisons": found 100000 removable, 300316 nonremovable
                                        row versions in 2165 out of 5406 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 6249883
There were 174 unused item pointers.
Skipped 0 pages due to buffer pins, 540 frozen pages.
```

```
0 pages are entirely empty.
CPU: user: 0.04 s, system: 0.00 s, elapsed: 0.08 s.
VACUUM
Temps : 88,990 ms
```

L'ordre `VACUUM` possède de nombreux paramètres que nous ne décrirons pas ici car ils servent peu au quotidien (au besoin, voir le module M5[12]).

Supervision :

La vue `pg_stat_progress_vacuum` permet de suivre un `VACUUM` simple pendant son déroulement.

La vue `pg_stat_user_tables` contient pour chaque table la date du dernier passage d'un `VACUUM` simple (champ `last_vacuum`) celle du dernier passage automatique (`last_autovacuum`). Pour les passages précédents, il faudra se rabattre sur les traces (on conseille de positionner `log_autovacuum_min_duration` suffisamment bas, ou à 0). Il est important de vérifier que les tables actives sont régulièrement nettoyées.

Outil en ligne de commande :

L'outil `vacuumdb` permet d'exécuter depuis le shell un `VACUUM` sur une ou toutes les bases. C'est l'outil idéal lorsque l'on planifie un nettoyage quotidien ou hebdomadaire.

Il sait exécuter des `VACUUM` sur plusieurs tables en parallèle, ou profiter des optimisations liées aux nettoyages en masse apparues en v16.

```
$ vacuumdb --echo --all

SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
 JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
 LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
 WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
 ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
…
…
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
…
…
VACUUM (ONLY_DATABASE_STATS);
```

[12] https://dali.bo/m5_html

L'ordre `VACUUM` connaît beaucoup de variantes. Surtout, ne confondez pas la version simple ci-dessus avec sa variante `VACUUM FULL` ci-dessous.

5.6.2 Maintenance : VACUUM FULL

- `VACUUM FULL nomtable ;`
  - défragmente la table
  - réécriture (place nécessaire !)
  - verrou exclusif (ni lecture ni écriture !)
  - réindexation
  - utilisation exceptionnelle
  - vue `pg_stat_progress_cluster` (v12)

Un `VACUUM` simple fait rarement gagner de l'espace disque. Il faut utiliser l'option `FULL` pour ça : la commande `VACUUM FULL nomtable ;` réécrit la table en ne gardant que les données actives, et au final libère donc l'espace consommé par les lignes périmées ou les colonnes supprimées, et le rend au système d'exploitation. Les index sont réécrits au passage.

Inconvénient principal : `VACUUM FULL` acquiert un verrou exclusif sur chaque table concernée : personne ne peut plus y écrire ni même lire avant la fin de l'opération, et les sessions accédant aux tables sont mises en attente. Cela peut être long pour de grosses tables. D'autre part, le `VACUUM FULL` peut lui-même attendre la libération d'un verrou, tout en bloquant les transactions suivantes (phénomène d'empilement des verrous). Il est conseillé d'opérer par exemple ainsi :

```
SET lock_timout TO '3s';
VACUUM (FULL, VERBOSE) nomtable ;
```

Ainsi l'ordre `VACUUM FULL` sera annulé s'il n'obtient pas son verrou assez vite.

Autre inconvénient : `VACUUM FULL` écrit la nouvelle version de la table *à côté* de l'ancienne version avant d'effacer cette dernière : l'espace occupé peut donc temporairement doubler. Si vos disques sont presque pleins, vous ne pourrez donc pas faire un `VACUUM FULL` d'une grosse table pour récupérer de l'espace !

L'autovacuum ne procédera jamais à un `VACUUM FULL`, vous devrez toujours le demander explicitement. On le réservera aux périodes de maintenance, dans les cas où il est vraiment nécessaire.

En effet, il ne sert à rien de chercher à réduire au strict minimum la taille des tables par des `VACUUM FULL` répétés. Dans une base active, les espaces libres sont vite réutilisés par de nouvelles

données. Le *bloat* (l'espace inutilisé d'une table) se stabilise généralement dans une proportion dépendant des débits d'insertions, suppressions et modifications dans la table.

Supervision :

La vue `pg_stat_progress_cluster` permet de suivre un `VACUUM FULL` (à partir de PostgreSQL 12). Son nom provient de la proximité avec la commande `CLUSTER` (voir plus bas).

Les `VACUUM FULL` ne sont pas tracés dans la vue `pg_stat_user_tables`.

Outil en ligne de commande :

L'outil `vacuumdb` possède une option `--full`.

5.6.3 VACUUM vs VACUUM FULL

- `VACUUM`
  - maintenance quotidienne
  - entre étapes d'un batch
  - l'autovacuum suffit généralement
- `VACUUM FULL`
  - après de grosses modifications
  - exceptionnel

Quand faut-il utiliser `VACUUM` sur une table ?

- pour des nettoyages réguliers ;
- entre les étapes d'un batch ;
- si vous constatez que l'autovacuum ne passe pas assez souvent et qu'un changement de paramétrage ne suffit pas ;
- et ce, pendant que votre base tourne.

Quand faut-il utiliser `VACUUM FULL` sur une table ?

- après des suppressions massives de données ;
- si le verrou exclusif ne gêne pas la production ;
- dans le cadre d'une maintenance exceptionnelle.

Des `VACUUM` standards et une fréquence modérée sont une meilleure approche que des `VACUUM FULL`, même non fréquents, pour maintenir des tables mises à jour fréquemment : faites confiance à l'autovacuum jusque preuve du contraire.

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, et que les espaces libres ne seront pas à nouveau remplis assez vite, de façon à ce que la taille de la table soit réduite de façon conséquente.

Les deux outils peuvent se lancer à la suite. Après un `VACUUM FULL` (bloquant) sur une table, on lance souvent immédiatement un `VACUUM ANALYZE`. Cela semble inutile du point de vue des données, mais les autres opérations de maintenance impliquées peuvent améliorer les performances.

5.6.4 Maintenance : ANALYZE

- Met à jour les statistiques sur les données pour l'optimiseur de requêtes
- Géré par l'autovacuum
  - Parfois manuel : batch, `ALTER TABLE`, tables temporaires…
- Échantillonnage :
  - `default_statistics_target` (défaut 100)
  - `ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500;`
  - Attention au temps de planification !
- Progression avec `pg_stat_progress_analyze` (v13)

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques calculées à partir des données des tables. Ces statistiques sont récupérées par la commande `ANALYZE`, qui peut être invoquée seule ou comme une option de `VACUUM`. Il est important d'avoir des statistiques relativement à jour sans quoi des mauvais choix dans les plans d'exécution pourraient pénaliser les performances de la base.

L'autovacuum de PostgreSQL appelle au besoin `ANALYZE` si l'activité de la table le nécessite. C'est généralement suffisant, même s'il est fréquent de modifier le paramétrage sur de grosses tables.

Il est possible de programmer `ANALYZE` périodiquement (le dimanche, la nuit par exemple, à l'aide d'une commande `cron` par exemple), éventuellement couplé à un `VACUUM` :

```
VACUUM ANALYZE nomdematable ;
```

Il existe des cas où lancer un `ANALYZE` manuellement est nécessaire :

- en mode « batch » : l'autovacuum n'a pas forcément le temps de passer entre deux étapes, on peut être amené à intercaler un `VACUUM ANALYZE` sur des tables modifiées ;

- quand certains plans de requêtes affichent des statistiques aberrantes : la mise à jour des statistiques peut suffire (et l'on regardera ensuite dans `pg_stat_user_tables.last_autoanalyze` si l'autovacuum a tardé et s'il y a un ajustement à faire ce côté) ;
- après un `ALTER TABLE [...] ALTER COLUMN`, car les statistiques de la colonne peuvent disparaître, ou bien sûr lors de l'ajout d'une colonne pré-remplie ;
- lors de l'ajout d'un index fonctionnel : l'`ANALYZE` mène à la création d'une nouvelle entrée dans `pg_statistics` ;
- lors de l'utilisation des tables temporaires : l'autovacuum ne les voit pas.

Le paramètre `default_statistics_target` définit l'échantillonnage par défaut des statistiques pour les colonnes de chacune des tables. La valeur par défaut est de 100. Ainsi, pour chaque colonne, 30 000 lignes sont choisies au hasard, et les 100 valeurs les plus fréquentes et un histogramme à 100 bornes sont stockés dans `pg_statistics` en guise d'échantillon représentatif des données.

Des valeurs supérieures provoquent un ralentissement important d'`ANALYZE`, un accroissement de la table `pg_statistics`, et un temps de calcul des plans d'exécution plus long. On conserve généralement la valeur `100` par défaut (sauf peut-être sur certaines grosses bases aux requêtes complexes et longues, comme des entrepôts de données).

Voici la commande à utiliser si l'on veut modifier cette valeur pour une colonne précise, la valeur ainsi spécifiée prévalant sur la valeur de `default_statistics_target` :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 200 ;
```

```
ANALYZE ma_table ;
```

Sans l'`ANALYZE` explicite, la mise à jour attendrait le prochain passage de l'autovacuum.

La vue `pg_stat_user_tables` contient aussi les dates du dernier passage d'un `ANALYZE` manuel (champ `last_analyze`) ou automatique (`last_autoanalyze`). Là encore, vérifier que les tables actives sont régulièrement analysées.

La version 13 apporte une vue appelée `pg_stat_progress_analyze` qui permet de suivre l'exécution des `ANALYZE` en cours.

5.6.5 Maintenance : REINDEX

- Lancer `REINDEX` régulièrement permet
  - de gagner de l'espace disque
  - d'améliorer les performances
  - de réparer un index corrompu/invalide
- `VACUUM` ne provoque pas de réindexation
- `VACUUM FULL` réindexe
- Clause `CONCURRENTLY` (v12+)
- Clause `TABLESPACE` (v14+)

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. La même commande peut réindexer tous les index d'une table :

```
REINDEX INDEX nomindex ;
REINDEX (VERBOSE) TABLE nomtable ;
```

Les pages d'index qui sont devenues complètement vides sont récupérées pour être réutilisées. Il existe toujours la possibilité d'une utilisation inefficace de l'espace : même s'il ne reste qu'une clé d'index dans une page, la page reste allouée. La possibilité d'inflation n'est pas indéfinie, mais il est souvent utile de planifier une réindexation périodique pour les index fréquemment modifiés.

De plus, pour les index B-tree, un index tout juste construit est plus rapide qu'un index qui a été mis à jour plusieurs fois. En effet, dans un index nouvellement créé, les pages logiquement adjacentes sont aussi physiquement adjacentes.

La réindexation est aussi utile dans le cas d'un index corrompu. Ce cas est heureusement très rare, et souvent lié à des problèmes matériels.

Les index « invalides » sont inutilisables et ignorés, et doivent également être reconstruits. Ce statut apparaît en bas de la description de la table associée :

```
# \d+ pgbench_accounts
…
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid) INVALID
```

Un index peut devenir invalide pour deux raisons. La première ne concerne plus les versions supportées : avant PostgreSQL 10, des index de type hash (uniquement) pouvaient devenir invalides après un redémarrage brutal, car ils n'étaient alors pas journalisés. La seconde raison est une conséquence de la clause `CONCURRENTLY` des ordres `CREATE INDEX` et `REINDEX`. Cette clause permet de créer/réindexer un index sans bloquer les écritures dans la table. Cependant, si, au bout de deux passes, l'index n'est toujours pas complet, il est considéré comme invalide, et doit être soit détruit, soit reconstruit avec la commande `REINDEX`.

Noter que, sans `CONCURRENTLY`, un `REINDEX` bloque non seulement les écritures, mais aussi souvent les lectures. On préférera donc le `CONCURRENTLY` si la table est utilisée :

```
REINDEX (VERBOSE) INDEX nomindex CONCURRENTLY ;
```

Enfin, depuis la version 14, il est possible de réindexer un index tout en le changeant de tablespace. Pour cela, il faut utiliser la clause `TABLESPACE` avec en argument le nom du tablespace de destination.

Il est à savoir que l'opération `VACUUM` (sans `FULL`) ne provoque pas de réindexation. Une réindexation est effectuée lors d'un `VACUUM FULL`.

La commande système `reindexdb` peut être utilisée pour réindexer une table, une base ou une instance entière.

5.6.6 Maintenance : CLUSTER

- `CLUSTER`
  - alternative à `VACUUM FULL`
  - tri des données de la table suivant un index
- Attention, `CLUSTER` nécessite près du double de l'espace disque utilisé pour stocker la table et ses index
- Progression avec `pg_stat_progress_cluster`

La commande `CLUSTER` provoque une réorganisation des données de la table en triant les lignes suivant l'ordre indiqué par l'index. Du fait de la réorganisation, le résultat obtenu est équivalent à un `VACUUM FULL` dans le contexte de la fragmentation. Elle verrouille tout aussi complètement la table et nécessite autant de place.

Attention, cette réorganisation est ponctuelle, et les données modifiées ou insérées par la suite n'en tiennent généralement pas compte. L'opération peut donc être à refaire après un certain temps.

Comme après un `VACUUM FULL`, lancer un `VACUUM ANALYZE` manuellement peut être bénéfique pour les performances.

En ligne de commande, l'outil associé `clusterdb` permet de lancer la réorganisation de tables ayant déjà fait l'objet d'une « clusterisation ».

La vue `pg_stat_progress_cluster` permet de suivre le déroulement du `CLUSTER`.

5.6.7 Maintenance : automatisation

- Automatisation des tâches de maintenance
- Cron sous Unix
- Tâches planifiées sous Windows

L'exécution des commandes `VACUUM`, `ANALYZE` et `REINDEX` peut se faire manuellement dans certains cas. Il est cependant préférable de mettre en place une exécution automatique de ces commandes. La plupart des administrateurs utilise cron sous Unix et les tâches planifiées sous Windows. pgAgent peut aussi être d'une aide précieuse pour la mise en place de ces opérations automatiques.

Peu importe l'outil. L'essentiel est que ces opérations soient réalisées et que le statut de leur exécution soit vérifié périodiquement.

La fréquence d'exécution dépend principalement de la fréquence des modifications et suppressions pour le `VACUUM` et de la fréquence des insertions, modifications et suppressions pour l'`ANALYZE`.

5.6.8 Maintenance : autovacuum

- Automatisation par cron
  - simple, voire simpliste
- Processus autovacuum
  - `VACUUM` / `ANALYZE` si nécessaire
  - Nombreux paramètres
  - Nécessite la récupération des statistiques d'activité

L'automatisation du vacuum par cron est simple à mettre en place. Cependant, elle s'exécute pour toutes les tables, sans distinction. Que la table ait été modifiée plusieurs millions de fois ou pas du tout, elle sera traitée par le script. À l'inverse, l'autovacuum est un outil qui vérifie l'état des tables et, suivant le dépassement d'une limite, déclenche ou non l'exécution d'un `VACUUM` ou d'un `ANALYZE`, voire des deux.

L'autovacuum est activé par défaut, et il est conseillé de le laisser ainsi. Son paramétrage permet d'aller assez loin si nécessaire selon la taille et l'activité des tables.

5.6.9 Maintenance : Script de REINDEX

- Automatisation par cron
- Recherche des index fragmentés
- Si clé primaire ou contrainte unique :
  - `REINDEX`
- Sinon :
  - `CREATE INDEX CONCURRENTLY`
- Exemple

Voici un script créé pour un client dans le but d'automatiser la réindexation uniquement pour les index le méritant. Pour cela, il vérifie les index fragmentés avec la fonction `pgstatindex()` de l'extension `pgstattuple` (installable avec un simple `CREATE EXTENSION pgstattuple ;` dans chaque base).

Au-delà de 30 % de fragmentation (par défaut), l'index est réindexé. Pour minimiser le risque de blocage, le script utilise `CREATE INDEX CONCURRENTLY` en priorité, et `REINDEX` dans les autres cas (clés primaires et contraintes uniques).

La version 12 permet d'utiliser l'option `CONCURRENTLY` avec `REINDEX`. Ce script pourrait l'utiliser après avoir détecté qu'il se trouve sur une version compatible.

```
#!/bin/bash
# Script de réindexation d'une base
# ce script va récupérer la liste des index disponibles sur la base
# et réindexer l'index s'il est trop fragmenté ou invalide

# Mode debug
#set -x

# Récupération de la base maintenance
if test -z "$PGDATABASE"; then
  export PGDATABASE=postgres
fi

# quelques constantes personnalisables
TAUX_FRAGMENTATION_MAX=30
NOM_INDEX_TEMPORAIRE=index_traitement_en_cours
NB_TESTS=3
BASES=""

# Quelques requêtes
REQ_LISTEBASES="SELECT array_to_string(array(
  SELECT datname
```

```
  FROM pg_database
  WHERE datallowconn AND datname NOT IN ('postgres', 'template1')), ' ')"
REQ_LISTEINDEX="
SELECT n.nspname as \"Schéma\", tc.relname as \"Table\", ic.relname as \"Index\",
  i.indexrelid as \"IndexOid\",
  i.indisprimary OR i.indisunique as \"Contrainte\", i.indisvalid as \"Valide?\",
  round(100-(pgstatindex(n.nspname||'.'||ic.relname)).avg_leaf_density)
    as \"Fragmentation\",
  pg_get_indexdef(i.indexrelid) as \"IndexDef\"
FROM pg_index i
JOIN pg_class ic ON i.indexrelid=ic.oid
JOIN pg_class tc ON i.indrelid=tc.oid
JOIN pg_namespace n ON tc.relnamespace=n.oid
WHERE n.nspname <> 'pg_catalog'
  AND n.nspname !~ '^pg_toast'
ORDER BY ic.relname;"

# vérification de la liste des bases
if test $# -gt 1; then
  echo "Usage: $0 [nom_base]"
  exit 1
elif test $# -eq 1; then
  BASE_PRESENTE=$(psql -XAtqc \
  "SELECT count(*) FROM pg_database WHERE datname='$1'" 2>/dev/null)
  if test $BASE_PRESENTE -ne 1; then
    echo "La base de donnees $BASE n'existe pas."
    exit 2
  fi

  BASES=$1
else
  BASES=$(psql -XAtqc "$REQ_LISTEBASES" 2>/dev/null)
fi

# Pour chaque base
for BASE in $BASES
do
    # Afficher la base de données
    echo "########## $BASE ##########"

  # Vérification de la présence de la fonction pgstatindex
  FONCTION_PRESENTE=$(psql -XAtqc \
  "SELECT count(*) FROM pg_proc WHERE proname='pgstatindex'" $BASE 2>/dev/null)
  if test $FONCTION_PRESENTE -eq 0; then
    echo "La fonction pgstatindex n'existe pas."
    echo "Veuillez installer le module pgstattuple."
    exit 3
  fi

  # pour chaque index
  echo "Récupération de la liste des index (ratio cible $TAUX_FRAGMENTATION_MAX)..."
  psql -XAtF " " -c "$REQ_LISTEINDEX" $BASE | \
  while read schema table index indexoid contrainte validite fragmentation definition
  do
  #  NaN (not a number) est possible si la table est vide
  #  dans ce cas, une réindexation est rapide
```

```
  if test "$fragmentation" = "NaN"; then
    fragmentation=0
  fi

#  afficher index, validité et fragmentation
    if test "$validite" = "t"; then
      chaine_validite="valide"
    else
      chaine_validite="invalide"
    fi
  echo "Index $index, $chaine_validite, ratio libre ${fragmentation}%"

#  si index fragmenté ou non valide
    if test "$validite" = "f" -o $fragmentation -gt $TAUX_FRAGMENTATION_MAX; then
#  verifier les verrous sur l'index, attendre un peu si nécessaire
      verrous=1
      tests=0
      while test $verrous -gt 0 -a $tests -le $NB_TESTS
      do
    if test $tests -gt 0; then
      echo \
  "  objet verrouillé, attente de $tests secondes avant nouvelle tentative..."
      sleep $tests
    fi
    verrous=$(psql -XAtqc \
    "SELECT count(*) FROM pg_locks WHERE relation=$indexoid" 2>/dev/null)
    tests=$(($tests + 1))
      done
    if test $verrous -gt 0; then
      echo "  objet toujours verrouillé, pas de reindexation pour $schema.$index"
      continue
    fi

#    si contrainte, reindexation simple
      if test "$contrainte" = "t"; then
        echo -n "  reindexation de la contrainte... "
      psql -Xqc "REINDEX INDEX $schema.$index;" $BASE
      if test $? -eq 0; then
        echo "OK"
      else
        echo "PROBLEME!!"
        continue
      fi
#    sinon
      else
#    renommer <ancien nom> en <index_traitement_en_cours>
          echo -n "  renommage... "
      psql -Xqc \
      "ALTER INDEX $schema.$index RENAME TO $NOM_INDEX_TEMPORAIRE;" $BASE
      if test $? -eq 0; then
        echo "OK"
      else
        echo "PROBLEME!!"
        continue
      fi
#    create index <ancien nom>
```

```
            echo -n "  création nouvel index..."
        psql -Xqc "$definition;" $BASE
  #     si create OK, drop index <index_traitement_en_cours>
        if test $? -eq 0; then
          echo "OK"
              echo -n "  suppression ancien index..."
          psql -Xqc "DROP INDEX $schema.$NOM_INDEX_TEMPORAIRE;" $BASE
          if test $? -eq 0; then
            echo "OK"
          else
            echo "PROBLEME!!"
            continue
          fi
  #     sinon, renommer <index_traitement_en_cours> en <ancien nom>
        else
          echo "PROBLEME!!"
              echo -n "  renommage inverse..."
          psql -Xqc \
          "ALTER INDEX $schema.$NOM_INDEX_TEMPORAIRE RENAME TO $index;" $BASE
          if test $? -eq 0; then
            echo "OK"
          else
            echo "PROBLEME!!"
            continue
          fi
        fi
      fi
    fi
  done
done
```

5.7 SÉCURITÉ

- Ce qu'un utilisateur standard peut faire
  - et ne peut pas faire
- Restreindre les droits
- Chiffrement
- Corruption de données

À l'installation de PostgreSQL, il est essentiel de s'assurer de la sécurité du serveur : sécurité au niveau des accès, au niveau des objets, ainsi qu'au niveau des données.

Ce chapitre va faire le point sur ce qu'un utilisateur peut faire par défaut et sur ce qu'il ne peut pas faire. Nous verrons ensuite comment restreindre les droits. Enfin, nous verrons les possibilités de chiffrement et de non-corruption de PostgreSQL.

5.7.1 Droits par défaut

Un utilisateur standard peut :

- `CONNECT` : accéder à toutes les bases de données
- `CREATE` :
  - créer des objets dans le schéma `public` de **toute** base de données
  - révocation fréquente
  - … mais plus en v15+ !
- `SELECT` : voir les données de ses tables
- `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` : les modifier
- `TEMP` : créer des objets temporaires
- `CREATE`, `USAGE ON LANGUAGE` : créer des fonctions
- `EXECUTE` : exécuter des fonctions définies par d'autres dans le schéma `public`

Par défaut, un utilisateur a beaucoup de droits.

Il peut accéder à toutes les bases de données. Il faut modifier le fichier `pg_hba.conf` pour éviter cela. Il est aussi possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE CONNECT ON DATABASE nom_base FROM nom_utilisateur ;
```

Avant la version 15, l'utilisateur peut créer des objets dans le schéma disponible par défaut (nommé `public`) sur chacune des bases de données où il peut se connecter, même si la base ne lui appartient pas. Il est assez courant de supprimer ce droit :

```
REVOKE CREATE ON SCHEMA public FROM nom_utilisateur ;
```

ou de manière globale :

```
REVOKE CREATE ON SCHEMA public FROM public ;
```

Certains DBA suppriment même le schéma de la base. Dans tous les cas, il faut faire attention à ce que le schéma ne réapparaisse pas lors d'une restauration de sauvegarde logique, car sa création est implicite.

L'utilisateur peut créer des objets temporaires sur chacune des bases de données où il peut se connecter. Il est possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE TEMP ON DATABASE nom_base FROM nom_utilisateur ;
```

L'utilisateur peut créer des fonctions, uniquement avec les langages de confiance, uniquement dans les schémas où il a le droit de créer des objets (donc dans le schéma `public` de toute base avant la V15). Il existe deux solutions :

- supprimer le droit d'utiliser un langage :

```
REVOKE USAGE ON LANGUAGE nom_langage FROM nom_utilisateur ;
```

- supprimer le droit de créer des objets dans un schéma :

```
REVOKE CREATE ON SCHEMA nom_schema FROM nom_utilisateur ;
```

L'utilisateur peut exécuter n'importe quelle fonction, y compris définie par quelqu'un d'autre, à condition que la fonction soit dans un schéma où il a accès (dont le schéma `public`, par défaut) (et donc `public`, par défaut) Il est possible d'empêcher cela en supprimant le droit d'exécution d'une fonction :

```
REVOKE EXECUTE ON FUNCTION nom_fonction FROM nom_utilisateur ;
```

5.7.2 Droits par défaut (suite)

Un utilisateur standard peut aussi :

- récupérer des informations sur l'instance
- visualiser les sources des vues et des fonctions
- Modifier des paramètres de la session :

```
SET parametre TO valeur ;
SET LOCAL parametre TO valeur ;
SHOW parametre ;

PGOPTIONS='-c param=valeur' psql
```

- Vue : `pg_settings`
- Dans `psql` : `\dconfig`

Il peut récupérer des informations sur l'instance car il a le droit de lire tous les catalogues systèmes. Par exemple, en lisant `pg_class`, il peut connaitre la liste des tables, vues, séquences, etc. En parcourant `pg_proc`, il dispose de la liste des fonctions. Il n'y a pas de contournement à cela : un utilisateur doit pouvoir accéder aux catalogues systèmes pour travailler normalement.

Il peut visualiser les sources des vues et des fonctions. Il existe des modules propriétaires de chiffrement (ou plutôt d'obfuscation) du code mais rien de natif. Le plus simple est certainement de coder les fonctions sensibles en C.

Un utilisateur peut agir sur de nombreux paramètres au sein de sa session pour modifier les valeurs par défaut du `postgresql.conf` ou ceux imposés à son rôle ou à sa base.

Un cas courant consiste à modifier la liste des schémas par défaut où chercher les tables :

```
SET search_path TO rh,admin,ventes,public ;
```

L'utilisateur peut aussi décider de s'octroyer plus de mémoire de tri :

```
SET work_mem TO '500MB' ;
```

Il est impossible d'interdire cela. Toutefois, cela permet de conserver un paramétrage par défaut prudent, tout en autorisant l'utilisation de plus de ressources quand cela s'avère nécessaire.

Les exemples suivants modifient le fuseau horaire du client, désactivent la parallélisation le temps de la session, et changent le nom de l'applicatif visible dans les outils de supervision :

```
SET timezone TO GMT ;
SET max_parallel_workers_per_gather TO 0 ;
SET application_name TO 'batch_comptabilite' ;
```

Pour une session lancée en ligne de commande, pour les outils qui utilisent la libpq, on peut fixer les paramètres à l'appel grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS="-c max_parallel_workers_per_gather=0 -c work_mem=4MB" psql < requete.sql
```

La valeur en cours est visible avec :

```
SHOW parametre ;
```

ou :

```
SELECT current_setting('parametre') ;
```

ou encore :

```
SELECT * FROM pg_settings WHERE name = 'parametre' ;
```

Sous `psql` à partir de la version 15 du client, la métacommande `\dconfig` affiche les paramètres qui n'ont pas leur valeur par défaut.

Ce paramétrage est limité à la session en cours, et disparaît avec elle à la déconnexion, ou si l'on demande un retour à la valeur par défaut :

```
RESET parametre ;
```

Enfin, on peut n'appliquer des paramètres que le temps d'une transaction, c'est-à-dire jusqu'au prochain `COMMIT` ou `ROLLBACK` :

```
SET LOCAL work_mem TO '100MB' ;
```

De nombreux paramètres sont cependant non modifiables, ou réservés aux superutilisateurs.

5.7.3 Droits par défaut (suite)

- Un utilisateur standard ne peut pas :
  - créer une base
  - créer un rôle
  - accéder au contenu des objets créés par d'autres
  - modifier le contenu d'objets créés par d'autres

Un utilisateur standard ne peut pas créer de bases et de rôles. Il a besoin pour cela d'attributs particuliers (respectivement `CREATEDB` et `CREATEROLE`).

Il ne peut pas accéder au contenu (aux données) d'objets créés par d'autres utilisateurs. Ces derniers doivent lui donner ce droit explicitement : `GRANT USAGE ON SCHEMA secret TO utilisateur ;` pour lire un schéma, ou `GRANT SELECT ON TABLE matable TO utilisateur ;` pour lire une table.

De même, il ne peut pas modifier le contenu et la définition d'objets créés par d'autres utilisateurs. Là-aussi, ce droit doit être donné explicitement : `GRANT INSERT,DELETE,UPDATE,TRUNCATE ON TABLE matable TO util`

Il existe d'autres droits plus rares, dont :

- `GRANT TRIGGER ON TABLE ...` autorise la création de trigger ;
- `GRANT REFERENCES ON TABLE ...` autorise la création d'une clé étrangère pointant vers cette table (ce qui est interdit par défaut car cela interdit au propriétaire de supprimer ou modifier des lignes, entre autres) ;
- `GRANT USAGE ON SEQUENCE...` autorise l'utilisation d'une séquence.

Par facilité, on peut octroyer des droits en masse :

- `GRANT ALL PRIVILEGES ON TABLE matable TO utilisateur ;`
- `GRANT SELECT ON TABLE matable TO public ;`
- `GRANT SELECT ON ALL TABLES IN SCHEMA monschema ;`

5.7.4 Restreindre les droits

- Sur les connexions
  - `pg_hba.conf`
- Sur les objets
  - `GRANT` / `REVOKE`
  - `SECURITY LABEL`
- Sur les fonctions
  - `SECURITY DEFINER`
  - `LEAKPROOF`
- Sur les vues
  - `security_barrier`
  - `WITH CHECK OPTION`

Pour sécuriser plus fortement une instance, il est nécessaire de restreindre les droits des utilisateurs.

Connexions :

Cela commence par la gestion des connexions. Les droits de connexion sont généralement gérés via le fichier de configuration `pg_hba.conf`. Cette configuration a déjà été abordée dans le chapitre *Droits*

de connexion de ce module de formation.

GRANT & REVOKE :

Cela passe ensuite par les droits sur les objets. On dispose pour cela des instructions `GRANT` et `REVOKE`, qui ont été expliquées dans le chapitre *Droits sur les objets* de ce module de formation.

SECURITY LABEL :

Il est possible d'aller plus loin avec l'instruction `SECURITY LABEL`. Un label de sécurité est un commentaire supplémentaire pris en compte par un module de sécurité qui disposera de la politique de sécurité. Le seul module de sécurité actuellement disponible est un module contrib pour l'intégration à SELinux, appelé sepgsql[13].

SECURITY DEFINER & LEAKPROOF :

Certains objets disposent de droits particuliers. Par exemple, les fonctions disposent des clauses `SECURITY DEFINER` et `LEAKPROOF`.

`SECURITY DEFINER` indique au système que la fonction doit s'exécuter avec les droits de son propriétaire (et non pas avec ceux de l'utilisateur l'exécutant). Cela permet d'éviter de donner des droits d'accès à certains objets.

`LEAKPROOF` permet de dire à PostgreSQL que cette fonction ne peut pas occasionner de fuites d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut optimiser l'exécution des fonctions.

Vues avec security_barrier :

Quant aux vues, elles disposent d'une option appelée `security_barrier`. Certains utilisateurs créent des vues pour filtrer des lignes, afin de restreindre la visibilité sur certaines données. Or, cela peut se révéler dangereux si un utilisateur malintentionné a la possibilité de créer une fonction car il peut facilement contourner cette sécurité si cette option n'est pas utilisée. Voici un exemple complet :

```
CREATE TABLE elements (id int, contenu text, prive boolean);
```

```
CREATE TABLE
```

```
INSERT INTO elements (id, contenu, prive)
VALUES (1, 'a', false), (2, 'b', false), (3, 'c super prive', true),
       (4, 'd', false), (5, 'e prive aussi', true) ;
```

```
INSERT 0 5
```

```
SELECT * FROM elements ;
```

```
 id |    contenu     | prive
----+----------------+-------
  1 | a              | f
  2 | b              | f
  3 | c super prive  | t
  4 | d              | f
  5 | e prive aussi  | t
```

[13] https://docs.postgresql.fr/current/sepgsql.html

La table `elements` contient cinq lignes, dont deux considérées comme privées. Nous allons donc créer une vue ne permettant de voir que les lignes publiques, sauf pour un utilisateur particulier :

```
CREATE OR REPLACE VIEW elements_public AS
  SELECT * FROM elements
  WHERE CASE WHEN current_user = 'guillaume' THEN TRUE
             ELSE NOT prive
        END ;

CREATE VIEW
```

Ce nouvel utilisateur ne verra donc pas les lignes privées masquées par la vue :

```
CREATE USER u1;

CREATE ROLE

GRANT SELECT ON elements_public TO u1;

GRANT

\c - u1

You are now connected to database "demo" as user "u1".

SELECT * FROM elements ;

ERROR:  permission denied for relation elements

SELECT * FROM elements_public ;

 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

L'utilisateur **u1** n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Avec une simple fonction, **u1** peut contourner le problème. (Il devra toutefois avoir le droit d'écrire dans un schéma de la base, ce qui est par défaut possible dans `public` avant la version 15.)

```
CREATE SCHEMA schemau1;
CREATE OR REPLACE FUNCTION schemau1.abracadabra (integer, text, boolean)
RETURNS bool AS $$
BEGIN
  RAISE NOTICE '% - % - %', $1, $2, $3 ;
  RETURN true;
END$$
LANGUAGE plpgsql
COST 0.00000000000000000001 ;

CREATE SCHEMA
CREATE FUNCTION

SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive) ;
```

```
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

Les lignes secrètes ne font toujours pas partie du résultat, mais la fonction est capable d'y accéder, et de les afficher. Que s'est-il passé ? Pour comprendre, il suffit de regarder le plan d'exécution de cette requête :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive) ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.75 rows=208 width=37)
   Filter: (schemau1.abracadabra(id, contenu, prive)
           AND CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
 ↪  prive) END)
```

La fonction `abracadrabra` a un coût délibérément si faible que PostgreSQL l'exécute avant le filtre de la vue : la fonction voit toutes les lignes de la table.

L'option `security_barrier` permet d'interdire cette optimisation au planificateur :

```
\c - postgres
```

```
You are now connected to database "demo" as user "postgres".
```

```
ALTER VIEW elements_public SET (security_barrier);
```

```
ALTER VIEW
```

```
\c - u1
```

```
You are now connected to database "demo" as user "u1".
```

```
SELECT * FROM elements_public WHEREschemau1.abracadabra(id, contenu, prive);
```

```
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

La fonction n'affiche plus les données privées. Son plan montre qu'elle filtre par la vue, puis appelle la fonction :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive);
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..35.00 rows=208 width=37)
   Filter: schemau1.abracadabra(elements_public.id, elements_public.contenu,
↪  elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.75 rows=625 width=37)
         Filter: CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
↪  prive) END
```

Noter que `security_barrier` peut avoir un effet négatif sur les performances puisqu'il interdit de « pousser » des critères de recherche dans la vue. Voir aussi cet article, par Robert Haas[14].

Vue WITH CHECK OPTION :

Écrire à travers une vue permet de modifier les lignes même si la nouvelle valeur les lui rend inaccessibles :

```
UPDATE elements_public
SET prive = true
WHERE id = 1 ;

UPDATE 1

SELECT * FROM elements_public ;

 id | contenu | prive
----+---------+-------
  2 | b       | f
  4 | d       | f
```

Pour bloquer ce comportement, la vue peut porter l'option `WITH CHECK OPTION` :

```
CREATE OR REPLACE VIEW elements_public
WITH (security_barrier)
AS
  SELECT * FROM elements
  WHERE CASE WHEN current_user = 'guillaume'
  THEN true ELSE NOT prive END
WITH CHECK OPTION ;

\c - u1

You are now connected to database "demo" as user "u1".

UPDATE elements_public
SET prive = true
WHERE id = 2 ;

ERROR:  new row violates check option for view "elements_public"
DETAIL:  Failing row contains (prive) = (t).
```

[14] https://rhaas.blogspot.com/2012/03/security-barrier-views.html

5.7.5 Arrêter une requête ou une session

- Annuler une requête
  - `pg_cancel_backend (pid int)`
- Fermer une connexion
  - `pg_terminate_backend(pid int, timeout bigint)`
  - `kill -SIGTERM pid`, `kill -15 pid` (éviter)
- Jamais `kill -9` !!

Les fonctions `pg_cancel_backend` et `pg_terminate_backend` sont le plus souvent utilisées. Le paramètre est le numéro du processus auprès de l'OS. À partir de la version 14, `pg_terminate_backend` comprend un deuxième argument, dont la valeur par défaut est 0. Si cet argument n'est pas indiqué ou vaut 0, la fonction renvoie le booléen `true` si elle a réussi à envoyer le signal. Ce résultat n'indique donc pas la bonne terminaison du processus serveur visé. À une valeur supérieure à 0, la fonction attend que le processus visé soit arrêté. S'il ne s'est pas arrêté dans le temps indiqué par cette valeur (en millisecondes), la valeur `false` est renvoyée avec un message de niveau `WARNING`.

La première permet d'annuler une requête en cours d'exécution. Elle requiert un argument, à savoir le numéro du PID du processus `postgres` exécutant cette requête. Généralement, l'annulation est immédiate. Voici un exemple de son utilisation.

L'utilisateur, connecté au processus de PID 10901 comme l'indique la fonction `pg_backend_pid`, exécute une très grosse insertion :

```
b1=# SELECT pg_backend_pid();

 pg_backend_pid
----------------
          10901

b1=# INSERT INTO t4 SELECT i, 'Ligne '||i
FROM generate_series(2000001, 3000000) AS i;
```

Supposons qu'on veuille annuler l'exécution de cette requête. Voici comment faire à partir d'une autre connexion :

```
b1=# SELECT pg_cancel_backend(10901);

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

L'utilisateur qui a lancé la requête d'insertion verra ce message apparaître :

```
ERROR:  canceling statement due to user request
```

Si la requête du `INSERT` faisait partie d'une transaction, la transaction elle-même devra se conclure par un `ROLLBACK` à cause de l'erreur. À noter cependant qu'il n'est pas possible d'annuler une transaction qui n'exécute rien à ce moment. En conséquence, `pg_cancel_backend` ne suffit pas pour parer à une session en statut `idle in transaction`.

Il est possible d'aller plus loin en supprimant la connexion d'un utilisateur. Cela se fait avec la fonction `pg_terminate_backend` qui se manie de la même manière :

```
b1=# SELECT pid, datname, usename, application_name, state
     FROM pg_stat_activity WHERE backend_type = 'client backend';

 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   13267 | b1      | u1        | psql             | idle
   10901 | b1      | guillaume | psql             | active

b1=# SELECT pg_terminate_backend(13267);

 pg_terminate_backend
----------------------
 t

b1=# SELECT pid, datname, usename, application_name, state
     FROM pg_stat_activity WHERE backend_type = 'client backend';

 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   10901 | b1      | guillaume | psql             | active
```

L'utilisateur de la session supprimée verra un message d'erreur au prochain ordre qu'il enverra. `psql` se reconnecte automatiquement mais cela n'est pas forcément le cas d'autres outils client.

```
b1=# select 1 ;

FATAL:  terminating connection due to administrator command
la connexion au serveur a été coupée de façon inattendue
        Le serveur s'est peut-être arrêté anormalement avant ou durant le
        traitement de la requête.
La connexion au serveur a été perdue. Tentative de réinitialisation : Succès.
Temps : 7,309 ms
```

Depuis la ligne de commande du serveur, un `kill <pid>` (c'est-à-dire `kill -SIGTERM` ou `kill -15`) a le même effet qu'un `SELECT pg_terminate_backend (<pid>)`. Cette méthode n'est pas recommandée car il n'y a pas de vérification que vous tuez bien un processus postgres.

N'utilisez jamais `kill -9 <pid>` (ou `kill -SIGKILL`), ou (sous Windows) `taskkill /f /pid <pid>` pour tuer une connexion : l'arrêt est alors brutal, et le processus principal n'a aucun moyen de savoir pourquoi. Pour éviter une corruption de la mémoire partagée, il va arrêter et redémarrer immédiatement tous les processus, déconnectant tous les utilisateurs au passage !

L'utilisation de `pg_terminate_backend` et `pg_cancel_backend` n'est disponible que pour les utilisateurs appartenant au même rôle que l'utilisateur à déconnecter, les utilisateurs membres du rôle `pg_signal_backend` (à partir de la 9.6) et bien sûr les superutilisateurs.

5.7.6 Chiffrements

- Connexions
  - SSL
  - avec ou sans certificats serveur et/ou client
- Données disques
  - pas en natif
  - par colonne : pgcrypto

Par défaut, les sessions ne sont pas chiffrées. Les requêtes et les données passent donc en clair sur le réseau. Il est possible de les chiffrer avec SSL, ce qui aura une conséquence négative sur les performances. Il est aussi possible d'utiliser les certificats (au niveau serveur et/ou client) pour augmenter encore la sécurité des connexions.

PostgreSQL ne chiffre pas les données sur disque. Si l'instance complète doit être chiffrée, il est conseillé d'utiliser un système de fichiers qui propose cette fonctionnalité. Attention au fait que cela ne vous protège que contre la récupération des données sur un disque non monté. Quand le disque est monté, les données sont lisibles suivant les règles d'accès d'Unix.

Néanmoins, il existe un module contrib appelé pgcrypto, permettant d'accéder à des fonctions de chiffrement et de hachage. Cela permet de protéger les informations provenant de colonnes spécifiques. Le chiffrement se fait du côté serveur, ce qui sous-entend que l'information est envoyée en clair sur le réseau. Le chiffrement SSL est donc obligatoire dans ce cas.

5.7.7 En cas de crash

- Ça arrive
- Redémarrage sans souci, en général
- Quelques statistiques d'activité sont perdues
  - `ANALYZE` (voire `VACUUM`)
- Et analyser la cause

Même si PostgreSQL est conçu pour être le plus robuste possible, un arrêt brutal est toujours possible, pour des causes externes (perte de courant, plantage du système d'exploitation, corruption de fichier

sur le disque…), liées à l'utilisation de PostgreSQL (requêtes menant à une saturation mémoire sur un système mal paramétré…) ou exceptionnellement internes (extension buggée, bug de PostgreSQL).

Dans l'écrasante majorité des cas, le système de journalisation de PostgreSQL vous permettra de redémarrer sans perte de données (sinon ce qui n'a pas été committé). Le rejeu des journaux est souvent si rapide qu'un redémarrage peut passer inaperçu.

Malheureusement, les statistiques d'activités sont perdues lors d'un arrêt brutal (ou d'un arrêt en mode *immediate*, ou d'une restauration physique). Cela peut provoquer par la suite un retard dans le déclenchement de l'autovacuum et la mise à jour des statistiques. Pour éviter cela, le DBA doit penser à relancer un ordre `ANALYZE`, ainsi que, dans l'idéal, un ordre `VACUUM`.

Surtout, il faut comprendre quelle est la cause pour que le problème ne se reproduise pas.

5.7.8 Corruption de données

- `initdb --data-checksums`
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Fortement conseillé !

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.

L'apparition des sommes de contrôles (*checksums*) permet de se prémunir contre des corruptions silencieuses de données. Leur mise en place est fortement recommandée sur une nouvelle instance.

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

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

Insérons une valeur de test, sur chacune des deux instances :

```
postgres=# CREATE TABLE test (name text);

CREATE TABLE

postgres=# INSERT INTO test (name) VALUES ('toto');

INSERT 0 1
```

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

```
postgres=# SELECT pg_relation_filepath('test');
```

```
 pg_relation_filepath
----------------------
 base/14415/16384
```

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/base/14415/16384
```

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

Sans *checksums* :

```
postgres=# TABLE test;

 name
------
 goto
```

Avec *checksums* :

```
postgres=# TABLE test;

WARNING:  page verification failed, calculated checksum 29393
          but expected 24228
ERROR:  invalid page in block 0 of relation base/14415/16384
```

L'outil `pg_verify_checksums`, disponible depuis la version 11 et renommé `pg_checksums` en 12, permet de vérifier une instance complète :

```
$ pg_checksums -D /tmp/avec_checksums

pg_checksums: error: checksum verification failed
  in file "/tmp/avec_checksums/base/14415/16384",
  block 0: calculated checksum 72D1 but block contains 5EA4
Checksum operation completed
Files scanned:  919
Blocks scanned: 3089
Bad checksums:  1
Data checksum version: 1
```

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.

L'outil `pg_checksums` permet aussi d'activer et de désactiver la gestion des sommes de contrôle (instance arrêtée). Ceci n'était pas possible avant la version 12. Il fallait donc le faire dès la création de l'instance.

5.8 CONCLUSION

- PostgreSQL demande peu de travail au quotidien
- À l'installation :
  - veiller aux accès et aux droits
  - mettre la maintenance en place
- Pour le reste, surveiller :
  - les scripts automatisés
  - le contenu des journaux applicatifs
- Supervisez le serveur !

5.8.1 Pour aller plus loin

- Documentation officielle, chapitre Planifier les tâches de maintenance[15]
- Documentation officielle, chapitre Catalogues système[16]
- Opérations de maintenance sous PostgreSQL[17]
- Total security in a PostgreSQL database (copie)[18]
- Managing rights in PostgreSQL, from the basics to SE PostgreSQL[19], Nicolas Thauvin, 2011

5.8.2 Questions

N'hésitez pas, c'est le moment !

5.9 QUIZ

https://dali.bo/f_quiz

6/ PostgreSQL : Politique de sauvegarde

Photo de l'incendie du datacenter OVHcloud à Strasbourg du 10 mars 2021 fournie gracieusement par l'ITBR67[1].

Cet incendie a provoqué de nombreux arrêts et pertes de données dans toute la France et ailleurs[2].

[1] https://itbr67.fr/
[2] https://fr.wikipedia.org/wiki/Incendie_du_centre_de_donn%C3%A9es_d%27OVHcloud_%C3%A0_Strasbourg

6.1 INTRODUCTION

- Le pire peut arriver
- Politique de sauvegarde

6.1.1 Au menu

- Objectifs
- Approche
- Points d'attention

6.2 DÉFINIR UNE POLITIQUE DE SAUVEGARDE

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

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

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

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

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

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

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

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

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

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

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

6.2.1 Objectifs

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

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

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

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

6.2.2 Différentes approches

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

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

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

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

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

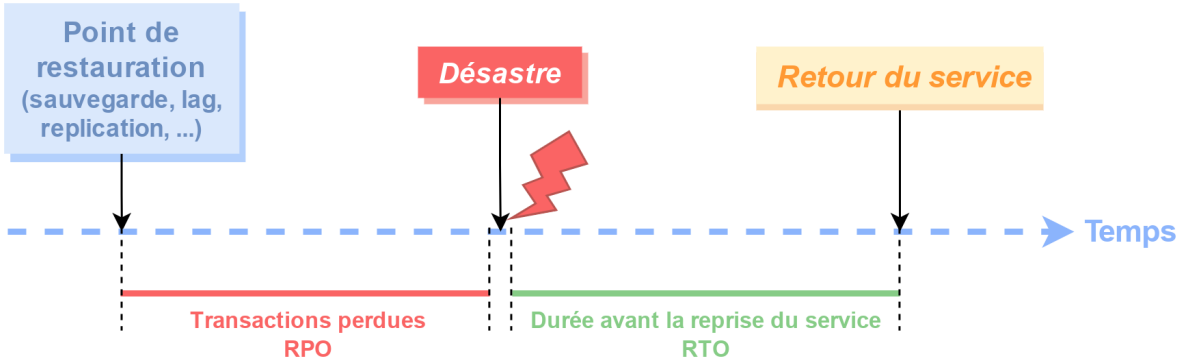
Suivant les prérequis et les limitations de chaque méthode, il est fort possible qu'une seule de ces solutions soit utilisable. Par exemple :

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

6.2.3 RTO/RPO

La politique de sauvegarde découle du :

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



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

La **RPO** (ou PDMA[3]) est la perte de données maximale admissible, ou quantité de données que l'on peut tolérer de perdre lors d'un sinistre majeur, souvent exprimée en heures ou minutes.

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

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

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

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

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

La **RTO** (ou DMIA[4]) est la durée maximale d'interruption du service.

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

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

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

- réplication ;
- solutions HA (Haute Disponibilité).

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

6.2.4 Industrialisation

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

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

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

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

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

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

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

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

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

6.2.5 Documentation

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

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

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

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

6.2.6 Règle 3-2-1

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

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

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

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

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

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

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

Stocker vos données dans le *cloud* n'est pas une solution miracle. Un datacenter peut brûler entièrement[5]. Dans la formule choisie, il faut donc bien vérifier que le fournisseur sauvegarde les données

[5] https://dali.bo/20210310-ovh-incendie-strasbourg

sur un autre site. Mais on a aussi vu un hébergeur perdre *toutes* les données de ses clients à cause d'un ransomware[6].

N'hésitez donc pas à faire vous-même une copie de vos données.
Il ne faut pas non plus négliger le risque d'une attaque (piratage, malveillance ou *ransomware*…), qui s'en prendra à toute sauvegarde accessible en ligne. Une copie physique hors ligne est donc chaudement recommandée pour les données les plus critiques.

6.2.7 Autres points d'attention

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

La sauvegarde ne concerne pas uniquement les données. Il est également fortement conseillé de sauvegarder les fichiers de configuration du serveur et les scripts d'administration. Ils sont parfois générés par l'outil d'industrialisation, gérés par un outil externe (Patroni…), ou versionnés ailleurs. L'idéal est de les copier avec les sauvegardes. On peut également déléguer cette tâche à une sauvegarde au niveau système, vu que ce sont de simples fichiers. Les principaux fichiers de PostgreSQL à prendre en compte sont : `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`. Ces fichiers ont parfois des clauses d'inclusion pour en appeler d'autres. Cette liste n'est en aucun cas exhaustive.

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

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

[6] https://dcmag.fr/cloudnordic-le-datacenter-qui-a-perdu-toutes-les-donnees-de-ses-clients-lors-dune-attaque-par-ransomware/

Just that backup tapes are seen to move, backup scripts are run for a lengthy period of time, should not be construed as verifying that data backups are properly being performed.
Que l'on voit bouger les bandes de sauvegardes, ou que les scripts de sauvegarde fonctionnent pendant une longue période, ne doit pas être interprété comme une validation que les sauvegardes sont faites.

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

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

- une base disparaît à l'arrêt et ses sauvegardes sont vides[7] : erreur de manipulation, script ne vérifiant pas qu'il a bien fait sa tâche, monitoring défaillant ;
- perte de 6 heures de données chez Gitlab en 2017[8] : procédures de sauvegarde et de réplication incomplètes, complexes et peu claires, outils mal choisis ou mal connus, sauvegardes vides et jamais testées, distraction d'un opérateur — Gitlab a le mérite d'avoir tout soigneusement documenté[9].

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

[7] http://www.pgdba.org/post/restoring_data/#a-database-horror-story
[8] https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/
[9] https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/

6.3 CONCLUSION

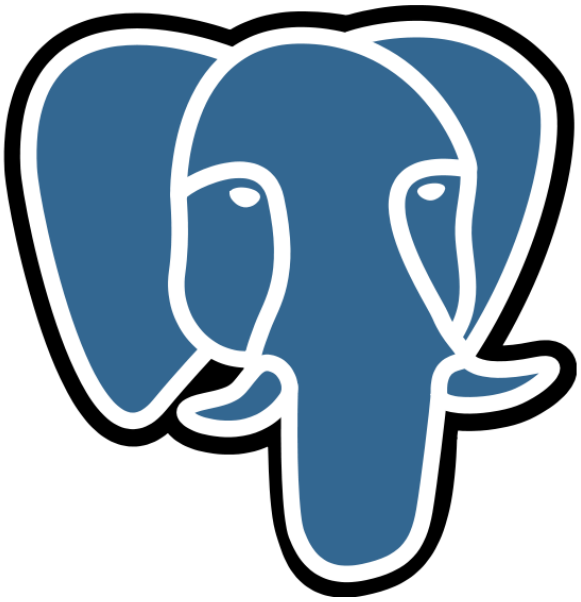
- Les techniques de sauvegarde de PostgreSQL sont :
  - complémentaires
  - automatisables
- La maîtrise de ces techniques est indispensable pour assurer un service fiable.
- **Testez vos sauvegardes !**

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

6.4 QUIZ

https://dali.bo/i0_quiz

7/ PostgreSQL : Sauvegarde et restauration



7.1 INTRODUCTION

- Opération essentielle pour la sécurisation des données
- PostgreSQL propose différentes solutions
  - de sauvegarde à froid ou à chaud, mais cohérentes
  - des méthodes de restauration partielle ou complète

La mise en place d'une solution de sauvegarde est une des opérations les plus importantes après avoir installé un serveur PostgreSQL. En effet, nul n'est à l'abri d'un bogue logiciel, d'une panne matérielle, voire d'une erreur humaine.

Cette opération est néanmoins plus complexe qu'une sauvegarde standard car elle doit pouvoir s'adapter aux besoins des utilisateurs. Quand le serveur ne peut jamais être arrêté, la sauvegarde à froid des fichiers ne peut convenir. Il faudra passer dans ce cas par un outil qui pourra sauvegarder les données alors que les utilisateurs travaillent et qui devra respecter les contraintes ACID pour fournir une sauvegarde cohérente des données.

PostgreSQL va donc proposer des méthodes de sauvegardes à froid (autrement dit serveur arrêté) comme à chaud, mais de toute façon cohérentes. Les sauvegardes pourront être partielles ou complètes, suivant le besoin des utilisateurs.

La méthode de sauvegarde dictera l'outil de restauration. Suivant l'outil, il fonctionnera à froid ou à chaud, et permettra même dans certains cas de faire une restauration partielle.

7.1.1 Au menu

- Sauvegardes logiques
- Sauvegarde physique à froid des fichiers

7.2 SAUVEGARDES LOGIQUES

- À chaud
- Cohérente
- Locale ou à distance
- 2 outils
  - `pg_dump`
  - `pg_dumpall`
- Pas d'impact sur les utilisateurs
  - sauf certaines opérations DDL
- Jamais inclus :
  - tables systèmes
  - fichiers de configuration

La sauvegarde logique nécessite que le serveur soit en cours d'exécution. Un outil se connecte à la base et récupère la déclaration des différents objets ainsi que les données des tables.

La technique alors utilisée permet de s'assurer de la cohérence des données : lors de la sauvegarde, l'outil ne voit pas les modifications faites par les autres utilisateurs. Pour cela, quand il se connecte à la base à sauvegarder, il commence une transaction pour que sa vision des enregistrements de l'ensemble des tables soit cohérente. Cela empêche le recyclage des enregistrements par `VACUUM` pour les enregistrements dont il pourrait avoir besoin. Par conséquent, que la sauvegarde dure 10 minutes ou 10 heures, le résultat correspondra au contenu de la base telle qu'elle était au début de la transaction.

Des verrous sont placés sur chaque table, mais leur niveau est très faible (*Access Share*). Il visent juste à éviter la suppression des tables pendant la sauvegarde, ou la modification de leur structure. Les opérations habituelles sont toutes permises en lecture ou écriture, sauf quand elles réclament un verrou très invasif, comme `TRUNCATE`, `VACUUM FULL` ou certains `LOCK TABLE`. Les verrous ne sont relâchés qu'à la fin de la sauvegarde.

Par ailleurs, pour assurer une vision cohérente de la base durant toute la durée de son export, cette transaction de longue durée est de type *REPEATABLE READ* et non de type *READ COMMITED* utilisé par défaut.

Il existe deux outils de ce type pour la sauvegarde logique dans la distribution officielle de PostgreSQL :

- `pg_dump`, pour sauvegarder une base (complètement ou partiellement) ;

- `pg_dumpall` pour sauvegarder toutes les bases ainsi que les objets globaux.

`pg_dump` permet d'extraire le contenu d'une seule base de données dans différents formats. (Pour rappel, une instance PostgreSQL contient plusieurs bases de données.)

`pg_dumpall` permet d'extraire le contenu d'une instance en totalité au format texte. Il s'agit des données globales (rôles, tablespaces), de la définition des bases de données et de leur contenu.

`psql` exécute les ordres SQL contenus dans des *dumps* (sauvegardes) au format texte.

`pg_restore` traite uniquement les *dumps* au format binaire, et produit le SQL qui permet de restaurer les données.

Il est important de bien comprendre que ces outils n'échappent pas au fonctionnement client-serveur de PostgreSQL. Ils « dialoguent » avec l'instance PostgreSQL uniquement en SQL, aussi bien pour le dump que la restore.

Comme ce type d'outil n'a besoin que d'une connexion standard à la base de données, il peut se connecter en local comme à distance. Cela implique qu'il doive aussi respecter les autorisations de connexion configurées dans le fichier `pg_hba.conf`.

L'export ne concerne que les données des utilisateurs : les tables systèmes ne sont jamais concernées, il n'y a pas de risque de les écraser lors d'un import. En effet, les schémas systèmes `pg_catalog` et `information_schema` et leurs objets sont gérés uniquement par PostgreSQL. Vous n'êtes d'ailleurs pas censé modifier leur contenu, ni y ajouter ou y effacer quoi que ce soit !

La configuration du serveur (fichiers `postgresql.conf`, `pg_hba.conf`…) n'est jamais incluse et doit être sauvegardée à part. Un export logique ne concerne que des données.

Les extensions ne posent pas de problème non plus : la sauvegarde contiendra une mention de l'extension, et les *données* des éventuelles tables gérées par cette extension. Il faudra que les binaires de l'extension soient installés sur le système cible.

7.2.1 pg_dump

- Sauvegarde une base de données :

```
pg_dump nombase > nombase.dump
```

- Sauvegarde complète ou partielle

`pg_dump` est l'outil le plus utilisé pour sauvegarder une base de données PostgreSQL. Une sauvegarde peut se faire de façon très simple. Par exemple :

```
$ pg_dump b1 > b1.dump
```

sauvegardera la base **b1** de l'instance locale sur le port 5432 dans un fichier `b1.dump`.

Sous Windows avec le Powershell, préférer la syntaxe `pg_dump -f b1.dump b1`, car une redirection avec `>` peut corrompre la sauvegarde.

Mais `pg_dump` permet d'aller bien plus loin que la sauvegarde d'une base de données complète. Il existe pour cela de nombreuses options en ligne de commande.

7.2.2 pg_dump - Format de sortie

Format	Dump	Restore
plain (SQL)	`pg_dump -Fp` ou `pg_dumpall`	`psql`
tar	`pg_dump -Ft`	`pg_restore`
custom	`pg_dump -Fc`	`pg_restore`
directory	`pg_dump -Fd`	`pg_restore`

Un élément important est le format des données extraites. Selon l'outil de sauvegarde utilisé et les options de la commande, l'outil de restauration diffère. Le tableau indique les outils compatibles selon le format choisi.

`pg_dump` accepte d'enregistrer la sauvegarde suivant quatre formats :

- le format SQL, soit un fichier texte unique pour toute la base, non compressé ;

- le format tar, un fichier binaire, non compressé, comprenant un index des objets ;
- le format « personnalisé » (*custom*), un fichier binaire, compressé, avec un index des objets ;
- le format « répertoire » (*directory*), arborescence de fichiers binaires généralement compressés, comprenant aussi un index des objets.

Pour choisir le format, il faut utiliser l'option `--format` (ou `-F`) et le faire suivre par le nom ou le caractère indiquant le format sélectionné :

- `plain` ou `p` pour un fichier SQL (texte) ;
- `tar` ou `t` pour un fichier tar ;
- `custom` ou `c` pour un fichier « personnalisé » ;
- `directory` ou `d` pour le répertoire.

Le format `plain` est lisible directement par psql. Les autres nécessitent de passer par `pg_restore` pour restaurer tout ou partie de la sauvegarde.

Le fichier SQL (`plain`) est naturellement lisible par n'importe quel éditeur texte. Le fichier texte est divisé en plusieurs parties :

- configuration de certaines variables ;
- création des objets de la base : schémas, tables, vues, procédures stockées, etc., à l'exception des index, contraintes et triggers ;
- ajout des données aux tables (ordres `COPY` par défaut) ;
- ajout des index, contraintes et triggers ;
- définition des droits d'accès aux objets ;
- rafraîchissement des vues matérialisées.

Les index figurent vers la fin pour des raisons de performance : il est plus rapide de créer un index à partir des données finales que de le mettre à jour en permanence pendant l'ajout des données. Les contraintes et le rafraîchissement des vues matérialisées sont aussi à la fin parce qu'il faut que les données soient déjà restaurées dans leur ensemble. Les triggers ne devant pas être déclenchés pendant la restauration, ils sont aussi restaurés vers la fin. Les propriétaires sont restaurés pour chacun des objets.

Voici un exemple de sauvegarde d'une base de 2 Go pour chaque format :

```
$ time pg_dump -Fp b1 > b1.Fp.dump
real  0m33.523s
user  0m10.990s
sys   0m1.625s

$ time pg_dump -Ft b1 > b1.Ft.dump
real  0m37.478s
user  0m10.387s
sys   0m2.285s

$ time pg_dump -Fc b1 > b1.Fc.dump
real  0m41.070s
user  0m34.368s
sys   0m0.791s
```

```
$ time pg_dump -Fd -f b1.Fd.dump b1
real  0m38.085s
user  0m30.674s
sys   0m0.650s
```

La sauvegarde la plus longue est la sauvegarde au format personnalisée (`custom`) car elle est compressée. La sauvegarde au format répertoire se trouve entre la sauvegarde au format personnalisée et la sauvegarde au format `tar` : elle est aussi compressée mais sur des fichiers plus petits.

En terme de taille :

```
$ du -sh b1.F?.dump
116M    b1.Fc.dump
116M    b1.Fd.dump
379M    b1.Fp.dump
379M    b1.Ft.dump
```

Le format compressé est évidemment le plus petit. Le format texte et le format tar[1] sont les plus lourds à cause du manque de compression. Le format tar est même généralement un peu plus lourd que le format texte à cause de l'entête des fichiers tar.

7.2.3 Choix du format de sortie

- Format `plain` (SQL)
  - restaurations partielles très difficiles (ou manuelles)
- Parallélisation du dump
  - uniquement format `directory`
- Utilisez les formats binaires

```
pg_dump -Fc
pg_dump -Fd
```

- Et en compléments, les objets globaux

```
pg_dumpall -g
```

Il convient de bien appréhender les limites de chaque outil de dump et des formats.

Tout d'abord, le format `tar` est à éviter. Il n'apporte aucune plus-value par rapport au format `custom`.

[1] https://fr.wikipedia.org/wiki/Tar_(informatique)

Ensuite, même si c'est le plus portable (et le seul disponible avec `pg_dumpall`), le format `plain` rend les restaurations partielles difficiles car il faut extraire manuellement le SQL d'un fichier texte souvent très volumineux. Ce peut être ponctuellement pratique cependant, mais on peut aisément regénérer un fichier SQL complet à partir d'une sauvegarde binaire et `pg_restore`.

Certaines informations (notamment les commandes `ALTER DATABASE … SET` pour modifier un paramètre pour une base précise) ne sont pas générées par `pg_dump -Fp`, à moins de penser à rajouter `--create` (pour les ordres de création). Par contre, elles sont incluses dans les entêtes des formats `custom` ou `directory`, où un `pg_restore --create` saura les retrouver.

On privilégiera donc les formats `custom` et `directory` pour plus de flexibilité à la restauration.

Le format `directory` ne doit pas être négligé : il permet d'utiliser la fonctionnalité de sauvegarde en parallèle de `pg_dump --jobs`, avec de gros gains de temps d'exécution (ou de migration) à la clé.

Enfin, l'outil `pg_dumpall`, initialement prévu pour les montées de versions majeures, permet de sauvegarder les objets globaux d'une instance : la définition des rôles et des tablespaces.

Ainsi, pour avoir la sauvegarde la plus complète possible d'une instance, il faut combiner `pg_dumpall -g` (pour la définition des objets globaux), et `pg_dump` (pour sauvegarder les bases de données une par une au format `custom` ou `directory`).

7.2.4 pg_dump - Compression

- `-z/--compress` : compression par zlib
  - de 0 à 9 (défaut : 6)
- (v16) mieux : `zstd`, `lz4`

```
pg_dump -Z1  # gzip
# v16+
pg_dump -Z gzip:3
pg_dump -Z lz4:'level=12'
pg_dump -Z zstd:'level=22,long'
```

La compression permet de réduire énormément la volumétrie d'une sauvegarde logique par rapport à la taille physique des fichiers de données.

Par défaut, `pg_dump -Fc` ou `-Fd` utilise le niveau de compression par défaut de la zlib, à priori le meilleur compromis entre compression et vitesse, correspondant à la valeur 6. `-Z1` comprimera peu mais rapidement, et `-Z9` sera nettement plus lent mais compressera au maximum. Seuls des tests permettent de déterminer le niveau acceptable pour un cas d'utilisation particulier.

Depuis PostgreSQL 16, des algorithmes de compression plus modernes sont disponibles : `zstd` et `lz4` sont généralement préférables. Une nouvelle syntaxe de l'option `-Z` permet de choisir l'algorithme, le niveau de compression (1 à 22 pour `zstd`, 1 à 12 pour `lz4`), éventuellement d'autres paramètres propres à l'algorithme (comme le mode `long` de `zstd`, plus consommateur en RAM). `lz4` a la réputation d'être très rapide mais bien moins bon en compression. Cependant, le choix du bon algorithme doit se faire en testant sur vos données réelles sur votre machine avec vos contraintes en CPU, RAM, place disque et temps.

Le format `plain` (pur texte) accepte aussi l'option `-Z`, ce qui permet d'obtenir un export texte compressé. Cependant, cela ne remplace pas complètement un format `custom`, plus souple.

Même les nouveaux algorithmes sont par défaut monothread. Si l'on veut paralléliser la compression sur plusieurs processeurs, nous verrons plus bas qu'il faut soit paralléliser `pg_dump` avec `--jobs`, soit compresser la sortie *plain* avec un autre outil.

7.2.5 pg_dump - Fichier ou sortie standard

- `-f` : fichier où stocker la sauvegarde
- sinon : sortie standard

Par défaut, et en dehors du format répertoire, toutes les données d'une sauvegarde sont renvoyées sur la sortie standard de `pg_dump`. Il faut donc utiliser une redirection pour renvoyer dans un fichier.

Cependant, il est aussi possible d'utiliser l'option `-f` pour spécifier le fichier de la sauvegarde. L'utilisation de cette option est conseillée car elle permet à `pg_restore` de trouver plus efficacement les objets à restaurer dans le cadre d'une restauration partielle.

7.2.6 pg_dump - Structure ou données ?

- `--schema-only` / `-s` : uniquement la structure
- `--data-only` / `-a` : uniquement les données

Il est possible de ne sauvegarder que la structure avec l'option `--schema-only` (ou `-s`). De cette façon, seules les requêtes de création d'objets seront générées. Cette sauvegarde est généralement très rapide. Cela permet de créer un serveur de tests très facilement.

Il est aussi possible de ne sauvegarder que les données pour les réinjecter dans une base préalablement créée avec l'option `--data-only` (ou `-a`).

7.2.7 pg_dump - Sélection de sections

- `--section`
  - `pre-data` : définition des objets (hors contraintes et index)
  - `data` : les données
  - `post-data` : définition des contraintes et index

Il est possible de sauvegarder une base par section. En fait, un fichier de sauvegarde complet est composé de trois sections : la définition des objets, les données, la définition des contraintes et index.

Il est parfois intéressant de sauvegarder par section plutôt que de sauvegarder schéma et données séparément car cela permet d'avoir la partie contrainte et index dans un script à part, ce qui accélère la restauration.

7.2.8 pg_dump - Sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf celui-là
- `-t <table>` : uniquement cette relation (sans dépendances !)
- `-T <table>` : toutes les tables sauf celle-là
- En option
  - possibilité d'en mettre plusieurs
  - exclure les données : `--exclude-table-data=<table>`
  - avoir une erreur si l'objet est inconnu : `--strict-names`

En dehors de la distinction structure/données, il est possible de demander de ne sauvegarder qu'un objet. Les seuls objets sélectionnables au niveau de `pg_dump` sont les tables et les schémas. L'option `-n` permet de sauvegarder seulement le schéma cité après alors que l'option `-N` permet de sauvegarder tous les schémas sauf celui cité après l'option. Le même système existe pour les tables avec les options `-t` et `-T`. Il est possible de mettre ces options plusieurs fois pour sauvegarder plusieurs tables spécifiques ou plusieurs schémas.

Les équivalents longs de ces options sont : `--schema`, `--exclude-schema`, `--table` et `--exclude-table`.

Notez que les dépendances ne sont pas gérées. Si vous demandez à sauvegarder une vue avec `pg_dump -t unevue`, la sauvegarde ne contiendra **pas** les définitions des objets nécessaires à la construction de cette vue.

Par défaut, si certains objets sont trouvés et d'autres non, `pg_dump` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_dump` n'a pas sauvegardé tous les objets souhaités. En voici un exemple (`t1` existe, `t20` n'existe pas) :

```
$ pg_dump -t t1 -t t20 -f postgres.dump postgres
$ echo $?
0
$ pg_dump -t t1 -t t20 --strict-names -f postgres.dump postgres
pg_dump: no matching tables were found for pattern "t20"
$ echo $?
1
```

7.2.9 pg_dump - Option de parallélisation

- `--jobs <nombre_de_threads>`
- format directory (`-Fd`) uniquement

Par défaut, `pg_dump` n'utilise qu'une seule connexion à la base de données pour sauvegarder la définition des objets et les données. Cependant, une fois que la première étape de récupération de la définition des objets est réalisée, l'étape de sauvegarde des données peut être parallélisée pour profiter des nombreux processeurs disponibles sur un serveur.

Cette option n'est compatible qu'avec le format de sortie `directory` (option `-Fd`).

La parallélisation de requêtes ne s'applique hélas pas aux ordres `COPY` utilisés par `pg_dump` . Mais ce dernier peut en lancer plusieurs simultanément avec l'option `--jobs` (`-j`). Elle permet de préciser le nombre de connexions vers la base de données, et aussi de connexions (Unix) ou *threads* (Windows).

Cela permet d'améliorer considérablement la vitesse de sauvegarde, à condition de pouvoir réellement paralléliser. Par exemple, si une table occupe 15 Go sur une base de données de 20 Go, il y a peu de chance que la parallélisation de `pg_dump` change fondamentalement la durée de sauvegarde. Par contre, elle profitera du partitionnement de cette table.

7.2.10 pg_dump - Options diverses

- `--create` (`-C`) : recréer la base
  - y compris paramétrage utilisateur sur base (>v11 et format `plain`)
  - inutile dans les autres formats
- `--no-owner` : ignorer le propriétaire
- `--no-privileges` : ignorer les droits
- `--no-tablespaces` : ignorer les tablespaces
- `--inserts` : remplacer `COPY` par `INSERT`
- `--rows-per-insert`, `--on-conflict-do-nothing`
- divers paramètres pour les tables partitionnées
- `-v` : progression

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

–create :

`--create` (ou `-C`) n'a d'intérêt qu'en format texte (`-Fp`), pour ajouter les instructions de création de base :

```
CREATE DATABASE b1 WITH TEMPLATE = template0
                  ENCODING = 'UTF8' LC_COLLATE = 'C' LC_CTYPE = 'C';
ALTER DATABASE b1 OWNER TO postgres;
```

et éventuellement, s'il y a un paramétrage propre à la base (si le client est en version 11 minimum) :

```
ALTER DATABASE b1 SET work_mem TO '100MB' ;
ALTER ROLE chef IN DATABASE b1 SET work_mem TO '1GB';
```

À partir de la version 11, ces dernières informations sont incluses d'office aux formats `custom` ou `directory`, où `pg_restore --create` peut les retrouver.
Jusqu'en version 10, elles ne se retrouvaient que dans `pg_dumpall` (sans `-g`), ce qui n'était pas pratique quand on ne restaurait qu'une base.
Il faut bien vérifier que votre procédure de restauration reprend ce paramétrage.

Avec `--create`, la restauration se fait en précisant une autre base de connexion, généralement `postgres`. L'outil de restauration basculera automatiquement sur la base nouvellement créée dès que possible.

Masquer des droits et autres propriétés :

`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces` permettent de ne pas récupérer respectivement le propriétaire, les droits, le commentaire et le tablespace de l'objet dans la sauvegarde s'ils posent problème.

Ordres INSERT au lieu de COPY :

Par défaut, `pg_dump` génère des commandes `COPY`, qui sont bien plus rapides que les `INSERT`. Cependant, notamment pour pouvoir restaurer plus facilement la sauvegarde sur un autre moteur de bases de données, il est possible d'utiliser des `INSERT` au lieu des `COPY`. Il faut forcer ce comportement avec l'option `--inserts`.

L'inconvénient des `INSERT` ligne à ligne est leur lenteur par rapport à un `COPY` massif (même avec des astuces comme `synchronous_commit=off`). Par contre, l'inconvénient de `COPY` est qu'en cas d'erreur sur une ligne, tout le `COPY` est annulé. Pour diminuer l'inconvénient des `INSERT` tout en conservant leur intérêt, il est possible d'indiquer le nombre de lignes à intégrer par `INSERT` avec l'option `--rows-per-insert` : si un `INSERT` échoue, seulement ce nombre de lignes sera annulé.

L'option `--on-conflict-do-nothing` permet d'éviter des messages d'erreur si un `INSERT` tente d'insérer une ligne violant une contrainte existante. Très souvent ce sera pour éviter des problèmes de doublons (de même clé primaire) dans une table déjà partiellement chargée, avec les contraintes déjà en place.

Il existe des options pour gérer l'export des tables partitionnées, que nous ne détaillerons pas ici : `--load-via-partition-root`, et depuis la version 16 : `--table-and-children`, `--exclude-table-and-children` et `--exclude-table-data-and-children`.

Enfin, l'option `-v` (ou `--verbose`) permet de voir la progression de la commande.

7.2.11 pg_dumpall

- Sauvegarde d'une instance complète
  - objets globaux (utilisateurs, tablespaces…)
  - toutes les bases de données
- Format texte (SQL) uniquement

`pg_dump` sauvegarde toute la structure et toutes les données locales à une base de données. Cette

commande ne sauvegarde pas la définition des objets globaux, comme par exemple les utilisateurs et les tablespaces.

De plus, il peut être intéressant d'avoir une commande capable de sauvegarder toutes les bases de l'instance. Reconstruire l'instance est beaucoup plus simple car il suffit de rejouer ce seul fichier de sauvegarde.

Contrairement à `pg_dump`, `pg_dumpall` ne dispose que d'un format en sortie : des ordres SQL en texte.

7.2.12 pg_dumpall - Fichier ou sortie standard

- `-f nomfichier.dmp`
  - fichier de la sauvegarde
- ou `-f -`
  - sortie standard

La sauvegarde est automatiquement envoyée sur la sortie standard, sauf si la ligne de commande précise l'option `-f` (ou `--file`) et le nom du fichier.

7.2.13 pg_dumpall - Sélection des objets

- `-g` : tous les objets globaux
- `-r` : uniquement les rôles
- `-t` : uniquement les tablespaces
- `--no-role-passwords` : sans les mots de passe
  - permet de ne pas être superutilisateur

`pg_dumpall` étant créé pour sauvegarder l'instance complète, il disposera de moins d'options de sélection d'objets. Néanmoins, il permet de ne sauvegarder que la déclaration des objets globaux, ou des rôles, ou des tablespaces. Leur versions longues sont respectivement : `--globals-only`, `--roles-only` et `--tablespaces-only`.

Par exemple, voici la commande pour ne sauvegarder que les rôles :

```
$ pg_dumpall -r
--
-- PostgreSQL database cluster dump
--

SET default_transaction_read_only = off;

SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;

--
-- Roles
--

CREATE ROLE admin;
ALTER ROLE admin WITH SUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                 NOREPLICATION NOBYPASSRLS;
CREATE ROLE dupont;
ALTER ROLE dupont WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md5505548e69dafa281a5d676fe0dc7dc43';
CREATE ROLE durant;
ALTER ROLE durant WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md56100ff994522dbc6e493faf0ee1b4f41';
CREATE ROLE martin;
ALTER ROLE martin WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md5d27a5199d9be183ccf9368199e2b1119';
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
                 REPLICATION BYPASSRLS;
CREATE ROLE utilisateur;
ALTER ROLE utilisateur WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                 NOREPLICATION NOBYPASSRLS;

--
-- User Configurations
--

--
-- User Config "u1"
--

ALTER ROLE u1 SET maintenance_work_mem TO '256MB';

--
-- Role memberships
--

GRANT admin TO dupont GRANTED BY postgres;
GRANT admin TO durant GRANTED BY postgres;
GRANT utilisateur TO martin GRANTED BY postgres;
```

```
--
-- PostgreSQL database cluster dump complete
--
```

On remarque que le mot de passe est sauvegardé sous forme de *hash*.

La sauvegarde des rôles se fait en lisant le catalogue système `pg_authid`. Seuls les superutilisateurs ont accès à ce catalogue système car il contient les mots de passe des utilisateurs.

À partir de PostgreSQL 10, on peut permettre d'utiliser `pg_dumpall` sans avoir un rôle superutilisateur, avec l'option `--no-role-passwords`. Celle-ci a pour effet de ne pas sauvegarder les mots de passe. Dans ce cas, `pg_dumpall` va lire le catalogue système `pg_roles` qui est accessible par tout le monde.

7.2.14 pg_dumpall - Exclure une base

- `--exclude-database` (v12+)

Par défaut, `pg_dumpall` sauvegarde les objets globaux et toutes les bases de l'instance. Dans certains cas, il peut être intéressant d'exclure une (ou plusieurs bases). L'option `--exclude-database` a été ajoutée pour cela. Elle n'est disponible qu'à partir de la version 12.

7.2.15 pg_dumpall - Options diverses

- Quelques options partagées avec `pg_dump`
- Très peu utilisées

Il existe d'autres options gérées par `pg_dumpall`. Elles sont déjà expliquées pour la commande `pg_dump` et sont généralement peu utilisées avec `pg_dumpall`.

7.2.16 pg_dump/pg_dumpall - Options de connexions

- `-h` / `$PGHOST` / socket Unix
- `-p` / `$PGPORT` / 5432
- `-U` / `$PGUSER` / utilisateur du système
- `-W` / `$PGPASSWORD`
  - ou `.pgpass`

Les commandes `pg_dump` et `pg_dumpall` se connectent au serveur PostgreSQL comme n'importe quel autre outil (psql, pgAdmin, etc.). Ils disposent donc des options habituelles pour se connecter :

- `-h` ou `--host` pour indiquer l'alias ou l'adresse IP du serveur ;
- `-p` ou `--port` pour préciser le numéro de port ;
- `-U` ou `--username` pour spécifier l'utilisateur ;
- `-W` ne permet pas de saisir le mot de passe en ligne de commande, mais force `pg_dump` à le demander (en interactif donc, et qu'il soit vérifié ou non, ceci dépendant de la méthode d'authentification).

En général, une sauvegarde automatique est effectuée sur le serveur directement par l'utilisateur système `postgres` (connexion par `peer` sans mot de passe), ou à distance avec le mot de passe stocké dans un fichier `.pgpass`.

À partir de la version 10, il est possible d'indiquer plusieurs hôtes et ports. L'hôte sélectionné est le premier qui répond au paquet de démarrage. Si l'authentication ne passe pas, la connexion sera en erreur. Il est aussi possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule.

Par exemple on effectuera une sauvegarde depuis le premier serveur disponible ainsi :

```
pg_dumpall -h secondaire,primaire -p 5432,5433 -U postgres -f sauvegarde.sql
```

Si la connexion nécessite un mot de passe, ce dernier sera réclamé lors de la connexion. Il faut donc faire attention avec `pg_dumpall` qui va se connecter à chaque base de données, une par une. Dans tous les cas, il est préférable d'utiliser un fichier `.pgpass` qui indique les mots de passe de connexion. Ce fichier est créé à la racine du répertoire personnel de l'utilisateur qui exécute la sauvegarde. Il contient les informations suivantes :

```
hote:port:base:utilisateur:mot de passe
```

Ce fichier est sécurisé dans le sens où seul l'utilisateur doit avoir le droit de lire et écrire ce fichier (c'est-à-dire des droits 600). L'outil vérifiera cela avant d'accepter d'utiliser les informations qui s'y trouvent.

7.2.17 Impact des privilèges

- Les outils se comportent comme des clients pour PostgreSQL
- Préférer un rôle superutilisateur
- Sinon :
  - connexion à autoriser
  - le rôle doit pouvoir lire tous les objets à exporter (`pg_read_all_data`)

Même si ce n'est pas obligatoire, il est recommandé d'utiliser un rôle de connexion disposant des droits de superutilisateur pour la sauvegarde et la restauration.

En effet, pour sauvegarder, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf` , être propriétaire ou avoir le privilège `CONNECT` ;
- voir le contenu des différents schémas : être propriétaire ou avoir le privilège `USAGE` sur le schéma ;
- lire le contenu des tables : être propriétaire ou avoir le privilège `SELECT` sur la table.

Pour restaurer, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf` , être propriétaire ou avoir le privilège `CONNECT` ;
- optionnellement, pouvoir créer la base de données cible et pouvoir s'y connecter (option `-C` de `pg_restore`)
- pouvoir créer des schémas : être propriétaire de la base de données ou avoir le privilège `CREATE` sur celle-ci ;
- pouvoir créer des objets dans les schémas : être propriétaire du schéma ou avoir le privilège `CREATE` sur celui-ci ;
- pouvoir écrire dans les tablespaces cibles : être propriétaire du tablespace ou avoir le privilège `CREATE` sur celui-ci ;
- avoir la capacité de donner ou retirer des privilèges : faire partie des rôles bénéficiant d'ACL dans le `dump` .

Le nombre de ces privilèges explique pourquoi il n'est parfois possible de ne restaurer qu'avec un superutilisateur.

7.2.18 Traiter automatiquement la sortie

- Pour compresser 1 fichier : `pg_dump | bzip2`
  - utile avec formats `plain`, `tar`, `custom`
- Outils multi-threads de compression, bien plus rapides :
  - `pbzip2`, `pigz`
  - `xz`, `zstd`...

À part pour le format `directory`, il est possible d'envoyer la sortie standard à un autre outil. Pour les formats non compressés (`tar` et `plain`), cela permet d'abord de compresser avec l'outil de son choix.

Il y a un intérêt même avec le format `custom` : celui d'utiliser des outils de compression plus performants que la zlib, comme `bzip2` ou `lzma` (compression plus forte au prix d'une exécution plus longue) ou `pigz` [2], `pbzip2` [3], ou encore `xz`, `zstd`... beaucoup plus rapides grâce à l'utilisation de plusieurs threads. On met ainsi à profit les nombreux processeurs des machines récentes, au prix d'un très léger surcoût en taille. Ces outils sont présents dans les distributions habituelles.

Au format `custom`, il faut penser à désactiver la compression par défaut, comme dans cet exemple avec `pigz` :

```
$ pg_dump -Fc -Z0 -v foobar | pigz > sauvegarde.dump.gzip
```

On peut aussi utiliser n'importe quel autre outil Unix. Par exemple, pour répartir sur plusieurs fichiers :

```
$ pg_dump | split
```

Nous verrons plus loin que la sortie de `pg_dump` peut même être fournie directement à `pg_restore` ou `psql`, ce qui est fort utile dans certains cas.

[2] https://www.zlib.net/pigz/
[3] http://compression.ca/pbzip2/

7.2.19 Objets binaires

- Deux types dans PostgreSQL : bytea et Large Objects
- Option `-b`
  - uniquement si utilisation des options `-n` / `-N` et/ou `-t` / `-T`
- Option `--no-blobs`
  - pour ne pas sauvegarder les Large Objects
- Option `bytea_output`
  - `escape`
  - `hex`

Il existe deux types d'objets binaires dans PostgreSQL : les Large Objects et les bytea.

Les Larges Objects sont stockées dans une table système appelé `pg_largeobjects`, et non pas dans les tables utilisateurs. Du coup, en cas d'utilisation des options `-n` / `-N` et/ou `-t` / `-T`, la table système contenant les Large Objects sera généralement exclue. Pour être sûr que les Large Objects soient inclus, il faut en plus ajouter l'option `-b`. Cette option ne concerne pas les données binaires stockées dans des colonnes de type bytea, ces dernières étant réellement stockées dans les tables utilisateurs.

Il est possible d'indiquer le format de sortie des données binaires, grâce au paramètre `bytea_output` qui se trouve dans le fichier `postgresql.conf`. Le défaut est `hex`.

Si vous restaurez une sauvegarde d'une base antérieure à la 9.0, notez que la valeur par défaut était différente (`escape`).

Le format `hex` utilise deux octets pour enregistrer un octet binaire de la base alors que le format `escape` utilise un nombre variable d'octets. Dans le cas de données ASCII, ce format n'utilisera qu'un octet. Dans les autres cas, il en utilisera quatre pour afficher textuellement la valeur octale de la donnée (un caractère d'échappement suivi des trois caractères du codage octal). La taille de la sauvegarde s'en ressent, sa durée de création aussi (surtout en activant la compression).

7.2.20 Extensions

- Option `--extension` (`-e`)
  - uniquement si sélection/exclusion (`-n` / `-N` et/ou `-t` / `-T`)

Les extensions sont sauvegardées par défaut.

Cependant, dans le cas où les options `-n` / `-N` (sélection/exclusion de schéma) et/ou `-t` / `-T` (sélection/exclusion de tables) sont utilisées, les extensions ne sont pas sauvegardées. Or, elles pourraient être nécessaires pour les schémas et tables sélectionnées. L'option `-e` permet de forcer la sauvegarde des extensions précisées.

7.3 RESTAURATION D'UNE SAUVEGARDE LOGIQUE

- `psql`
  - restauration de SQL (option `-Fp`) :
- `pg_restore`
  - restauration binaire (`-Ft` / `-Fc` / `-Fd`)

`pg_dump` permet de réaliser deux types de sauvegarde : une sauvegarde texte (via le format `plain`) et une sauvegarde binaire (via les formats tar, personnalisé et répertoire).

Chaque type de sauvegarde aura son outil :

- `psql` pour les sauvegardes textes ;
- `pg_restore` pour les sauvegardes binaires.

7.3.1 psql

- Client standard PostgreSQL
- Capable d'exécuter des requêtes
  - donc de restaurer une sauvegarde texte (*plain*)
- Très limité dans les options de restauration

`psql` est la console interactive de PostgreSQL. Elle permet de se connecter à une base de données et d'y exécuter des requêtes, soit une par une, soit un script complet. Or, la sauvegarde texte de `pg_dump` et de `pg_dumpall` fournit un script SQL. Ce dernier est donc exécutable via `psql`.

7.3.2 psql - Options

- `-f`
  - pour indiquer le fichier contenant la sauvegarde
  - sans `-f` : lit l'entrée standard
- `-1` (`--single-transaction`)
  - pour tout restaurer en une seule transaction
- `-e`
  - pour afficher les ordres SQL exécutés
- `ON_ERROR_ROLLBACK` / `ON_ERROR_STOP`

`psql` étant le client standard de PostgreSQL, le dump au format `plain` se trouve être un script SQL qui peut également contenir des commandes psql, comme `\connect` pour se connecter à une base de données (ce que fait `pg_dumpall` pour changer de base de données).

On bénéficie alors de toutes les options de `psql`, les plus utiles étant celles relatives au contrôle de l'aspect transactionnel de l'exécution.

On peut restaurer avec `psql` de plusieurs manières :

- envoyer le script sur l'entrée standard de psql :

```
cat b1.dump | psql b1
```

- utiliser l'option en ligne de commande -f :

```
psql -f b1.dump b1
```

- utiliser la méta-commande ı :

```
b1 =# \i b1.dump
```

Dans les deux premiers cas, la restauration peut se faire à distance alors que dans le dernier cas, le fichier de la sauvegarde doit se trouver sur le serveur de bases de données.

Le script est exécuté comme tout autre script SQL. Comme il n'y a pas d'instruction `BEGIN` au début, l'échec d'une requête ne va pas empêcher l'exécution de la suite du script, ce qui va généralement apporter un flot d'erreurs. De plus `psql` fonctionne par défaut en autocommit : après une erreur, les requêtes précédentes sont déjà validées. La base de données sera donc dans un état à moitié modifié, ce qui peut poser un problème s'il ne s'agissait pas d'une base vierge.

Il est donc souvent conseillé d'utiliser l'option en ligne de commande `-1` pour que le script complet soit exécuté dans une seule transaction. Dans ce cas, si une requête échoue, aucune modification n'aura réellement lieu sur la base, et il sera possible de relancer la restauration après correction du problème.

Enfin, il est à noter qu'une restauration partielle de la sauvegarde est assez complexe à faire. Deux solutions existent, parfois pénibles :

- modifier le script SQL dans un éditeur de texte, ce qui peut être impossible si ce fichier est suffisamment gros ;
- utiliser des outils tels que `grep` et/ou `sed` pour extraire les portions voulues, ce qui peut facilement devenir long et complexe.

Deux variables psql peuvent être modifiées, ce qui permet d'affiner le comportement de psql lors de l'exécution du script :

ON_ERROR_ROLLBACK :

Par défaut il est à `off`, et toute erreur dans **une transaction** entraîne le `ROLLBACK` de toute la transaction. Les commandes suivantes échouent toutes. Activer `ON_ERROR_ROLLBACK` permet de n'annuler que la commande en erreur. psql effectue des *savepoints* avant chaque ordre, et y retourne en cas d'erreur, avant de continuer le script, toujours dans la même transaction.

Cette option n'est donc **pas** destinée à tout arrêter en cas de problème, au contraire. Mais elle peut être utile pour passer outre à une erreur quand on utilise `-1` pour enrober le script dans une transaction.

`ON_ERROR_ROLLBACK` peut valoir `interactive` (ne s'arrêter dans le script qu'en mode interactif, c'est-à-dire quand c'est une commande `\i` qui est lancée) ou `on` dans quel cas il est actif en permanence.

ON_ERROR_STOP :

Par défaut, dans **un script**, une erreur n'arrête pas le déroulement du script. On se retrouve donc souvent avec un ordre en erreur, et beaucoup de mal pour le retrouver, puisqu'il est noyé dans la masse des messages. Quand `ON_ERROR_STOP` est positionné à `on`, le script est interrompu dès qu'une erreur est détectée.

C'est l'option à privilégier quand on veut arrêter un script au moindre problème. Si `-1` est utilisé, et que `ON_ERROR_ROLLBACK` est resté à `off`, le script entier est bien sûr annulé, et on évite les nombreux messages de type :

```
ERROR: current transaction is aborted,
commands ignored until end of transaction block
```

après la première requête en erreur.

Les variables psql peuvent être modifiées :

- par édition du `.psqlrc` (à déconseiller, cela va modifier le comportement de psql pour toute personne utilisant le compte) :

```
cat .psqlrc
\set ON_ERROR_ROLLBACK interactive
```

- en option de ligne de commande de psql :

```
psql --set=ON_ERROR_ROLLBACK='on'
psql -v ON_ERROR_ROLLBACK='on'
```

- de façon interactive dans psql:

```
psql>\set ON_ERROR_ROLLBACK on
```

7.3.3 pg_restore

- Restaure uniquement les sauvegardes au format binaire
  - format autodétecté (`-F` inutile)
- Nombreuses options très intéressantes
- Restaure une base de données
  - complètement ou partiellement

`pg_restore` est un outil capable de restaurer les sauvegardes au format binaire, quel qu'en soit le format. Il offre de nombreuses options très intéressantes, la plus essentielle étant de permettre une restauration partielle de façon aisée.

L'exemple typique d'utilisation de `pg_restore` est le suivant :

```
pg_restore -d b1 b1.dump
```

La base de données où la sauvegarde va être restaurée est indiquée avec l'option `-d` et le nom du fichier de sauvegarde est le dernier argument dans la ligne de commande.

7.3.4 pg_restore - Base de données

- `-d` : base de données de connexion
- `-C` (`--create`) :
  - connexion (`-d`) et `CREATE DATABASE`
  - connexion à la nouvelle base et exécute le SQL

Avec `pg_restore`, il est indispensable de fournir le nom de la base de données de connexion avec l'option `-d`.

Le fichier à restaurer s'indique en dernier argument sur la ligne de commande.

L'option `--create` (ou `C`) permet de créer la base de données cible. Dans ce cas l'option `-d` doit indiquer une base de données existante afin que `pg_restore` se connecte pour exécuter l'ordre `CREATE DATABASE`. Après cela, il se connecte à la base nouvellement créée pour exécuter les ordres SQL de restauration. Pour vérifier cela, on peut lancer la commande sans l'option `-d`. En observant le code SQL renvoyé on remarque un `\connect` :

```
$ pg_restore -C b1.dump

--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: b1; Type: DATABASE; Schema: -; Owner: postgres
--

CREATE DATABASE b1 WITH TEMPLATE = template0 ENCODING = 'UTF8'
                        LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';


ALTER DATABASE b1 OWNER TO postgres;

\connect b1

SET statement_timeout = 0;
-- Suite du dump…
```

Rappelons que cette option ne récupère la configuration spécifique de la base (paramétrage et droits) que pour une sauvegarde effectuée par un outil client à partir de la version 11.

Il n'est par contre pas possible de restaurer dans une base de données ayant un nom différent de la base de données d'origine avec l'option `-C`.

7.3.5 pg_restore - Fichiers en entrée / sortie

- Entrée : Fichier à restaurer en dernier argument de la ligne de commande
- Sortie :
  - `-f` : fichier SQL ou liste (`-l`)
  - sortie standard : défaut si < v12, sinon `-f -`
- Attention à ne pas écraser la sauvegarde !

L'option `-f` envoie le SQL généré dans un script, qui sera donc du SQL parfaitement lisible :

```
$ pg_restore base.dump -f script_restauration.sql
```

`-f` n'indique **pas** le fichier de sauvegarde, mais bien la sortie de `pg_restore` quand on ne restaure pas vers une base. N'écrasez pas votre sauvegarde !

Avec `-f -`, le SQL transmis au serveur est affiché sur la sortie standard, ce qui est très pratique pour voir ce qui va être restauré, et par exemple valider les options d'une restauration partielle, récupérer des définitions d'index ou de table, voire « piper » le contenu vers un autre outil.

`pg_restore` exige soit `-d`, soit `-f` : soit on restaure dans une base, soit on génère un fichier SQL. (Avant la version 12, `-f` était le le défaut si ni `-d` ni `-f` n'étaient précisés ; il était alors conseillé de rediriger la sortie standard plutôt que d'utiliser `-f` pour éviter toute ambiguïté.)

Pour obtenir le journal d'activité complet d'une restauration, il suffit classiquement de rediriger la sortie :

```
$ pg_restore -d cible --verbose base.dump > restauration.log 2>&1
```

7.3.6 pg_restore - Structure ou données ?

- `--schema-only` : uniquement la structure
- `--data-only` : uniquement les données

ou :

- `--section`
  - `pre-data`
  - `data`
  - `post-data`

Comme pour `pg_dump`, il est possible de ne restaurer que la structure, ou que les données.

Il est possible de restaurer une base section par section. En fait, un fichier de sauvegarde complet est composé de trois sections : la définition des objets, les données, la définition des contraintes et index. Il est plus intéressant de restaurer par section que de restaurer schéma et données séparément car cela permet d'avoir la partie contrainte et index dans un script à part tout à la fin, ce qui accélère la restauration.

Dans les cas un peu délicats (modification des fichiers, imports partiels), on peut vouloir traiter séparément chaque étape. Par exemple, si l'on veut modifier le SQL (modifier des noms de champs, renommer des index…) tout en tenant à compresser ou paralléliser la sauvegarde pour des raisons de volume :

```
$ mkdir data.dump
$ pg_dump -d source --section=pre-data -f predata.sql
$ pg_dump -d source --section=data -Fd --jobs=8 -f data.dump
$ pg_dump -d source --section=post-data -f postdata.sql
```

Après modification, on réimporte :

```
$ psql -d cible < predata.sql
$ pg_restore -d cible --jobs=8  data.dump
$ psql -d cible < postdata.sql
```

Le script issu de `--section=pre-data` (ci-dessous, allégé des commentaires) contient les `CREATE TABLE`, les contraintes de colonne, les attributions de droits mais aussi les fonctions, les extensions, etc. :

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
```

```
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

CREATE EXTENSION IF NOT EXISTS plperl WITH SCHEMA pg_catalog;
COMMENT ON EXTENSION plperl IS 'PL/Perl procedural language';

CREATE EXTENSION IF NOT EXISTS pg_stat_statements WITH SCHEMA public;
COMMENT ON EXTENSION pg_stat_statements
 IS 'track execution statistics of all SQL statements executed';

CREATE FUNCTION public.impair(i integer) RETURNS boolean
    LANGUAGE sql IMMUTABLE
    AS $$
select mod(i,2)=1 ;
$$;

ALTER FUNCTION public.impair(i integer) OWNER TO postgres;

SET default_tablespace = '';
SET default_table_access_method = heap;

CREATE TABLE public.fils (
    i integer,
    CONSTRAINT impair_ck CHECK ((public.impair(i) IS TRUE)),
    CONSTRAINT nonzero_ck CHECK ((i > 0))
);
ALTER TABLE public.fils OWNER TO postgres;

CREATE TABLE public.pere (
    i integer NOT NULL
);

ALTER TABLE public.pere OWNER TO postgres;
```

La partie `--section=data`, compressée ou non, ne contient que des ordres `COPY` :

```
# lecture du fichier data.dump sur la sortie standard (-)
$ pg_restore -f - data.dump

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

COPY public.fils (i) FROM stdin;
1
```

```
2
…
\.
COPY public.pere (i) FROM stdin;
1
2
…
\.
```

Quant au résultat de `--section=pre-data`, il regroupe notamment les contraintes de clés primaire, de clés étrangères, et les créations d'index. Il est nettement plus rapide de charger la table avant de poser contraintes et index que l'inverse.

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
SET default_tablespace = '';

ALTER TABLE ONLY public.pere
    ADD CONSTRAINT pere_pkey PRIMARY KEY (i);

CREATE UNIQUE INDEX fils_i_idx ON public.fils USING btree (i);

ALTER TABLE ONLY public.fils
    ADD CONSTRAINT fk FOREIGN KEY (i) REFERENCES public.pere(i);
```

7.3.7 pg_restore - Sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf ce schéma
- `-t <table>` : cette relation
- `-T <trigger>` : ce trigger
- `-I <index>` : cet index
- `-P <fonction>` : cette fonction
- En option
  - possibilité d'en mettre plusieurs
  - `--strict-names`, pour avoir une erreur si l'objet est inconnu

`pg_restore` fournit quelques options supplémentaires pour sélectionner les objets à restaurer. Il y a les options `-n` et `-t` qui ont la même signification que pour `pg_dump`. `-N` n'existe que depuis la version 10 et a la même signification que pour `pg_dump`. Par contre, `-T` a une signification différente : `-T` précise un trigger dans `pg_restore`.

Il existe en plus les options `-I` et `-P` (respectivement `--index` et `--function`) pour restaurer respectivement un index et une routine stockée spécifique.

Là aussi, il est possible de mettre plusieurs fois les options pour restaurer plusieurs objets de même type ou de type différent.

Par défaut, si le nom de l'objet est inconnu, `pg_restore` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_restore` n'a pas restauré l'objet souhaité. En voici un exemple :

```
$ pg_restore -t t2 -d postgres pouet.dump
$ echo $?
0
$ pg_restore -t t2 --strict-names -d postgres pouet.dump
pg_restore: [archiver] table "t2" not found
$ echo $?
1
```

7.3.8 pg_restore - Sélection avancée

- `-l` : récupération de la liste des objets
- `-L <liste_objets>` : restauration uniquement des objets listés dans ce fichier

Les options précédentes sont intéressantes quand on a peu de sélection à faire. Par exemple, cela convient quand on veut restaurer deux tables ou quatre index. Quand il faut en restaurer beaucoup plus, cela devient plus difficile. `pg_restore` fournit un moyen avancé pour sélectionner les objets.

L'option `-l` (`--list`) permet de connaître la liste des actions que réalisera `pg_restore` avec un fichier particulier. Par exemple :

```
$ pg_restore -l b1.dump
;
; Archive created at 2020-09-16 15:44:35 CET
;     dbname: b1
;     TOC Entries: 15
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
```

```
;     Dumped from database version: 13.0
;     Dumped by pg_dump version: 13.0
;
;
; Selected TOC Entries:
;
200; 1255 24625 FUNCTION public f1() postgres
201; 1255 24626 PROCEDURE public p1() postgres
197; 1259 24630 TABLE public t2 postgres
199; 1259 24637 MATERIALIZED VIEW public mv1 postgres
196; 1259 24627 TABLE public t1 postgres
198; 1259 24633 VIEW public v1 postgres
3902; 0 24627 TABLE DATA public t1 postgres
3903; 0 24630 TABLE DATA public t2 postgres
3778; 2606 24642 CONSTRAINT public t2 t2_pkey postgres
3776; 1259 24643 INDEX public t1_c1_idx postgres
3904; 0 24637 MATERIALIZED VIEW DATA public mv1 postgres
```

Toutes les lignes qui commencent avec un point-virgule sont des commentaires. Le reste indique les objets à créer : la fonction `f1`, la procédure stockée `p1`, les tables `t1` et `t2`, la vue `v1`, la clé primaire sur `t2` et l'index sur `t1`. Il indique aussi les données à restaurer avec des lignes du type `TABLE DATA`. Donc, dans cette sauvegarde, il y a les données pour les tables `t1` et `t2`. Enfin, il y a le rafraîchissement des données de la vue matérialisée `mv1`.

Il est possible de stocker cette information dans un fichier, de modifier le fichier pour qu'il ne contienne que les objets que l'on souhaite restaurer, et de demander à `pg_restore`, avec l'option `-L` (`--use-list`), de ne prendre en compte que les actions contenues dans le fichier. Voici un exemple complet :

```
$ pg_restore -l b1.dump > liste_actions

$ cat liste_actions | \
  grep -v "f1" | \
  grep -v "TABLE DATA public t2" | \
  grep -v "INDEX public t1_c1_idx" \
  > liste_actions_modifiee

$ createdb b1_new

$ pg_restore -L liste_actions_modifiee -d b1_new -v b1.dump
pg_restore: connecting to database for restore
pg_restore: creating PROCEDURE "public.p1()"
pg_restore: creating TABLE "public.t2"
pg_restore: creating MATERIALIZED VIEW "public.mv1"
pg_restore: creating TABLE "public.t1"
pg_restore: creating VIEW "public.v1"
pg_restore: processing data for table "public.t1"
pg_restore: creating CONSTRAINT "public.t2 t2_pkey"
pg_restore: creating MATERIALIZED VIEW DATA "public.mv1"
```

L'option `-v` de `pg_restore` permet de visualiser sa progression dans la restauration. On remarque bien que la fonction `f1` ne fait pas partie des objets restaurés, tout comme l'index sur `t1` et les données de la table `t2`.

Enfin, il est à la charge de l'utilisateur de fournir une liste cohérente en terme de dépendances. Par exemple, sélectionner seulement l'entrée `TABLE DATA` alors que la table n'existe pas dans la base de données cible provoquera une erreur.

7.3.9 pg_restore - Option de parallélisation

- `-j <nombre_de_threads>`
  - formats `custom` ou `directory`

Historiquement, `pg_restore` n'utilise qu'une seule connexion à la base de données pour y exécuter en série toutes les requêtes nécessaires pour restaurer la base. Cependant, une fois que la première étape de création des objets est réalisée, l'étape de copie des données et celle de création des index peuvent être parallélisées pour profiter des nombreux processeurs disponibles sur un serveur. L'option `-j` permet de préciser le nombre de connexions réalisées vers la base de données. Chaque connexion est gérée dans `pg_restore` par un processus sous Unix et par un thread sous Windows.

Cela permet d'améliorer considérablement la vitesse de restauration, en partie parce que les index peuvent être calculés en parallèle. Un test effectué a montré qu'une restauration d'une base de 150 Go prenait 5 h avec une seule connexion, mais seulement 3 h avec plusieurs connexions.

Il est possible qu'il n'y ait pas de gain. Par exemple, si une table occupe 15 Go sur une sauvegarde de 20 Go, la parallélisation ne changera pas fondamentalement la durée de restauration, car la table ne sera importée que par une seule connexion.

Le format *plain* (texte) n'est pas compatible avec cette option.

Il est à noter que, même si PostgreSQL supporte la parallélisation de certains types de requêtes, cela ne concerne pas la commande `COPY` de `pg_restore`.

7.3.10 pg_restore - Options diverses

- `-O` (`--no-owner`) : ignorer le propriétaire
- `-x` (`--no-privileges`) : ignorer les droits
- `--no-comments` : ignorer les commentaires
- `--no-tablespaces` : ignorer le tablespace
- `-1` (`--single-transaction`) : pour tout restaurer en une seule transaction
- `-c` (`--clean`) : pour détruire un objet avant de le restaurer

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

Les quatre suivantes (`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces`) permettent de ne pas restaurer respectivement le propriétaire, les droits, le commentaire et le tablespace des objets.

L'option `-1` permet d'exécuter `pg_restore` dans une seule transaction. Attention, ce mode est incompatible avec le mode `-j` car on ne peut pas avoir plusieurs sessions qui partagent la même transaction.

L'option `-c` permet d'exécuter des `DROP` des objets avant de les restaurer. Ce qui évite les conflits à la restauration de tables par exemple : l'ancienne est détruite avant de restaurer la nouvelle.

L'option `--disable-triggers` est très dangereuse mais peut servir dans certaines situations graves : elle inhibe la vérification des contraintes lors d'un import et peut donc mener à une base incohérente !

Enfin, l'option `-v` permet de voir la progression de la commande.

7.4 AUTRES CONSIDÉRATIONS SUR LA SAUVEGARDE LOGIQUE

- Versions des outils & version du serveur
- Script de sauvegarde
- Sauvegarder sans passer par un fichier
- Statistiques et maintenance après import
- Durée d'exécution d'une sauvegarde
- Taille d'une sauvegarde

La sauvegarde logique est très simple à mettre en place. Mais certaines considérations sont à prendre en compte lors du choix de cette solution : comment gérer les statistiques, quelle est la durée d'exécution d'une sauvegarde, quelle est la taille d'une sauvegarde, etc.

7.4.1 Versions des outils clients et version de l'instance

- `pg_dump` : reconnaît les versions de PG antérieures
- `pg_restore`
  - minimum la version du `pg_dump` utilisé
  - si possible celle du serveur cible
- Pas de problème entre OS différents

Il est fréquent de générer une sauvegarde sur une version de PostgreSQL pour l'importer sur un serveur de version différente, en général plus récente. Une différence de version mineure n'est d'habitude pas un problème, ce sont les versions majeures (9.6, 10, 11…) qui importent.

Il faut bien distinguer les versions des instances source et cible, et les versions des outils `pg_dump` et `pg_restore`. Il peut y avoir plusieurs de ces dernières sur un poste. La version sur le poste où l'on sauvegarde peut différer de la version du serveur. Or, leurs formats de sauvegarde `custom` ou `directory` diffèrent.

`pg_dump` sait sauvegarder depuis une instance de version antérieure à la sienne (du moins à partir de PostgreSQL 8.0). Il refusera de tenter une sauvegarde d'une instance de version postérieure. Donc, pour sauvegarder une base PostgreSQL 12, utilisez un `pg_dump` de version 12, 13 ou supérieure.

`pg_restore` sait lire les sauvegardes des versions antérieures à la sienne. Il peut restaurer vers une instance de version supérieure à la sienne, même s'il vaut mieux utiliser le `pg_restore` de la même version que l'instance. La restauration vers une version antérieure a de bonnes chances d'échouer sur une évolution de la syntaxe.

Bien sûr, le format `plain` (SQL pur) est toujours lisible par `psql`, et `pg_restore -f dump.sql` permet toujours d'en regénérer un depuis une sauvegarde `plain` ou `directory`. Il peut même être envoyé directement sans fichier intermédiaire, par exemple ainsi :

```
pg_restore -f dump.sql | psql -h serveurcible -d basecible
```

S'il y a une nouveauté ou une régression que l'instance cible ne sait pas interpréter, il est possible de modifier ce SQL. Dans beaucoup de cas, il suffira d'adapter le SQL dans les parties générées par `--section=pre-data` et `--section=post-data`, et de charger directement les données avec `pg_restore --section=data`.

Enfin, puisqu'il s'agit de sauvegardes logiques, des différences de système d'exploitation ne devraient pas poser de problème de compatibilité supplémentaire.

7.4.2 Script de sauvegarde idéal

- Objets globaux :

```
pg_dumpall -g
```

- Chaque base :

```
pg_dump -Fc
pg_dump -Fd
```

- Outils client >= v11
  - sinon reprendre les paramètres des rôles sur les bases :

```
ALTER role xxx IN DATABASE xxx SET param=valeur;
```

- Bien tester !

`pg_dumpall` n'est intéressant que pour récupérer les objets globaux. Le fait qu'il ne supporte pas les formats binaires entraîne que sa sauvegarde n'est utilisable que dans un cas : la restauration de toute une instance. C'est donc une sauvegarde très spécialisée, ce qui ne conviendra pas à la majorité des cas.

Le mieux est donc d'utiliser `pg_dumpall` avec l'option `-g`, puis d'utiliser `pg_dump` pour sauvegarder chaque base dans un format binaire.

Attention :

- Avant la version 11, les paramètres sur les bases (`ALTER DATABASE xxx SET param=valeur;`) ne seront pas du tout dans les sauvegardes : `pg_dumpall -g` n'exporte pas les définitions des bases (voir `pg_dump --create` plus haut).
- Les paramètres sur les rôles dans les bases figureront dans l'export de `pg_dumpall -g` ainsi :

```
ALTER role xxx IN DATABASE xxx SET param=valeur;
```

mais les bases n'existeront pas forcément au moment où l'`ALTER ROLE` sera exécuté ! Il faudra donc penser à les restaurer à la main…

Voici un exemple de script minimaliste :

```
#!/bin/sh
# Script de sauvegarde pour PostgreSQL

REQ="SELECT datname FROM pg_database WHERE datallowconn ORDER BY datname"

pg_dumpall -g > globals.dump
psql -XAtc "$REQ" postgres | while read base
do
  pg_dump -Fc $base > ${base}.dump
done
```

Évidemment, il ne conviendra pas à tous les cas, mais donne une idée de ce qu'il est possible de faire. (Voir plus bas `pg_back` pour un outil plus complet.)

Exemple de script de sauvegarde adapté pour un serveur Windows :

```
@echo off

SET PGPASSWORD=super_password
SET PATH=%PATH%;C:\Progra~1\PostgreSQL\11\bin\

pg_dumpall -g -U postgres postgres > c:\pg-globals.sql

for /F %%v in ('psql -XAt -U postgres -d cave
             -c "SELECT datname FROM pg_database WHERE NOT datistemplate"') do (
    echo "dump %%v"
    pg_dump -U postgres -Fc %%v > c:\pg-%%v.dump
)
pause
```

Autre exemple plus complet de script de sauvegarde totale de toutes les bases, avec une période de rétention :

```
#!/bin/sh
#------------------------------------------------------------------
#
# Script used to perform a full backup of all databases from a
# PostgreSQL Cluster. The pg_dump use the custom format is done
# into one file per database. There's also a backup of all global
# objects using pg_dumpall -g.
#
# Backup are preserved following the given retention days (default
# to 7 days).
#
# This script should be run daily as a postgres user cron job:
#
# 0 23 * * * /path/to/pg_fullbackup.sh >/tmp/fullbackup.log 2>&1
#
#------------------------------------------------------------------

# Backup user who must run this script, most of the time it should be postgres
BKUPUSER=postgres
# Number of days you want to preserve your backup
RETENTION_DAYS=7
# Where the backup files should be saved
#BKUPDIR=/var/lib/pgsql/backup
BKUPDIR=/var/lib/postgresql/backup_bases
# Prefix used to prefix the name of all backup files
PREFIX=bkup
# Optional hostname of remote server
# Leave empty to use unix socket
HOSTNAME=""
# Optional user to log in to the remote server
USERNAME=""

WHO=`whoami`
if [ "${WHO}" != "${BKUPUSER}" ]; then
        echo "FATAL: you must run this script as ${BKUPUSER} user."
        exit 1;
fi

# Testing backup directory
if [ ! -e "${BKUPDIR}" ]; then
        mkdir -p "${BKUPDIR}"
fi
echo "Begining backup at "`date`

# Set the query to list the available databases
REQ="SELECT datname FROM pg_database WHERE datistemplate = 'f'
     AND datallowconn ORDER BY datname"

# Set the date variable to be used in the backup file names
DATE=$(date +%Y-%m-%d_%Hh%M)

# Define the additional pg program options
PG_OPTION=""
if [ $HOSTNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -h $HOSTNAME"
fi;
```

```
if [ $USERNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -U $USERNAME"
fi;

# Dump PostgreSQL Cluster global objects
echo "Dumping global object into ${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
pg_dumpall ${PG_OPTION} -g > "${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
if [ $? -ne 0 ]; then
        echo "FATAL: Can't dump global objects with pg_dumpall."
        exit 2;
fi

# Extract the list of database
psql ${PG_OPTION} -Atc "$REQ" postgres | while read base

# Dump content of all databases
do
  echo "Dumping database $base into ${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  pg_dump ${PG_OPTION} -Fc $base > "${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  if [ $? -ne 0 ]; then
          echo "FATAL: Can't dump database ${base} with pg_dump."
          exit 3;
  fi
done
if [ $? -ne 0 ]; then
        echo "FATAL: Can't list databases with psql query."
        exit 4;
fi

# Starting deletion of obsolete backup files
if [ ${RETENTION_DAYS} -gt 0 ]; then
  echo "Removing obsolete backup files older than ${RETENTION_DAYS} day(s)."
  find ${BKUPDIR}/ -maxdepth 1 -name "${PREFIX}_*" -mtime ${RETENTION_DAYS} \
                   -exec rm -rf '{}' ';'
fi

echo "Backup ending at "`date`

exit 0
```

7.4.3 pg_back - Présentation

- https://github.com/orgrim/pg_back
- Type de sauvegardes : **logiques** (`pg_dump`)
- Langage : **bash** (v1) / **go** (v2)
- Licence : **BSD** (libre)
- Type de stockage : **local** + export cloud
- Planification : **crontab**
- **Unix/Linux** (v1 & 2) / **Windows (v2)**
- Compression : via pg_dump
- Versions compatibles : **toutes**
- Rétention : **durée**

pg_back[4] a été écrit par Nicolas Thauvin, consultant de Dalibo, également auteur original de pitrery[5].

Ce programme assez complet vise à gérer le plus simplement possible des sauvegardes logiques (`pg_dump` , `pg_dumpall`), y compris au niveau de la rétention.

La version 1 est en bash, directement utilisable et éprouvée, mais ne sera plus maintenue à terme.

La version 2, parue en 2021, a été réécrite en go. Le binaire est directement utilisable, et permet notamment une configuration différente par base, une meilleure gestion des paramètres de connexion à PostgreSQL et le support de Windows. Pour les versions avant la 11, le script pg_dumpacl[6] est intégré (v2) ou supporté (v1) pour sauvegarder le paramétrage au niveau des bases.

Vous pouvez les obtenir sur le site du projet[7], tout comme le source[8].

Les sauvegardes sont aux formats gérés par `pg_dump` : SQL, custom, par répertoire, compressées ou non…

Les anciennes sauvegardes sont automatiquement purgées, et l'on peut en conserver un nombre minimum. La version 2.1 permet en plus de chiffrer les sauvegardes avec une phrase de passe, de générer des sommes de contrôle, et d'exporter vers Azure, Google Cloud ou Amazon S3, ou n'importe quel serveur distant accessible avec `ssh` en SFTP.

L'outil ne propose pas d'options pour restaurer les données : il faut utiliser ceux de PostgreSQL (`pg_restore` , `psql`).

[4] https://github.com/orgrim/pg_back
[5] http://dalibo.github.io/pitrery/
[6] https://github.com/dalibo/pg_dumpacl
[7] https://github.com/orgrim/pg_back/tags
[8] https://github.com/orgrim/pg_back/

7.4.4 Sauvegarde et restauration sans fichier intermédiaire

- `pg_dump -Fp | psql`
- `pg_dump -Ft | pg_restore`
- `pg_dump -Fc | pg_restore`
- Utilisation des options `-h`, `-p`, `-d`
- Attention à la gestion des erreurs !

La duplication d'une base ne demande pas forcément de passer par un fichier intermédiaire. Il est possible de fournir la sortie de `pg_dump` (format `plain` implicite) à `psql` ou `pg_restore`. Par exemple :

```
$ createdb nouvelleb1
$ pg_dump -Fp b1 | psql nouvelleb1
```

Ces deux commandes permettent de dupliquer `b1` dans `nouvelleb1`.

L'utilisation des options `-h`, `-p`, `-d` permet de sauvegarder et restaurer une base sur des instances différentes, qu'elles soient locales ou à distance.

Le gain de temps est appréciable : l'import peut commencer avant la fin de l'export. Comme toujours avec `pg_dump`, les données restaurées correspondent à celles qui existaient au début de la sauvegarde. On épargne aussi la place nécessaire au stockage du backup. Par contre on ne peut pas paralléliser l'export.

Avec `-Fp`, le flux circule en pur texte : il est possible d'intercaler des appels à des outils comme `awk` ou `sed` pour effectuer certaines opérations à la volée (renommage…).

Cette version peut être intéressante :

```
$ pg_dump -Fc  | pg_restore
```

car elle permet de profiter des options propres au format `custom` à commencer par la compression. Il n'y a alors pas besoin d'intercaler des commandes comme `gzip` / `gunzip` pour alléger la charge réseau.

Le point délicat est la gestion des erreurs puisqu'il y a deux processus à surveiller : par exemple dans un script `bash`, on testera le contenu de `${PIPESTATUS[@]}` (et pas seulement `$?`) pour vérifier que tout s'est bien déroulé, et l'on ajoutera éventuellement `set -o pipefail` en début de script.

7.4.5 Statistiques et maintenance après import

- Statistiques non sauvegardées
  - `ANALYZE` impérativement après une restauration !
- Pour les performances :
  - `VACUUM` (ou `VACUUM ANALYZE`)
- À plus long terme :
  - `VACUUM FREEZE`

Les statistiques sur les données, utilisées par le planificateur pour sélectionner le meilleur plan d'exécution possible, ne font pas partie de la sauvegarde. Il faut donc exécuter un `ANALYZE` après avoir restauré une sauvegarde. Sans cela, les premières requêtes pourraient s'exécuter très mal du fait de statistiques non à jour.

Lancer un `VACUUM` sur toutes les tables restaurées est également conseillé. S'il n'y a pas besoin de les défragmenter, certaines opérations de maintenance effectuées par un `VACUUM` ont un impact sur les performances. En premier lieu, les *hint bits* (« bits d'indice ») de chaque enregistrement seront mis à jour au plus tard à la relecture suivante, et généreront de nombreuses écritures. Un `VACUUM` explicite forcera ces écritures, si possible avant la mise en production de la base. Il créera aussi la *visibility map* des tables, ce qui autorisera les *Index Only Scans*, une optimisation extrêmement puissante, sans laquelle certaines requêtes seront beaucoup plus lentes.

L'`autovacuum` se chargera bien sûr progressivement de tout cela. Il est cependant par défaut bridé pour ne pas gêner les opérations, et pas toujours assez réactif. Jusqu'en PostgreSQL 12 compris, les insertions ne provoquent que l'`ANALYZE`, pas le `VACUUM`, qui peut donc tarder sur les tables un peu statiques.

Il est donc préférable de lancer un `VACUUM ANALYZE` manuel à la fin de la restauration, afin de procéder immédiatement au passage des statistiques et aux opérations de maintenance.

Il est possible de séparer les deux étapes. L'`ANALYZE` est impératif et rapide ; le `VACUUM` est beaucoup plus lent mais peut avoir lieu durant la production si le temps presse.

À plus long terme, dans le cas d'un gros import ou d'une restauration de base, existe un autre danger : le `VACUUM FREEZE`. Les numéros de transaction étant cycliques, l'autovacuum les « nettoie » des

tables quand il sont assez vieux. Les lignes ayant été importées en même temps, cela peut générer l'écriture de gros volumes de manière assez soudaine. Pour prévenir cela, lancez un `VACUUM FREEZE` dans une période calme quelques temps après l'import.

7.4.6 Durée d'exécution

- Difficile à chiffrer
- Dépend de l'activité sur le serveur
- Option `-v`
- Suivre les `COPY`
  - vue `pg_stat_progress_copy` (v14+)

La durée d'exécution d'une sauvegarde et d'une restauration est difficile à estimer. Cela dépend beaucoup de l'activité présente sur le serveur, de la volumétrie, du matériel, etc.

L'option `-v` de `pg_dump` et de `pg_restore` permet de suivre les opérations, action par action (donc la création des objets, mais aussi leur remplissage). Cependant, sans connaître la base sauvegardée ou restaurée, il est difficile de prédire le temps restant pour la fin de l'opération. Nous savons seulement qu'il a fini de traiter tel objet.

Depuis la version 14, il est possible de suivre individuellement les opérations de copie des données, si ces dernières passent par l'instruction `COPY`. Là encore, il est difficile d'en tirer beaucoup d'informations sans bien connaître la base en cours de traitement. Cependant, cela permet de savoir si une table a bientôt fini d'être traitée par le `COPY` en cours. Pour les tables volumineuses, c'est intéressant.

7.4.7 Taille d'une sauvegarde logique

- Difficile à évaluer
- Contenu des index non sauvegardé
  - donc sauvegarde plus petite
- Objets binaires :
  - entre 2 et 4 fois plus gros
  - donc sauvegarde plus grosse

Il est très difficile de corréler la taille d'une base avec celle de sa sauvegarde.

Le contenu des index n'est pas sauvegardé. Donc, sur une base contenant 10 Go de tables et 10 Go d'index, avoir une sauvegarde de 10 Go ne serait pas étonnant. Le contenu des index est généré lors de la restauration.

Par contre, les données des tables prennent généralement plus de place. Un objet binaire est stocké sous la forme d'un texte, soit de l'octal (donc 4 octets), soit de l'hexadécimal (2 octets). Donc un objet binaire prend 2 à 4 fois plus de place dans la sauvegarde que dans la base. Mais même un entier ne va pas avoir la même occupation disque. La valeur 10 ne prend que 2 octets dans la sauvegarde, alors qu'il en prend quatre dans la base. Et la valeur 1 000 000 prend 7 octets dans la sauvegarde alors qu'il en prend toujours 4 dans la base.

Tout ceci permet de comprendre que la taille d'une sauvegarde n'a pas tellement de lien avec la taille de la base. Il est par contre plus intéressant de comparer la taille de la sauvegarde de la veille avec celle du jour. Tout gros changement peut être annonciateur d'un changement de volumétrie de la base, changement voulu ou non.

7.4.8 Avantages de la sauvegarde logique

- Simple et rapide
- Sans interruption de service
- Indépendante de la version de PostgreSQL
- Granularité de sélection à l'objet
- Taille réduite
- Ne conserve pas la fragmentation des tables et des index
- Éventuellement depuis un serveur secondaire

La sauvegarde logique ne nécessite aucune configuration particulière de l'instance, hormis l'autorisation de la connexion du client effectuant l'opération. La sauvegarde se fait sans interruption de service. Par contre, l'instance doit être disponible, ce qui n'est pas un problème dans la majorité des cas.

Elle est indépendante de la version du serveur PostgreSQL, source et cible. Elle ne contient que des ordres SQL nécessaires à la création des objets à l'identique et permet donc de s'abstraire du format de stockage sur le serveur. De ce fait, la fragmentation des tables et des index disparaît à la restauration.

Une restauration de sauvegarde logique est d'ailleurs la méthode officielle de montée de version majeure. Même s'il existe d'autres méthodes de migration de version majeure, elle reste le moyen le plus sûr parce que le plus éprouvé.

Une sauvegarde logique ne contenant que les données utiles, sa taille est généralement beaucoup plus faible que la base de données source, sans parler de la compression qui peut encore réduire

l'occupation sur disque. Par exemple, seuls les ordres DDL permettant de créer les index sont stockés, pas leur contenu. Ils sont alors créés de zéro à la restauration.

Les outils permettent de sélectionner très finement les objets sur lesquels on travaille, à la sauvegarde comme à la restauration.

Il est assez courant d'effectuer une sauvegarde logique à partir d'un serveur secondaire (réplica de la production), pour ne pas charger les disques du primaire ; quitte à mettre la réplication du secondaire en pause le temps de la sauvegarde.

7.4.9 Inconvénients de la sauvegarde logique

- Durée : dépendante des données et de l'activité
- Restauration : uniquement au démarrage de l'export
- Efficace si < 200 Go
- Plusieurs outils pour sauvegarder une instance complète
- `ANALYZE`, `VACUUM ANALYZE`, `VACUUM FREEZE` après import

L'un des principaux inconvénients d'une sauvegarde et restauration porte sur la durée d'exécution. Elle est proportionnelle à la taille de la base de données, ou à la taille des objets choisis pour une sauvegarde partielle.

En conséquence, il est généralement nécessaire de réduire le niveau de compression pour les formats `custom` et `directory` afin de gagner du temps. Avec des disques mécaniques en RAID 10, il est généralement nécessaire d'utiliser d'autres méthodes de sauvegarde lorsque la volumétrie dépasse 200 Go.

Le second inconvénient majeur de la sauvegarde logique est l'absence de granularité temporelle. Une « photo » des données est prise au démarrage de la sauvegarde, et on ne peut restaurer qu'à cet instant, quelle que soit la durée d'exécution de l'opération. Il faut cependant se rappeler que cela garantit la cohérence du contenu de la sauvegarde d'un point de vue transactionnel.

Comme les objets et leur contenu sont effectivement recréés à partir d'ordres SQL lors de la restauration, on se débarrasse de la fragmentation des tables et index, mais on perd aussi les statistiques de l'optimiseur, ainsi que certaines méta-données des tables. Il est donc nécessaire de lancer ces opérations de maintenance après l'import.

7.5 SAUVEGARDE PHYSIQUE À FROID DES FICHIERS

- Instance arrêtée : sauvegarde cohérente
- Ne pas oublier : journaux, tablespaces, configuration !
- Outils : système, aucun spécifique à PostgreSQL
  - `cp`, `tar`…
  - souvent : `rsync` en 2 étapes : à chaud puis à froid
  - snapshots SAN/LVM (attention à la cohérence)

Toutes les données d'une instance PostgreSQL se trouvent dans des fichiers. Donc sauvegarder les fichiers permet de sauvegarder une instance. Cependant, cela ne peut pas se faire aussi simplement que ça. Lorsque PostgreSQL est en cours d'exécution, il modifie certains fichiers du fait de l'activité des utilisateurs ou des processus (interne ou non) de maintenances diverses. Sauvegarder les fichiers de la base sans plus de manipulation ne peut donc se faire qu'à froid. Il faut donc arrêter PostgreSQL pour disposer d'une sauvegarde cohérente si la sauvegarde se fait au niveau du système de fichiers.

Il est cependant essentiel d'être attentif aux données qui ne se trouvent pas directement dans le répertoire des données (`PGDATA`), notamment :

- le répertoire des journaux de transactions (`pg_wal` ou `pg_xlog`), qui est souvent placé dans un autre système de fichiers pour gagner en performances : sans lui, la sauvegarde ne sera pas utilisable ;
- les répertoires des tablespaces s'il y en a, sinon une partie des données manquera ;
- les fichiers de configuration (sous `/etc` sous Debian, notamment), y compris ceux des outils annexes à PostgreSQL.

Voici un exemple de sauvegarde (Cent OS 7) :

```
$ systemctl stop postgresql-12
$ tar cvfj data.tar.bz2 /var/lib/pgsql/12/data
$ systemctl start postgresql-12
```

Le gros avantage de cette sauvegarde se trouve dans le fait que vous pouvez utiliser tout outil de sauvegarde de fichier : `cp`, `scp`, `tar`, `ftp`, `rsync`, etc. et tout outil de sauvegarde utilisant lui-même ces outils.

Comme la sauvegarde doit être effectuée avec l'instance arrêtée, la durée de l'arrêt est dépendante du volume de données à sauvegarder. On peut optimiser les choses en réduisant le temps d'interruption avec l'utilisation de snapshots au niveau système de fichier ou avec `rsync`.

Pour utiliser des snapshots, il faut soit disposer d'un SAN offrant cette possibilité ou bien utiliser la fonctionnalité équivalente de LVM voire du système de fichier. Dans ce cas, la procédure est la suivante :

- arrêt de l'instance PostgreSQL ;
- création des snapshots de l'ensemble des systèmes de fichiers ;
- démarrage de l'instance ;
- sauvegarde des fichiers à partir des snapshots ;
- destruction des snapshots.

Si on n'a pas la possibilité d'utiliser des snapshots, on peut utiliser `rsync` de cette manière :

- `rsync` de l'ensemble des fichiers de l'instance PostgreSQL en fonctionnement pour obtenir une première copie (incohérente) ;
- arrêt de l'instance PostgreSQL ;
- `rsync` de l'ensemble des fichiers de l'instance pour ne transférer que les différences ;
- redémarrage de l'instance PostgreSQL.

7.5.1 Avantages des sauvegardes à froid

- Simple
- Rapide à la sauvegarde
- Rapide à la restauration
- Beaucoup d'outils disponibles

L'avantage de ce type de sauvegarde est sa rapidité. Cela se voit essentiellement à la restauration où les fichiers ont seulement besoin d'être créés. Les index ne sont pas recalculés par exemple, ce qui est certainement le plus long dans la restauration d'une sauvegarde logique.

7.5.2 Inconvénients des sauvegardes à froid

- Arrêt de la production
- Sauvegarde de l'instance complète (donc aucune granularité)
- Restauration de l'instance complète
- Conservation de la fragmentation
- Impossible de changer d'architecture
    - Réindexation si changement OS

Il existe aussi de nombreux inconvénients à cette méthode.

Le plus important est certainement le fait qu'il faut arrêter la production. L'instance PostgreSQL doit être arrêtée pour que la sauvegarde puisse être effectuée.

Il ne sera pas possible de réaliser une sauvegarde ou une restauration partielle, il n'y a pas de granularité. C'est forcément l'intégralité de l'instance qui sera prise en compte.

Étant donné que les fichiers sont sauvegardés, toute la fragmentation des tables et des index est conservée.

De plus, la structure interne des fichiers implique l'architecture où cette sauvegarde sera restaurée. Donc une telle sauvegarde impose de conserver un serveur 32 bits pour la restauration si la sauvegarde a été effectuée sur un serveur 32 bits. De même, l'architecture *little endian/big endian* doit être respectée.

De plus, des différences entre deux systèmes, et même entre deux versions d'une même distribution, peuvent mener à devoir réindexer toute l'instance[9].

Tous ces inconvénients ne sont pas présents pour la sauvegarde logique. Cependant, cette sauvegarde a aussi ses propres inconvénients, comme une lenteur importante à la restauration.

7.5.3 Diminuer l'immobilisation

- Utilisation de rsync
- Une fois avant l'arrêt
- Une fois après

Il est possible de diminuer l'immobilisation d'une sauvegarde de fichiers en utilisant la commande `rsync`.

`rsync` permet de synchroniser des fichiers entre deux répertoires, en local ou à distance. Il va comparer les fichiers pour ne transférer que ceux qui ont été modifiés. Il est donc possible d'exécuter `rsync` avec PostgreSQL en cours d'exécution pour récupérer un maximum de données, puis d'arrêter PostgreSQL, de relancer `rsync` pour ne récupérer que les données modifiées entre temps, et enfin de relancer PostgreSQL. Notez que l'utilisation des options `--delete` et `--checksum` est *fortement conseillée* lors de la deuxième passe, pour rendre la copie totalement fiable.

Voici un exemple de ce cas d'utilisation :

```
$ rsync -a /var/lib/postgresql/ /var/lib/postgresql2
$ /etc/init.d/postgresql stop
$ rsync -a --delete --checksum /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql start
```

[9] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html

7.6 SAUVEGARDE À CHAUD DES FICHIERS PAR SNAPSHOT DE PARTITION

- Avec certains systèmes de fichiers
- Avec LVM
- Avec la majorité des SAN
- Attention : cohérence entre partitions

Certains systèmes de fichiers (principalement les systèmes de fichiers ZFS et BTRFS) ainsi que la majorité des SAN sont capables de faire une sauvegarde d'un système de fichiers en instantané. En fait, ils figent les blocs utiles à la sauvegarde. S'il est nécessaire de modifier un bloc figé, ils utilisent un autre bloc pour stocker la nouvelle valeur. Cela revient un peu au fonctionnement de PostgreSQL dans ses fichiers.

L'avantage est de pouvoir sauvegarder instantanément un système de fichiers. L'inconvénient est que cela ne peut survenir que sur un seul système de fichiers : impossible dans ce cas de déplacer les journaux de transactions sur un autre système de fichiers pour gagner en performance ou d'utiliser des tablespaces pour gagner en performance et faciliter la gestion de la volumétrie des disques. De plus, comme PostgreSQL n'est pas arrêté au moment de la sauvegarde, au démarrage de PostgreSQL sur la sauvegarde restaurée, ce dernier devra rejouer les journaux de transactions.

Une baie SAN assez haut de gamme pourra disposer d'une fonctionnalité de snapshot cohérent sur plusieurs volumes (« LUN »), ce qui permettra, si elle est bien paramétrée, de réaliser un snapshot de tous les systèmes de fichiers composant la base de façon cohérente.

Néanmoins, cela reste une méthode de sauvegarde très appréciable quand on veut qu'elle ait le moins d'impact possible sur les utilisateurs.

7.7 SAUVEGARDE À CHAUD DES FICHIERS AVEC POSTGRESQL

- PITR : *Point In Time Recovery*
  - nécessite d'avoir activé l'archivage des WAL
  - technique avancée, complexe à mettre en place et à maintenir
  - pas de coupure de service
  - outils dédiés (pgBackRest, barman)
- `pg_basebackup`
  - sauvegarde ponctuelle

Il est possible de réaliser les sauvegardes de fichiers sans arrêter l'instance (à chaud), sans perte ou presque de données, et même avec une possible restauration à un point précis dans le temps. Il s'agit cependant d'une technique avancée (dite PITR, ou *Point In Time Recovery*), qui nécessite la compréhension de concepts non abordés dans le cadre de cette formation, comme l'archivage des journaux de transaction. En général on la mettra en place avec des outils dédiés et éprouvés comme pgBackRest[10] ou barman[11].

Pour une sauvegarde à chaud ponctuelle au niveau du système de fichiers, on peut utiliser `pg_basebackup`, fourni avec PostgreSQL. Cet outil se base sur le protocole de réplication par flux (*streaming*) et les slots de réplications pour créer une copie des répertoires de données. Son maniement s'est assez simplifié depuis les dernières versions. Il dispose en version 13 d'un outil de vérification de la sauvegarde (`pg_verifybackup`).

En raison de la complexité de ces méthodes, on testera d'autant plus soigneusement la procédure de restauration.

[10] https://pgbackrest.org/
[11] https://www.pgbarman.org/

7.8 RECOMMANDATIONS GÉNÉRALES

- Prendre le temps de bien choisir sa méthode
- Bien la tester
- Bien tester la restauration
- Et tester régulièrement !
- Ne pas oublier de sauvegarder les fichiers de configuration

7.9 MATRICE

	Simplicité	Coupure	Restauration	Fragmentation
copie à froid	facile	longue	rapide	conservée
snapshot FS	facile	aucune	rapide	conservée
pg_dump	facile	aucune	lente	perdue
rsync + copie à froid	moyen	courte	rapide	conservée
PITR	difficile	aucune	rapide	conservée

Ce tableau indique les points importants de chaque type de sauvegarde. Il permet de faciliter un choix entre les différentes méthodes.

7.10 CONCLUSION

- Plusieurs solutions pour la sauvegarde et la restauration
- Sauvegarde/Restauration complète ou partielle
- Toutes cohérentes
- La plupart à chaud
- Méthode de sauvegarde avancée : PITR

PostgreSQL propose plusieurs méthodes de sauvegardes et de restaurations. Elles ont chacune leurs avantages et leurs inconvénients. Cependant, elles couvrent à peu près tous les besoins : sauvegardes et restaurations complètes ou partielles, sauvegardes cohérentes, sauvegardes à chaud comme à froid.

La méthode de sauvegarde avancée dite *Point In Time Recovery*, ainsi que les dernières sophistications du moteur en matière de réplication par streaming, permettent d'offrir les meilleures garanties afin de minimiser les pertes de données, tout en évitant toute coupure de service liée à la sauvegarde. Il s'agit de techniques avancées, beaucoup plus complexes à mettre en place et à maintenir que les méthodes évoquées précédemment.

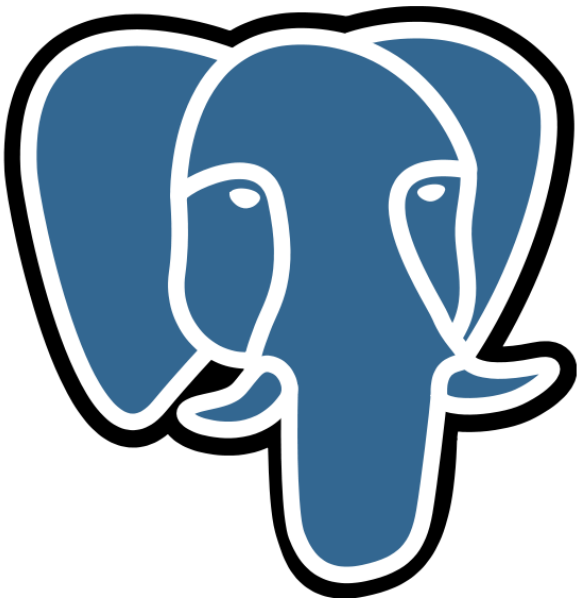
7.10.1 Questions

N'hésitez pas, c'est le moment !

7.11 QUIZ

https://dali.bo/i1_quiz

8/ Supervision



8.1 INTRODUCTION

- Deux types de supervision
  - occasionnelle
  - automatique
- Superviser PostgreSQL et le système
- Pour PostgreSQL, statistiques et traces

Superviser un serveur de bases de données consiste à superviser le SGBD lui-même, mais aussi le système d'exploitation et le matériel. Ces deux derniers sont importants pour connaître la charge système, l'utilisation des disques ou du réseau, qui pourraient expliquer des lenteurs au niveau du SGBD. PostgreSQL propose lui aussi des informations qu'il est important de surveiller pour détecter des problèmes au niveau de l'utilisation du SGBD ou de sa configuration.

Ce module a pour but de montrer comment effectuer une supervision occasionnelle (au cas où un problème survient, savoir comment interpréter les informations fournies par le système et par PostgreSQL) et comment mettre en place une supervision automatique (pour des alertes ou la supervision à long terme).

8.1.1 Menu

- Politique de supervision
- Supervision de PostgreSQL
- Traces : configuration & analyses
- Statistiques d'activité

8.2 POLITIQUE DE SUPERVISION

- Pour quoi ?
- Pour qui ?
- Quels critères ?
- Quels outils

Il n'existe pas qu'une seule supervision. Suivant la personne concernée par la supervision et son objectif, les critères de la supervision seront différents.

Lors de la mise en place de la supervision, il est important de se demander l'objectif de cette supervision, à qui elle va servir, les critères qui importent à cette personne.

Répondre à ces questions permettra de mieux choisir l'outil de supervision à mettre en place, ainsi que sa configuration.

8.2.1 Objectifs de la supervision

- Améliorer/mesurer les performances
- Améliorer l'applicatif
- Anticiper/prévenir les incidents
- Réagir vite en cas de crash

Généralement, les administrateurs mettant en place la supervision veulent pouvoir anticiper les problèmes, qu'ils soient matériels, de performance, de qualité de service, etc.

Améliorer les performances du SGBD sans connaître les performances globales du système est très difficile. Si un utilisateur se plaint d'une perte de performance, pouvoir corroborer ses dires avec des informations provenant du système de supervision aide à s'assurer qu'il y a bien un problème de performances et peut fréquemment aider à résoudre ce problème. De plus, il est important de pouvoir mesurer les gains de performances.

Une supervision des traces de PostgreSQL permet aussi d'améliorer les applications qui utilisent une base de données. Toute requête en erreur est tracée dans les journaux applicatifs, ce qui permet de trouver rapidement les problèmes que les utilisateurs rencontrent.

Un suivi régulier de la volumétrie ou du nombre de connexions permet de prévoir les évolutions nécessaires du matériel ou de la configuration : achat de matériel, création d'index, amélioration de la configuration.

Prévenir les incidents peut se faire en ayant une sonde de supervision des erreurs disques par exemple. La supervision permet aussi d'anticiper les problèmes de configuration. Par exemple, surveiller le nombre de sessions ouvertes sur PostgreSQL permet de s'assurer que ce nombre n'approche pas trop du nombre maximum de sessions configuré avec le paramètre `max_connections` dans le fichier `postgresql.conf`.

Enfin, une bonne configuration de la supervision implique d'avoir configuré finement la gestion des traces de PostgreSQL. Avoir un bon niveau de trace (autrement dit : ni trop, ni pas assez) permet de réagir rapidement après un crash.

8.2.2 Acteurs concernés

- Développeur
  - correction et optimisation de requêtes
- Administrateur de bases de données
  - surveillance, performance
  - mise à jour
- Administrateur système
  - surveillance, qualité de service

Il y a trois types d'acteurs concernés par la supervision.

Le développeur doit pouvoir visualiser l'activité de la base de données. Il peut ainsi comprendre l'impact du code applicatif sur la base. De plus, le développeur est intéressé par la qualité des requêtes que son code exécute. Donc des traces qui ramènent les requêtes en erreur et celles qui ne sont pas performantes sont essentielles pour ce profil.

L'administrateur de bases de données a besoin de surveiller les bases pour s'assurer de la qualité de service, pour garantir les performances et pour réagir rapidement en cas de problème. Il doit aussi faire les mises à jours mineures dès qu'elles sont disponibles.

Enfin, l'administrateur système doit s'assurer de la présence du service. Il doit aussi s'assurer que le service dispose des ressources nécessaires, en terme de processeur (donc de puissance de calcul), de mémoire et de disque (notamment pour la place disponible).

8.2.3 Exemples d'indicateurs - système d'exploitation

- Charge CPU
- Entrées/sorties disque
- Espace disque
- Sur-activité et non-activité du serveur
- Temps de réponse
- Outils Unix habituels :
  - `top`, `atop`, `free`, `df`, `vmstat`, `sar`, `iotop`

Voici quelques exemples d'indicateurs intéressants à superviser pour la partie du système d'exploitation.

La charge CPU (processeur) est importante. Elle peut expliquer pourquoi des requêtes, auparavant rapides, deviennent lentes. Cependant, la suractivité comme la non-activité sont un problème. En fait, si le service est tombé, le serveur sera en sous-activité, ce qui est un excellent indice.

Les entrées/sorties disque permettent de montrer un souci au niveau du système disque. Par exemple, PostgreSQL peut écrire trop à cause d'une mauvaise configuration des journaux de transactions, ou des fichiers temporaires issus de requêtes lourdes ou mal écrites ; ou il peut lire trop de données par manque de cache en RAM, ou de requêtes mal écrites.

L'espace disque est essentiel à surveiller. PostgreSQL ne propose rien pour cela, il faut donc le faire au niveau système. L'espace disque peut poser problème s'il manque, surtout si cela concerne la partition des journaux de transactions.

Unix possède de nombreux outils pour surveiller les différents éléments du système. Les grands classiques sont `top` et ses innombrables clones comme `atop` pour le CPU, `free` pour la RAM, `df` pour l'espace disque, `vmstat` pour la mémoire virtuelle, `iotop` pour les entrées/sorties, ou `sar` (généraliste). Sous Windows, les premiers outils sont bien sûr le Gestionnaire des tâches, et Process Monitor[1] des outils Sysinternals.

Il est conseillé d'avoir une requête étalon dont la durée d'exécution sera testée de temps à autre pour détecter les moments problématiques sur le serveur.

[1] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

8.2.4 Exemples d'indicateurs - base de données

- Nombre de connexions
- Requêtes lentes et/ou fréquentes
- Ratio d'utilisation du cache
- Verrous
- Volumétries
- …

Il existe de nombreux indicateurs intéressant sur les bases : nombre de connexions (en faisant par exemple la différence entre connexions inactives, actives, en attente de verrous), nombre de requêtes lentes et/ou fréquentes, volumétrie (en taille, en nombre de lignes), des ratios (utilisation du cache par exemple)…

8.3 SUPERVISION DE POSTGRESQL

- Supervision occasionnelle
  - sur incident…
- Supervision automatique
  - remonter des informations rapidement
  - archiver, suivre les tendances

La supervision occasionnelle est la conséquence d'une plainte d'un utilisateur : on se contente de réagir à un problème. C'est généralement insuffisant.

Il est important de mettre en place une solution de supervision automatique. Le but est de récupérer périodiquement des données statistiques sur les objets et sur l'utilisation du serveur pour avoir des graphes de tendance, et recevoir des alertes quand des seuils sont dépassés.

8.3.1 Informations internes

- PostgreSQL propose :
  - statistiques d'activité
  - traces
- …mais rien pour les historiser

PostgreSQL propose deux canaux d'informations : les statistiques d'activité (à ne pas confondre avec les statistiques sur les données, à destination de l'optimiseur de requêtes) et les traces applicatives (ou « logs »), souvent dans un fichier comme `postgresql.log` (le nom exact varie avec la distribution et l'installation).

PostgreSQL stocke un ensemble d'informations (métadonnées des schémas, informations sur les tables et les colonnes, données de suivi interne, etc.) dans des tables systèmes qui peuvent être consultées par les administrateurs. PostgreSQL fournit également des vues combinant des informations puisées dans différentes tables systèmes. Ces vues simplifient le suivi de l'activité de la base.

PostgreSQL est aussi capable de tracer un grand nombre d'informations qui peuvent être exploitées pour surveiller l'activité de la base de données.

Pour pouvoir mettre en place un système de supervision automatique, il est essentiel de s'assurer que les statistiques d'activité et les traces applicatives sont bien configurées et il faut aussi leur associer un outil permettant de sauvegarder les données, les alertes et de les historiser.

8.3.2 Outils externes

- Pour conserver les informations
- …et exécuter automatiquement des actions
  - graphiques (Munin, Zabbix…)
  - envoi d'alertes (Nagios, tail_n_mail)

Pour récupérer et enregistrer les informations statistiques, les historiser, envoyer des alertes, il faut faire appel à un outil externe. Cela peut être un outil très simple comme munin ou un outil très complexe comme Nagios ou Zabbix.

8.3.3 check_pgactivity

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - nombreuses métriques remontées
- Développé au départ par Dalibo
  - utilisable indépendamment
- https://github.com/OPMDG/check_pgactivity

Le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios (entre autres).

Au départ, Dalibo utilisait une sonde nommmée `check_postgres` et participait activement à son développement, avec même un commiter dans le projet. Cependant, rapidement, nous nous sommes aperçus que nous ne pouvions pas aller aussi loin que nous le souhaitions. C'est à ce moment que, dans le cadre de sa R&D, Dalibo a conçu `check_pgactivity`.

La plupart des sondes de `check_postgres` y ont été réimplémentées. Le script corrige certaines sondes existantes et en fournit de nouvelles répondant mieux aux besoins de notre supervision, avec notamment un nombre plus important de métriques de performances. Il renvoie directement des ratios par rapport à la valeur précédente plutôt que des valeurs fixes. Pour les besoins les plus simples, le script peut être utilisé de façon autonome, sans nécessité d'installer toute l'infrastructure d'un outil comme Nagios, Icinga ou Grafana.

La supervision d'un serveur PostgreSQL passe par la surveillance de sa disponibilité, des indicateurs sur son activité, l'identification des besoins de maintenance, et le suivi de la réplication le cas échéant. Ci-dessous figurent les sondes `check_pgactivity` à mettre en place sur ces différents aspects. Le site du projet contient toute la documentation de chaque sonde.

Disponibilité :

- `connection` : réalise un test de connexion pour vérifier que le serveur est accessible ;
- `backends` : compte le nombre de connexions au serveur comparé au paramètre `max_connections` ;
- `backends_status` : permet d'obtenir des statistiques plus précises sur l'état des connexions clientes et d'être alerté lorsqu'un certain nombre de connexions clientes sont dans un état donné (*waiting*, *idle in transaction*…) ;
- `uptime` : détecte un redémarrage du serveur ou du rechargement de la configuration.

Vacuum :

- `autovacuum` : suit le fonctionnement de l'autovacuum et des tâches en cours (`VACUUM`, `ANALYZE`, `FREEZE`…) ;
- `table_bloat` : vérifie le volume de données « mortes » et la fragmentation des tables ;
- `btree_bloat` : vérifie le volume de données « mortes » et la fragmentation des index - par rapport à `check_postgres`, le calcul est séparé entre tables et index ;
- `last_analyze` : vérifie si le dernier analyze (relevé des statistiques relatives aux calculs des plans d'exécution) est trop ancien ;
- `last_vacuum` : vérifie si le dernier vacuum (relevé des espaces réutilisables dans les tables) est trop ancien.

Activité :

- `locks` : permet d'obtenir des statistiques plus détaillées sur les verrous obtenus et tient notamment compte des spécificités des *predicate locks* du niveau d'isolation `SERIALIZABLE` ;
- `wal_files` : compte le nombre de segments du journal de transaction présents dans le répertoire `pg_wal` ;
- `longest_query` : permet d'être alerté si une requête est en cours d'exécution depuis plus d'un certain temps ;
- `oldest_xact` : permet d'être alerté si une transaction est ouverte depuis un certain temps sans être utilisée ;
- `oldest_2pc` : calcule l'âge de la plus ancienne transaction préparée (*two-phase commit transaction*) ;

- `oldest_xmin` : repère la plus ancienne transaction de chaque base, et ce à quoi elle est liée (requête, slot…) ;
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL ;
- `hit_ratio` : calcule le *hit ratio* (utilisation du cache de PostgreSQL) ;
- `commit_ratio` : calcule la proportion de `COMMIT` et `ROLLBACK` ;
- `checksum_errors` : détecte l'apparition d'erreurs de sommes de contrôle (à partir de PostgreSQL 12) ;
- `database_size` : suit la volumétrie des bases et leurs variations ;
- `max_freeze_age` : calcule l'âge des plus vieilles lignes stockées dans chaque base pour suivre le bon passage des `VACUUM FREEZE` ;
- `stat_snapshot_age` : calcule l'âge des statistiques d'activité pour repérer un blocage du collecteur ;
- `temp_files` : suivi des fichiers temporaires.

Configuration :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut ;
- `minor_version` : détecte les instances n'ayant pas la dernière version mineure ;
- `settings` : repère un changement des paramètres ;
- `invalid_indexes` : repérer tout index invalide ;
- `pgdata_permission` : vérifie les droits sur `PGDATA` pour éviter un blocage au redémarrage ;
- `table_unlogged` : remonte le nombre de tables *unlogged* ;
- `extensions_versions` : détecte les extensions à mettre à jour.

Réplication & archivage :

- `archiver` : compte le nombre de segments du journal de transaction en attente d'archivage ;
- `archive_folder` : vérifie qu'il n'y a pas de journal manquant dans les archives de sauvegarde PITR ;
- `hot_standby_delta` : calcule le délai de réplication entre un serveur primaire et un serveur secondaire ;
- `is_master` / `is_hot_standby` : vérifie que l'instance est bien démarrée en lecture/écriture, ou une instance secondaire ;
- `is_replay_paused` : vérifie si la réplication est en pause ;
- `replication_slots` : calcule la volumétrie conservée pour chaque slot de réplication.

Sauvegarde physique et logique :

- `backup_label_age` : calcule l'âge du fichier `backup_label` (sauvegardes PITR exclusives) ;
- `pg_dump_backup` : contrôle l'âge et la variation de taille des sauvegardes logiques.

8.3.4 check_postgres

- Script de monitoring PostgreSQL pour Nagios ou MRTG
- Quelques autres sondes
- https://bucardo.org/wiki/Check_postgres

Le script de monitoring `check_postgres` est la première sonde à avoir été écrite pour PostgreSQL. Comme vu précédemment, `check_pgactivity` est un remplaçant plus fonctionnel, donnant plus de métriques, sur un nombre plus limité de sondes.

`check_postgres` évolue cependant toujours et est intéressant dans certains cas : alerte pour les triggers désactivés, taille des relations, estimation du retard en réplication logique (native et Slony), comparaison de schémas, détection de proximité du wraparound. Elle intègre aussi beaucoup de sondes pour l'outil de pooling pgBouncer.

8.4 TRACES

- Configuration
  - traces peu fournies par défaut
- Récupération
  - des problèmes importants
  - des requêtes lentes/fréquentes
- Outils externes de classement

La première information que fournit PostgreSQL sur son activité sort dans les traces. Chaque requête en erreur génère une trace indiquant la requête erronée et l'erreur. Chaque problème de lecture ou d'écriture de fichier génère une trace. En fait, tout problème détecté par PostgreSQL fait l'objet d'un message dans les traces. PostgreSQL peut aussi y envoyer d'autres messages suivant certains événements, comme les connexions, l'activité de processus système en tâche de fond, etc.

Nous allons donc aborder la configuration des traces (où tracer, quoi tracer, quel niveau d'informations). Nous verrons au passage nombre d'informations intéressantes à récupérer. Enfin, nous verrons quelques outils permettant de traiter automatiquement les fichiers de trace.

8.4.1 Configuration des traces : principes

- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
  - durée, fichiers temporaires…
- Tracer certains comportements
  - erreurs

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas à la fois trop lourdes (pour ne pas être submergé par les informations) et incomplètes (il manque des informations). Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

Suivant la configuration réalisée, les journaux applicatifs peuvent contenir quantité d'informations importantes. La plus fréquemment recherchée est la durée d'exécution des requêtes. L'intérêt principal est de récupérer les requêtes les plus lentes. L'autre information importante concerne les messages d'erreur, de niveau PANIC en premier lieu. Ces messages indiquent un état anormal du serveur qui s'est soldé par un arrêt brutal. Ce genre de problème est anormal et doit être surveillé.

8.4.2 Événements exceptionnels tracés

- Crash de PostgreSQL :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

- Rechargement de la configuration :

```
LOG:  received SIGHUP, reloading configuration files
```

- Envoi immédiat d'une alerte
- Outil : tail_n_mail

Les messages `PANIC` sont très importants. Généralement, vous ne les verrez pas au moment où ils se produisent. Un crash va survenir et vous allez chercher à comprendre ce qui s'est passé. Il est possible à ce moment-là que vous trouviez dans les traces des messages `PANIC`, comme celui-ci :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

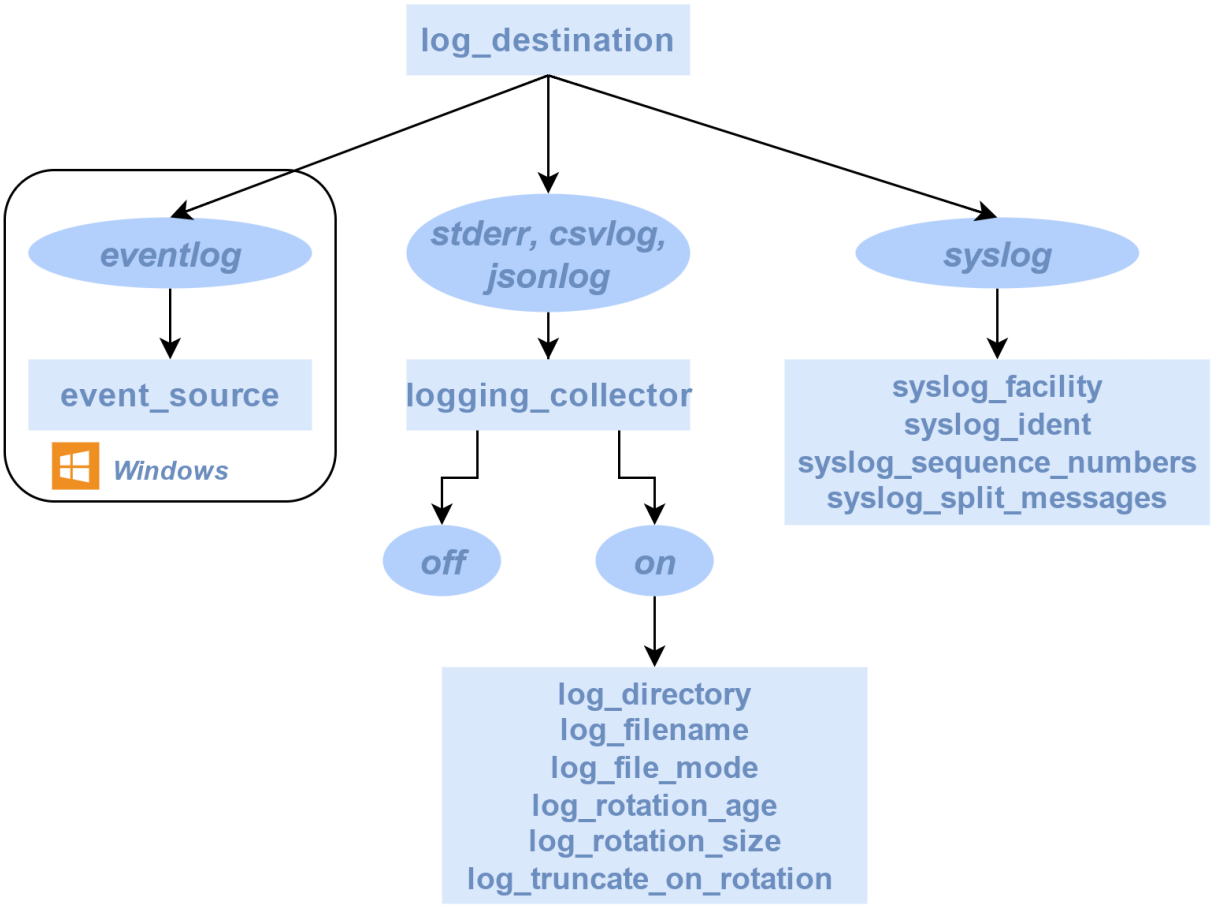
Là, le problème est très simple : PostgreSQL n'arrive pas à créer un journal de transactions à cause d'un manque d'espace sur le disque. Du coup, le système ne peut plus fonctionner, il panique et s'arrête.

Un outil comme tail_n_mail peut aider à détecter automatiquement ce genre de problème et à envoyer un mail à la personne d'astreinte. Il n'est d'ailleurs pas si rare que PostgreSQL, après un problème grave, redémarre si vite qu'il n'y a aucune conséquence visible sérieuse, et que ce genre de détection automatique soit le seul symptôme d'un problème.

Un autre événement à suivre est le changement de la configuration du serveur, et surtout la valeur des paramètres modifiés. PostgreSQL envoie un message niveau `LOG` lorsque la configuration est relue. Il indique aussi les nouvelles valeurs des paramètres, ainsi que les paramètres modifiés qu'il n'a pas pu prendre en compte (cela peut arriver pour tous les paramètres exigeant un redémarrage du serveur).

Là-aussi, tail_n_mail est l'outil adapté pour être prévenu dès que la configuration du serveur est relue.

8.4.3 Où tracer ?



8.4.4 Configuration de la destination des traces

- `log_destination` :
  - `stderr` / `csvlog` / `jsonlog` (v15)
  - `syslog` / `eventlog`
- `logging_collector` : géré par PostgreSQL (Red Hat)
  - `log_directory`, `log_filename`, `log_file_mode`
  - `log_rotation_age`, `log_rotation_size`, `log_truncate_on_rotation`
- Sinon : si `off`, penser à `logrotate` (Debian)
- `syslog` (Unix)
  - `syslog_facility`, `syslog_ident`
  - `syslog_sequence_numbers`, `syslog_split_messages`
- `eventlog` (Windows) : `event_source`

PostgreSQL peut envoyer les traces sur plusieurs destinations selon la valeur de `log_destination` :

stderr, csvlog et jsonlog

`stderr` (valeur par défaut, y compris sur Debian & Red Hat), `csvlog` et `jsonlog`, disponibles sur toutes les plateformes, correspondent à la sortie des erreurs.

La différence entre les trois réside dans le format des traces (respectivement texte simple ou CSV ou JSON). La sortie JSON n'existe que depuis la version 15 mais il est à noter qu'il existe une extension jsonlog[2], par Michaël Paquier, qui offre en plus le format JSON pour les versions antérieures.

Les paramètres suivants sont spécifiques à `stderr`, `csvlog` et `jsonlog`.

`logging_collector` indique ensuite si PostgreSQL doit s'occuper lui-même de la gestion des fichiers de logs.

S'il est à `on` (défaut sous Red Hat/CentOS/Rocky Linux) :

- `log_directory` désigne l'emplacement des journaux applicatifs. Par défaut, il est dans le répertoire de l'instance (sous-répertoire `log`, ou `pg_log` jusqu'en version 9.6 comprise) ;
- `log_filename` indique le nom des fichiers. Sa valeur varie suivant la distribution :

[2] https://github.com/michaelpq/pg_plugins/tree/master/jsonlog

  - `postgresql-%Y-%m-%d_%H%M%S.log` est le défaut des versions compilées (avec rotation quotidienne) ;
  - `postgresql-%a.log` est le défaut sur Red Hat/CentOS/Rocky Linux : on obtient une rotation quotidienne sur une semaine (`postgresql-Mon.log` , `postgresql-Tue.log` , etc.) ;

- `log_file_mode` précise les droits sur les fichiers (`0600` par défaut, les réservant à l'utilisateur sous lequel l'instance tourne). Une rotation est configurable suivant la taille (`log_rotation_size`) et la durée de vie (`log_rotation_age` , souvent `1d` , à garder en cohérence avec `log_filename`) ;

- `log_truncate_on_rotation` à `on` entraîne l'effacement de tout fichier dont le nom serait réutilisé, ce qui est en général une bonne idée si les mêmes noms de fichiers sont réutilisés.

Par contre, sous Debian, `logging_collector` est par défaut à `off` , les paramètres ci-dessus sont ignorés et la gestion des logs est gérée par le système d'exploitation :

- les traces vont dans `/var/log/postgresql/` , donc hors du PGDATA ;
- elles sont nommées en fonction de l'instance, de son nom et du nom fourni lors de la création avec l'outil `pg_ctlcluster` (donc pour l'instance installée par défaut, en version 14, le fichier sera `postgresql-14-main.log`), sauf à modifier le `pg_ctl.conf` de l'instance ;
- la rotation des fichiers est gérée par `logrotate` (comme les autres fichiers de log), et paramétrée dans `/etc/logrotate.d/postgresql-common` avec par défaut une rotation sur 10 jours.

Une instance compilée n'utilise pas non plus le *logging collector*.

syslog

`syslog` fonctionne uniquement sur un serveur Unix et est intéressant pour centraliser la configuration des traces.

Dans cette configuration, il reste à définir le niveau avec `syslog_facility` , et l'identification du programme avec `syslog_ident` . Les valeurs par défaut de ces deux paramètres sont généralement bonnes. Il est intéressant de modifier ces valeurs surtout si plusieurs instances de PostgreSQL sont installées sur le même serveur car cela permet de différencier leur traces.

`syslog_sequence_numbers` préfixe chaque message d'un numéro de séquence incrémenté automatiquement, pour éviter le message `--- last message repeated N times ---` , utilisé par un grand nombre d'implémentations de syslog. Ce comportement est activé par défaut. Quant à `syslog_split_messages` , s'il est activé, les messages envoyés à syslog sont divisés par lignes, elles-mêmes divisées pour tenir sur 1024 octets. Attention à ce que `syslog` soit configuré pour accepter des débits élevés quand on tient à tout tracer.

eventlog

`eventlog` est disponible uniquement sur Windows et alimente le journal des événements. Pour identifier les messages de PostgreSQL, il faut aussi renseigner `event_source` [3], qui par défaut est à « PostgreSQL ».

[3] https://docs.postgresql.fr/current/event-log-registration.html

8.4.5 Niveau des traces

- `log_min_messages`
  - défaut : `panic` / `fatal` / `log` / `error` / `warning`
- `log_min_error_statement`
  - défaut : `error` (ou `warning`)
- `log_error_verbosity`
  - `default` / `terse` / `verbose`

`log_min_messages` est le paramètre à configurer pour avoir plus ou moins de traces. Par défaut, PostgreSQL enregistre tous les messages de niveau `panic`, `fatal`, `log`, `error` et `warning`. Cela peut sembler beaucoup mais, dans les faits, c'est assez discret. Cependant, il est possible de descendre le niveau ou de l'augmenter.

`log_min_error_statement` indique à partir de quel niveau la requête est elle-aussi tracée. Par défaut, la requête n'est tracée que si une erreur est détectée. Généralement, ce paramètre n'est pas modifié, sauf dans un cas précis. Les messages d'avertissement (niveau `warning`) n'indiquent pas la requête qui a généré l'affichage du message. Cela est assez important, notamment dans le cadre de l'utilisation d'antislash dans les chaînes de caractères. On verra donc parfois un abaissement au niveau `warning` pour cette raison.

`log_error_verbosity` vaut `default`, qui convient généralement. Si une requête est tracée, plusieurs lignes peuvent apparaître, chacune de niveau `DETAIL`, `HINT`, `QUERY` ou `CONTEXT`. La valeur `terse` les masque. À l'inverse, `verbose` rajoute encore d'autres informations sur le code source d'origine.

8.4.6 Tracer les requêtes et leur durée

- Toutes les requêtes :
  - `log_min_duration_statement` (ex : `1s`)
  - ou `log_statement` + `log_duration`
- Extrait aléatoire :
  - `log_transaction_sample_rate`
  - `log_statement_sample_rate` + `log_min_duration_sample`

Pour répérer les problèmes de performances, il est intéressant de pouvoir tracer les requêtes et leur durée d'exécution. PostgreSQL propose deux solutions à cela.

log_statement & log_duration :

La première solution disponible concerne les paramètres `log_statement` et `log_duration`. Le premier permet de tracer toute requête exécutée si la requête correspond au filtre indiqué par le paramètre :

- `none` : aucune requête n'est tracée ;
- `ddl` : seules les requêtes DDL (autrement dit de changement de structure) sont tracées ;
- `mod` : seules les requêtes de changement de structure et de données sont tracées ;
- `all` : toutes les requêtes sont tracées.

Le paramètre `log_duration` est un simple booléen. S'il vaut `true` ou `on`, chaque requête exécutée envoie en plus un message dans les traces indiquant la durée d'exécution de la requête. Évidemment, il vaut mieux alors configurer `log_statement` à `all`, ou il sera impossible de dire à quelles requêtes les temps correspondent.

Donc pour tracer toutes les requêtes et leur durée d'exécution, une solution serait de réaliser la configuration suivante :

```
log_statement = 'all'
log_duration = on
```

Une requête générera deux entrées dans les traces, de cette façon :

```
2019-01-28 15:43:27.993 CET [25575] LOG:  statement: SELECT * FROM pg_stat_activity;
2019-01-28 15:43:27.999 CET [25575] LOG:  duration: 7.093 ms
```

log_min_duration_statement :

Il est préférable de désactiver ces deux paramètres et de préférer `log_min_duration_statement`. Son but est d'abord de cibler les requêtes lentes, par exemple celles qui prennent plus de deux secondes à s'exécuter :

```
log_min_duration_statement = '2s'
```

La requête et la durée d'exécution seront alors tracées dans le même message :

```
2019-01-28 15:49:56.193 CET [32067] LOG:
  duration: 2906.270 ms
  statement: insert into t1 select i, i from generate_series(1, 200000) as i;
```

En plus de la trace par `log_min_duration_statement`, rien n'interdit de tracer des requêtes sensibles, notamment le DDL :

```
log_statement = 'ddl'
```

Échantillonnage :

Quelle que soit la méthode, tracer toutes les requêtes peut poser problème pour de simples raisons de volumétrie du fichier de traces. Même s'il est possible de configurer finement la durée à partir de laquelle une requête est tracée, il faut bien comprendre que plus la durée minimale est importante, plus la vision des performances est partielle. Passeront ainsi « sous le radar » des requêtes relativement rapides mais très nombreuses qui, ensemble, peuvent représenter l'essentiel de la charge.

Cela étant dit, laisser 0 en permanence n'est pas recommandé. Il est préférable de configurer ce paramètre à une valeur plus importante en temps normal pour détecter seulement les requêtes longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

Une nouvelle fonctionnalité a donc été ajoutée : tracer une certaine proportion des requêtes ou des transactions.

`log_transaction_sample_rate`, à partir de PostgreSQL 12, indique une proportion de **transactions** à tracer. Par exemple, en le configurant à `0.01`, toutes les requêtes d'un centième des transactions, choisies au hasard, seront tracées.

De manière similaire, à partir de PostgreSQL 13, `log_statement_sample_rate` indique la proportion de **requêtes** à tracer, parmi celles durant plus d'une certaine durée, à indiquer dans `log_min_duration_sample` :

```
log_min_duration_sample = '10ms'
log_statement_sample_rate = 0.01
```

Évidemment, une requête dépassant la durée de `log_min_duration_statement` sera toujours tracée.

8.4.7 Configuration : tracer certains comportements

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits` (mini 1s)
- `log_recovery_conflict_waits` (v14+)

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements. Le paramétrage par défaut est peu bavard, et il est généralement conseillé d'activer tous les paramètres qui suivent.

`log_connections` et son pendant `log_disconnections`, à `on`, permettent de suivre qui se (dé)connecte, depuis où, et durant combien de temps :

```
2019-01-28 13:34:32 CEST LOG:  connection received: host=[local]
2019-01-28 13:34:32 CEST LOG:  connection authorized: user=u1 database=b1
…
2016-09-01 13:34:35 CEST LOG:  disconnection: session time: 0:01:04.634
                               user=u1 database=b1 host=[local]
```

Il est possible de récupérer cette durée de session pour calculer leur durée moyenne. Cette information est importante pour savoir si un outil de pooling de connexions a un intérêt.

`log_autovacuum_min_duration` équivaut à `log_min_duration_statement`, mais pour l'autovacuum. Le but est de tracer son activité, au-delà d'une certaine durée, de vérifier qu'il passe suffisamment fréquemment et rapidement.

`log_checkpoints` à `on` ajoute un message dans les traces pour indiquer qu'un checkpoint commence ou se termine, auquel cas s'ajoutent des statistiques :

```
2019-01-28 13:34:17 CEST LOG: checkpoint starting: xlog
2019-01-28 13:34:20 CEST LOG: checkpoint complete:
  wrote 13115 buffers (80.0%);
  0 WAL file(s) added, 0 removed, 0 recycled;
  write=3.007 s, sync=0.324 s, total=3.400 s;
  sync files=16, longest=0.285 s, average=0.020 s;
  distance=404207 kB, estimate=404207 kB
```

Le message indique donc en plus le nombre de blocs écrits sur disque, le nombre de journaux de transactions ajoutés, supprimés et recyclés. Il est rare que des journaux soient ajoutés, ils sont plutôt recyclés. Des journaux sont supprimés quand il y a eu une très grosse activité qui a généré plus de journaux que d'habitude. Les statistiques incluent aussi la durée des écritures, de la synchronisation sur disque, la durée totale, etc. Le plus important est de pouvoir vérifier que l'écriture des checkpoints est bien régulière (essentiellement périodique).

`log_lock_waits` à `on` permet de tracer les attentes de verrous (par exemple, un `UPDATE` bloqué par un autre `UPDATE`, un `SELECT` bloqué par `TRUNCATE` ou un `VACUUM FULL`, etc…) Lorsque l'attente dépasse la durée indiquée par le paramètre `deadlock_timeout` (1 seconde par défaut), un message est enregistré, comme dans cet exemple :

```
2019-01-28 13:38:40 CEST LOG:  process 15976 still waiting for
                               AccessExclusiveLock on relation 26160 of
                               database 16384 after 1000.123 ms
2019-01-28 13:38:40 CEST STATEMENT:  DROP TABLE t1;
```

Ici, un `DROP TABLE` attend depuis 1 seconde de pouvoir poser un verrou exclusif sur une relation.

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur certains objets, ce qui peut diminuer fortement les performances. Ces messages peuvent permettre d'analyser la cause première d'une accumulation de verrous, à condition que les requêtes soient tracées.

En version 14 apparaît le paramètre `log_recovery_conflict_waits`. Ce dernier, une fois activé, permet de tracer toute attente due à un conflit de réplication. Il n'est donc valable et pris en compte que sur un serveur secondaire.

8.4.8 Repérer les fichiers temporaires

- Exemple :

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp9894.0",
      size 26927104
```

- `log_temp_files` à activer !
- Alerte : problème potentiel de performances

Quand PostgreSQL ne peut effectuer un tri en mémoire, il le fait sur disque dans un fichier temporaire, ce qui est beaucoup plus lent qu'en mémoire, même avec un SSD. Typiquement, cela concerne le tri de données et le hachage, quand la valeur du paramètre `work_mem` ne permet pas de tout faire en mémoire. Cela ne sera pas forcément gênant pour une grosse requête ponctuelle, mais, répétés, ces fichiers peuvent avoir un impact sur la performance du système. Ils sont parfois inévitables quand on brasse beaucoup de données.

Être averti lors de la création de ce type de fichiers peut être intéressant, mais ils sont parfois trop fréquents pour que ce soit réaliste. Il est préférable de faire analyser après coup un fichier de traces pour savoir combien de fichiers temporaires ont été créés, et de quelles tailles. Cela peut mener à vérifier les requêtes exécutées, les optimiser, vérifier la configuration, réviser la valeur de `work_mem`…

Le paramètre `log_temp_files` à 0 permet de tracer toutes les créations de fichiers temporaires, comme ici :

```
2019-01-28 13:41:11 CEST LOG:  temporary file: path
                               "base/pgsql_tmp/pgsql_tmp15617.1",
                               size 59645952
```

Pour le même tri, il peut y avoir de nombreux fichiers temporaires. De plus, la requête est aussi tracée, et si elle est longue et fréquente, le volume de traces peut être conséquent.

8.4.9 Configuration : divers

- `log_line_prefix`
  - Conseillé : `%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h`
- `lc_messages` = `C`
- `log_timezone` = `'Europe/Paris'`

Le paramètre `log_line_prefix` permet d'ajouter un préfixe à une trace. Le défaut (`'%m [%p] '`, soit horodatage et numéro de processus), est insuffisant :

```
2021-02-05 14:12:12.343 UTC [2917] LOG:  duration: 3.276 ms
                           statement: SELECT count(*) FROM pgbench_branches ;
```

Il est conseillé de rajouter le nom de l'application cliente, le nom de l'utilisateur, le nom de la base, etc. Une valeur habituellement conseillée pour pgBadger[4], pour une sortie vers `stderr`, est :

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
```

ce qui nous donnera ce genre de traces :

```
2021-02-05 14:30:01 UTC [3006]: user=durand,db=bench,app=test,client=[local]
    LOG:  duration: 0.184 ms  statement: SELECT count(*) FROM pgbench_branches ;
```

Pour une sortie vers `syslog`, l'horodatage est inutile :

```
log_line_prefix = 'user=%u,db=%d,app=%a,client=%h '
```

Par défaut, les traces sont enregistrées dans la locale par défaut du serveur. Des traces en français peuvent présenter certains intérêts pour des débutants, mais ont plusieurs gros inconvénients : un moteur de recherche renverra beaucoup moins de résultats avec des traces en français qu'en anglais, et les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il vaut mieux préciser systématiquement :

```
lc_messages = 'C'
```

[4] https://pgbadger.darold.net/documentation.html#POSTGRESQL-CONFIGURATION

Quant à `log_timezone`, il permet de choisir le fuseau horaire pour l'horodatage des traces. C'est inestimable quand on administre différents serveurs dispersés sur la planète.

```
log_timezone = 'UTC'
```

```
log_timezone = 'Europe/Paris'
```

8.5 OUTILS D'ANALYSE DES TRACES

- Beaucoup d'outils existent
  - en temps réel / rétro-analyse
  - généralistes / spécifiques PostgreSQL
- Exemples :
  - pgBadger
  - logwatch
  - tail_n_mail

Il existe de nombreux programmes qui analysent les traces. On peut distinguer deux catégories :

- ceux qui le font en temps réel ;
- ceux qui le font après coup (de la rétro-analyse en fait).

Mais également :

- ceux généralistes, connaissant plus ou moins bien beaucoup d'outils et logiciels ;
- ceux dédiés à PostgreSQL.

L'analyse en temps réel des traces permet de réagir rapidement à certains messages. Par exemple, il est important d'avoir une réaction rapide à l'archivage échoué d'un journal de transactions, ainsi qu'en cas de manque d'espace disque. Dans cette catégorie, il existe des outils généralistes comme logwatch[5], et des outils spécifiques pour PostgreSQL comme tail_n_mail[6].

L'analyse après coup permet une analyse plus fine, se terminant généralement par un rapport en HTML, parfois avec des graphes. Cette analyse plus fine nécessite des outils spécialisés. Il en a existé plusieurs qui ne sont plus maintenus. La référence dans le domaine est pgBadger[7].

[5] https://sourceforge.net/projects/logwatch/
[6] https://bucardo.org/tail_n_mail/
[7] https://pgbadger.darold.net/

8.5.1 pgBadger

- Site officiel : https://pgbadger.darold.net/
- Licence : PostgreSQL
- Analyse des traces de durée d'exécution des requêtes
- Analyse des traces du `VACUUM`, des connexions, des checkpoints
- Compatible syslog, stderr, csvlog

Gilles Darold a créé pgBadger, un analyseur des journaux applicatifs de PostgreSQL. Il permet de générer des rapports détaillés depuis ceux-ci. pgBadger est très souvent utilisé pour déterminer les requêtes à améliorer en priorité pour accélérer son application basée sur PostgreSQL. C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL, au point qu'il est cité dans le manuel de PostgreSQL.

pgBadger est écrit en Perl et est facilement extensible si vous avez besoin de rapports spécifiques.

Il est conçu pour traiter rapidement de gros fichiers de traces avec une mémoire réduite, mais permet d'exploiter plusieurs CPU pour accélérer considérablement l'analyse.

8.5.2 pgBadger : exemple de rapport

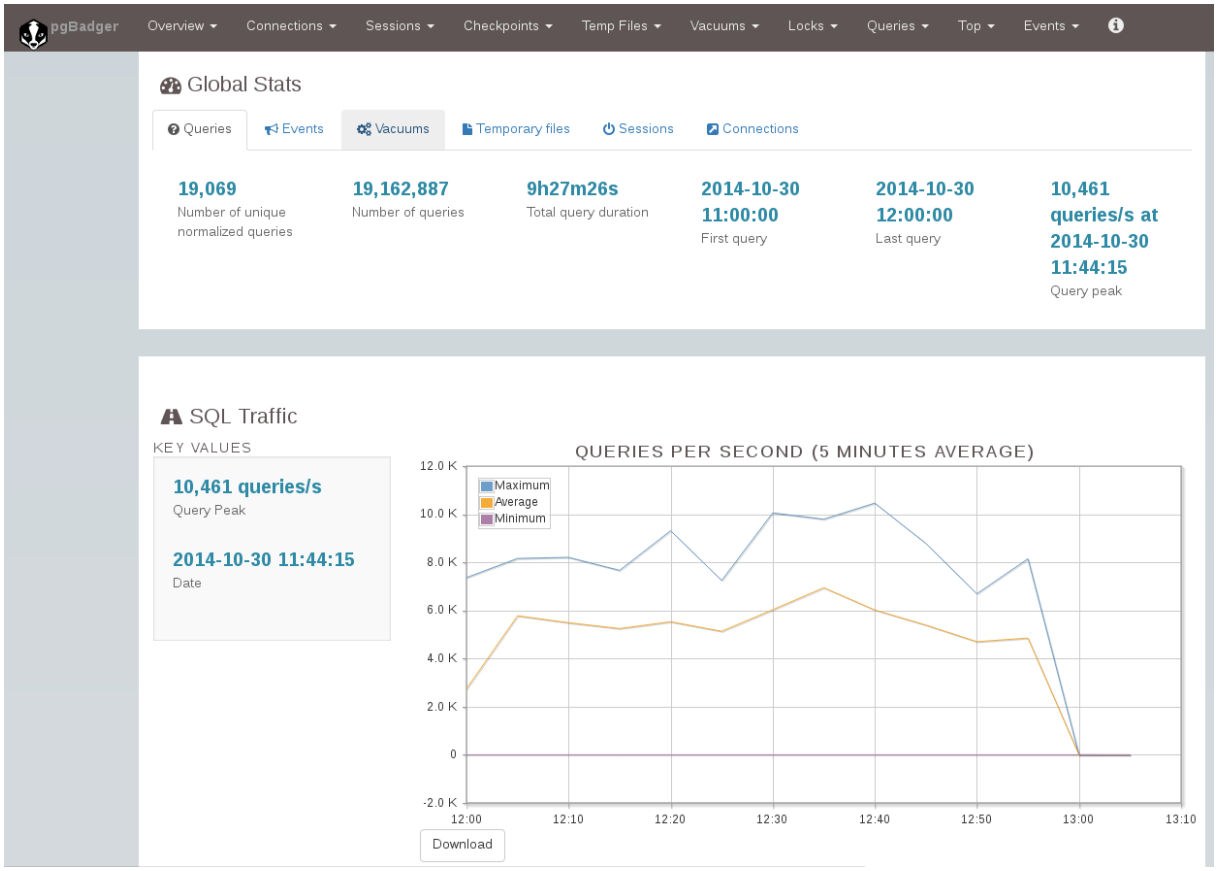


Figure 8/ .1: Capture pgBadger

8.5.3 Utiliser pgBadger

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations
  - sur les requêtes (normalisées) et leur durée d'exécution
  - sur les connexions et sessions
  - sur les checkpoints
  - sur l'autovacuum
  - sur les verrous
  - etc.
- Génération d'un rapport HTML très détaillé

pgBadger s'utilise en ligne de commande : il suffit de lui fournir le ou les fichiers de trace à analyser et il rend un rapport HTML sur les requêtes exécutées, sur les connexions, sur les bases, etc. Le rapport est très complet. Les graphes sont zoomables. Les requêtes sont reformatées pour plus de lisibilité.

8.5.4 Configurer PostgreSQL pour pgBadger

- Minimum :
  - `log_destination`
  - `log_line_prefix`
  - `lc_messages` = `C`
- Base :
  - `log_connections`, `log_disconnections`
  - `log_checkpoints`
  - `log_lock_waits`
  - `log_temp_files`
  - `log_autovacuum_min_duration`
- Pour un audit :
  - `log_min_duration_statement = 0` (attention !)

pgBadger a besoin d'un minimum d'informations dans les traces : timestamp (`%t`), pid (`%p`) et numéro de ligne dans la session (`%l`). Il n'y a pas de conseil particulier sur la destination des traces (en dehors de `eventlog` que pgBadger ne sait pas traiter). De même, le préfixe des traces est laissé au choix de l'utilisateur. Dans certains cas, il faudra le préciser à pgBadger avec l'option `--prefix` (par exemple si la valeur de `log_line_prefix` a changé entre le début et la fin du fichier). Une configuration courante et vraiment informative est par exemple :

```
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Noter que même sans cela, pgBadger essaie de récupérer les informations sur les adresses IP, les utilisateurs connectés, les bases de données à partir des traces sur les connexions. Ajouter le joker `%e` (code SQLState) permet d'obtenir un tableau sur les erreurs.

La langue des traces doit être l'anglais (`lc_messages` à `C`), ce qui de toute manière est la valeur conseillée.

Pour tracer les requêtes, il est préférable de passer par `log_min_duration_statement` plutôt que `log_statement` et `log_duration` : pgBadger fera plus facilement l'association entre chaque requête et sa durée :

```
log_min_duration_statement = 0
log_statement = none
log_duration = off
```

Comme déjà dit plus haut, `log_min_duration_statement = 0` peut générer un énorme volume de traces et n'est souvent configuré ainsi que le temps d'un audit. Avec une valeur supérieure, pgBadger ne verra absolument pas les requêtes les plus rapides.

Il est aussi conseillé de tirer parti d'autres informations dans les traces :

- `log_checkpoints` pour des statistiques sur les checkpoints ;
- `log_connections` et `log_disconnections` pour des informations sur les connexions et déconnexions ;
- `log_lock_waits` pour des statistiques sur les verrous en attente ;
- `log_temp_files` pour des statistiques sur les fichiers temporaires ;
- `log_autovacuum_min_duration` pour des statistiques sur l'activité de l'autovacuum.

8.5.5 Options de pgBadger

- Génération :

```
pgbadger … -o rapport.html postgresql-Mon.log postgresql-Tue.log …
```

- Très nombreuses options, dont :
  - `--outfile`
  - `--prefix`
  - `--begin` / `--end`
  - `--dbname`, `--dbuser`, `--dbclient`, `--appname`
  - `--jobs`

Pour l'utilisation la plus simple, il suffit de fournir au script en paramètre les noms des différents fichiers de traces, éventuellement compressés, pour obtenir le rapport sur tous les événements qu'il y trouvera.

Il existe énormément d'options[8], et Gilles Darold en rajoute fréquemment. L'aide fournie sur le site web officiel les cite intégralement.

Parmi les plus utiles, `--outfile` permet d'indiquer le nom du rapport (par défaut `out.html`).

`--prefix` permet de préciser une valeur de `log_line_prefix` si la détection automatique échoue.

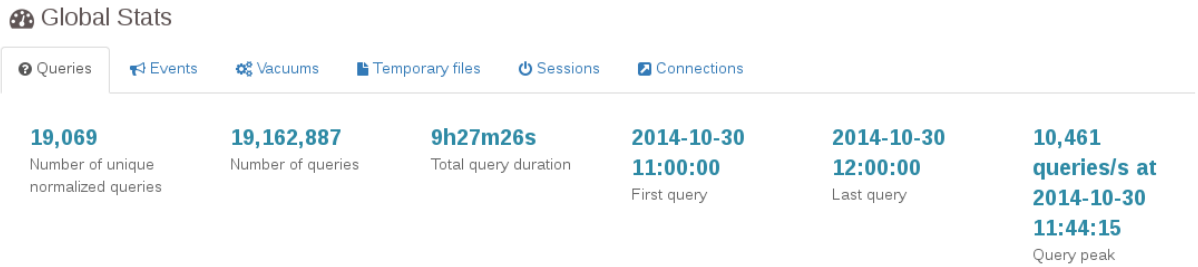
Le rapport peut être restreint à une tranche horaire précise avec `--begin` et `--end` (par exemple `--begin '2019-04-01 16:00:00`).

Pour mieux cibler le rapport, il est possible de restreindre l'analyse à un utilisateur (`--dbuser`), une base de données (`--dbname`), une machine cliente (`--dbclient`), ou une application (`--appname`, si elle définit bien `application_name` à la connexion).

`--jobs` permet de paralléliser la lecture des traces sur plusieurs processeurs. Attention, de très gros fichiers les satureront probablement plusieurs minutes.

[8] https://pgbadger.darold.net/documentation.html#SYNOPSIS

8.5.6 pgBadger : exemple 1



Au tout début du rapport, pgBadger donne des statistiques générales sur les fichiers de traces.

Dans les informations importantes se trouve le nombre de requêtes normalisées. En fait, des requêtes telles que :

```
SELECT * FROM utilisateurs WHERE id = 1;
```

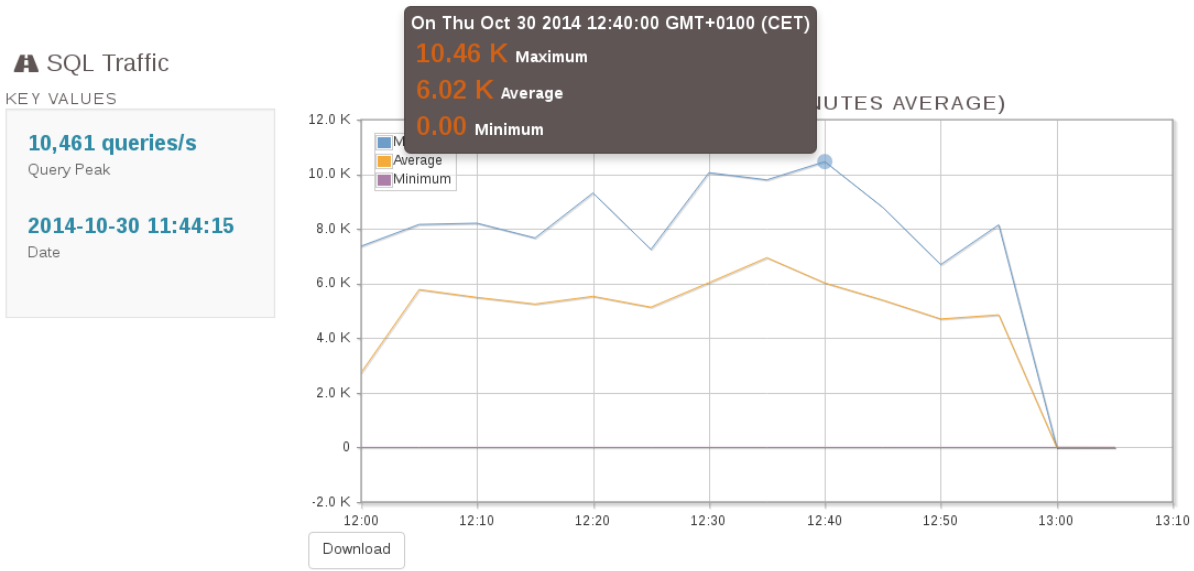
et

```
SELECT * FROM utilisateurs WHERE id = 2;
```

sont différentes car elles ne vont pas récupérer la même fiche utilisateur. Cependant, en enlevant la partie constante, les requêtes sont identiques. La seule différence est la ligne récupérée mais pas la requête. pgBadger est capable de faire cette différence. Toute constante, qu'elle soit de type numérique, textuelle, horodatage ou booléenne, peut être supprimée de la requête. Dans l'exemple ci-dessus, pgBadger a comptabilisé environ 19 millions de requêtes, mais seulement 19 069 requêtes différentes après normalisation. Ceci est important dans le fait où nous n'allons pas devoir travailler sur plusieurs millions de requêtes mais « seulement » sur 19 000.

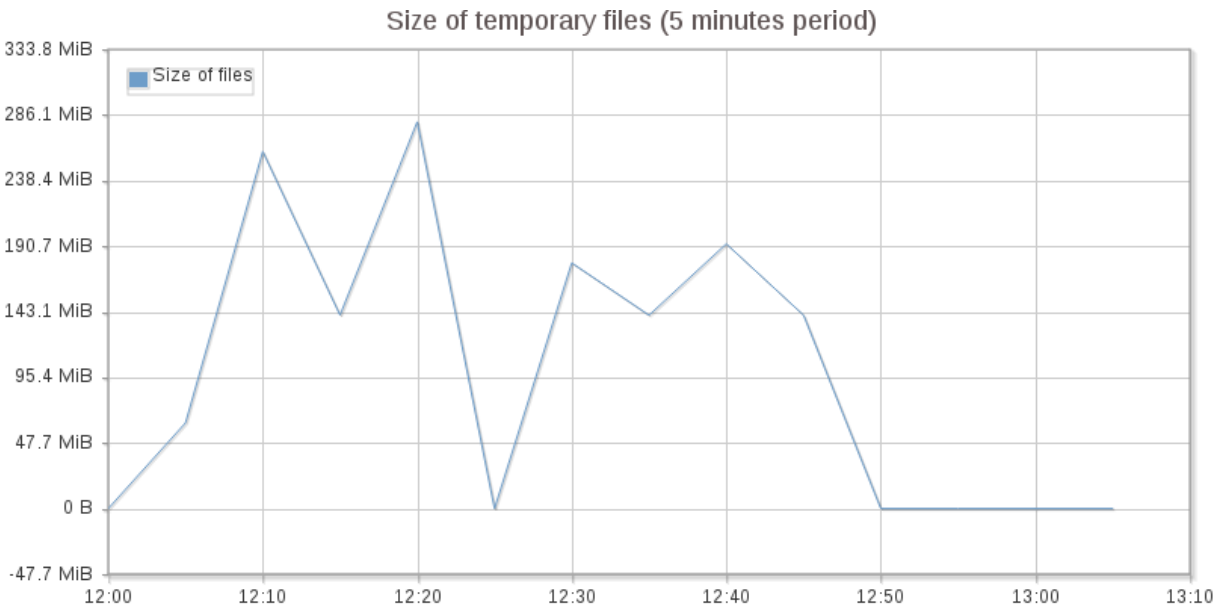
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 9 heures d'exécution de requêtes. Cependant, les traces ne couvrent que 11 h à 12 h, soit une heure. Cela indique que le serveur est assez sollicité. Il est fréquent que la durée d'exécution sérielle des requêtes soit plusieurs fois plus importantes que la durée des traces.

8.5.7 pgBadger : exemple 2



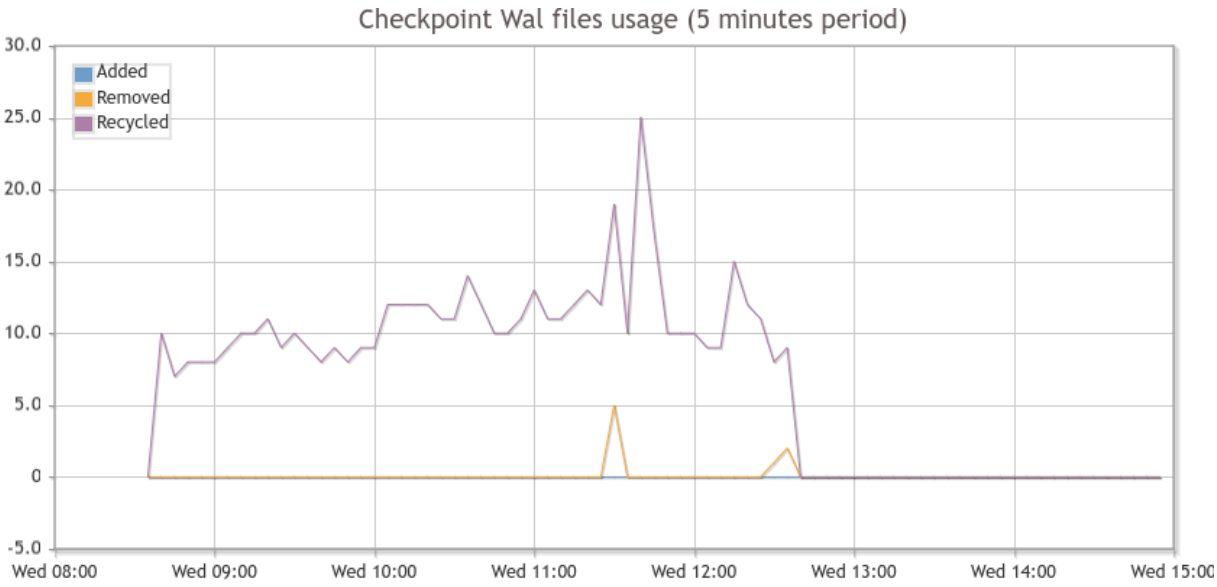
Ce graphe indique le nombre de requêtes par seconde : en fait, environ 33 requêtes/s en pointe.

8.5.8 pgBadger : exemple 3



Ce graphe affiche en vert le nombre de fichiers temporaires créés sur la durée des traces. La ligne bleue correspond à la taille des fichiers. Nous remarquons ainsi la création à peu près régulière de fichiers temporaires, à raison d'environ 200 Mo par tranche de 5 minutes.

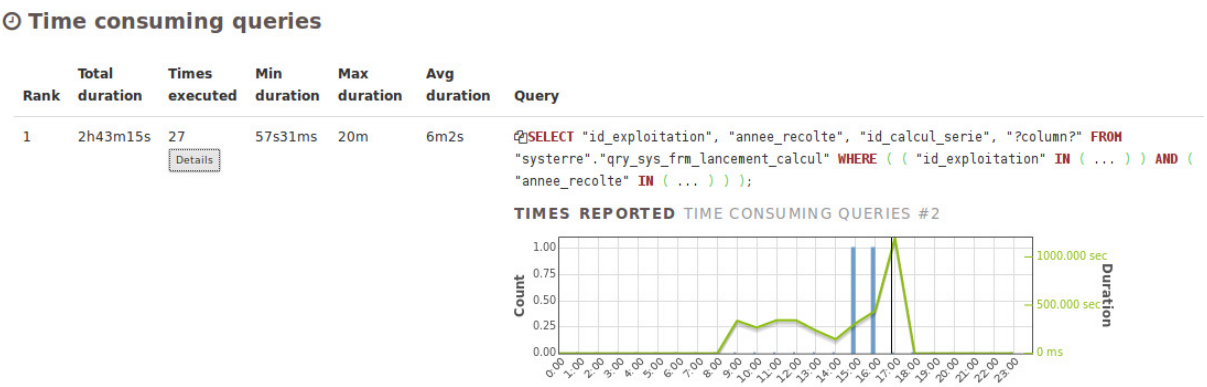
8.5.9 pgBadger : exemple 4



De grosses écritures peuvent mener à un nombre important de journaux. Cette courbe montre une création relativement régulière de journaux. En fait, il s'agit uniquement de recyclage de journaux existants : PostgreSQL n'a pas à en créer et en initialiser en masse. Le pic n'est que de 5 journaux de 16 Mo à la minute.

Plus bas dans l'onglet *Checkpoints* figurent des informations sur l'écart (en octets) entre deux checkpoints, la durée d'écriture, la durée de synchronisation sur le disque, et le nombre d'avertissements sur des checkpoints non périodiques.

8.5.10 pgBadger : exemple 5



Le plus important est certainement l'onglet *Top/Time consuming queries*. Il liste les requêtes qui ont pris le plus de temps, que ce soit parce que les requêtes en question sont vraiment très lentes ou parce qu'elles sont exécutées un très grand nombre de fois.

Ci-dessus n'est affichée que la première requête de la liste. Le bouton *Details* permet d'afficher la courbe indiquant les heures d'occurrences et les durées.

Nous remarquons d'ailleurs dans cet exemple qu'avec la seule requête affichée, nous arrivons à un total de 2 h 43. Le premier exemple nous indique que l'exécution sérielle des requêtes aurait pris 9 heures. En ne travaillant que sur elle, nous travaillons en fait sur presque le tiers du temps total d'exécution des requêtes comprises dans les traces.

Les autres requêtes les plus consommatrices ne sont pas listées ici, mais il est peu probable que l'on ait à travailler sur les 19 000 requêtes normalisées. Il est fréquent que l'essentiel de la charge soit contenu dans les quelques premières requêtes du tableau. Ce sont évidemment les premières cibles pour une tentative d'optimisation : réécriture, modification du paramétrage, index dédiés…

8.5.11 logwatch

- Outil externe écrit en Perl
  - https://sourceforge.net/projects/logwatch/
  - Licence MIT
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs
- Exemple :

```
/usr/sbin/logwatch --detail Med --service postgresql --range All
```

Surveiller ses journaux applicatifs (ou fichiers de trace) est une activité nécessaire mais bien souvent rébarbative. De nombreux programmes existent pour nous faciliter la tâche, mais le propre de ces programmes est d'être exhaustif. De plus, ils demandent une action de la part de l'administrateur, à savoir : penser à aller les regarder.

logwatch est une petite application permettant d'analyser les journaux de nombreux services et de produire un rapport synthétique. Le nombre de services connus est impressionnant et il est simple d'en ajouter de nouveaux. logwatch est le plus souvent intégré à votre distribution Linux. Depuis la version 7.4.2 il sait analyser les traces de PostgreSQL.

Après son installation, il se lancera tous les jours grâce au fichier `/etc/cron.daily/00logwatch`. Vous recevrez tous les jours par mail les rapports des événements importants détectés dans les fichiers de traces. Vous pouvez aussi le lancer manuellement ainsi :

```
/usr/sbin/logwatch --detail Low --service postgresql --range All
```

`--range All` indique que l'on veut un rapport sur tous les fichiers de traces existants. La valeur par défaut est `Yesterday`. Pour n'avoir un rapport que sur la journée, choisir `Today`.

Avec le niveau de détail minimal (`--detail` à `Low`), seuls les messages de type `FATAL`, `PANIC` et `ERROR` seront remontés. Avec la valeur `Med`, apparaîtront aussi les messages de type `WARNING` et `HINTS`.

Exemple de résultat :

```
################### Logwatch 7.3.6 (05/19/07) ####################
       Processing Initiated: Tue Dec 13 12:28:46 2011
       Date Range Processed: all
       Detail Level of Output: 5
       Type of Output/Format: stdout / text
       Logfiles for Host: devel
 ##################################################################
---------------------- PostgreSQL Begin ------------------------
Fatals:
-------
9 times:
[2011-12-04 04:28:46 +/-9 day(s)]  password authentication failed for
                                   user "postgres"
8 times:
[2011-11-21 10:15:01 +/-11 day(s)]  terminating connection due to
                                    administrator command
…
Errors:
-------
7 times:
[2011-11-14 12:20:44 +/-58 minute(s)]  relation "COUNTRIES" does not exist
5 times:
[2011-11-14 12:21:24 +/-59 minute(s)]  syntax error at or near
                                       "role_permission_view"
…
Warnings:
---------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  terminating connection because of
                                   crash of another server process
Hints:
------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  In a moment you should be able to
                                   reconnect to the database and repeat
                                   you command.
14 times:
[2011-12-08 09:47:16 +/-19 day(s)]  No function matches the given name and
                                    argument types. You might need to add
                                    explicit type casts.
---------------------- PostgreSQL End ------------------------
###################### Logwatch End #########################
```

logwatch dispose de nombreuses options, et la page de manuel est certainement la meilleure documentation à l'heure actuelle.

8.5.12 tail_n_mail

- Outil externe écrit en Perl
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs

tail_n_mail est un outil écrit par la société EndPointCorporation. Son but est d'analyser périodiquement le contenu des fichiers de traces et d'envoyer un mail en cas de la détection d'un motif d'intérêt. Par exemple, il peut envoyer un mail lorsqu'il rencontre un message de niveau `PANIC` ou `FATAL` dans les traces. Ainsi une personne d'astreinte sera prévenue rapidement et pourra agir en conséquence.

8.5.13 Configurer tail_n_mail

```
EMAIL: astreinte@dalibo.com
MAILSUBJECT: HOST Postgres fatal errors (FILE)
FILE: /var/log/postgresql-%Y-%m-%d.log
INCLUDE: PANIC:
INCLUDE: FATAL:
EXCLUDE: database ".+" does not exist
INCLUDE: temporary file
INCLUDE: reloading configuration files
```

Les clés `INCLUDE` et `EXCLUDE` permettent d'indiquer les motifs à inclure et à exclure respectivement. Tout motif non inclus ne sera pas pris en compte. Cette configuration permet donc d'envoyer un mail à l'adresse `astreinte@dalibo.com` à chaque fois qu'un message contenant les mots `PANIC`, `FATAL`, `temporary file` ou `reloading configuration files` sont enregistrés dans les traces. Par contre, tous les messages contenant la phrase `database ... does not exist` ne sont pas pris en compte.

8.5.14 tail_n_mail : exemple

Exemple:

```
[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement: user=root,db=rojogrande
        FATAL: password authentication failed for user "root"
```

Voici un exemple de mail envoyé:

```
Matches from /var/log/postgresql/postgresql-10-main.log: 42
Date: Fri Jan  1 10:34:00 2010
Host: pollo

[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement:  user=root,db=rojogrande FATAL:  password authentication failed
                                            for user "root"

[2] Between lines 147999 and 148213, occurs 2 times.
First:  Jan  1 10:31:01 rojogrande postgres[3561]
Last:   Jan  1 10:31:10 rojogrande postgres[15312]
Statement: FATAL  main: write to worker pipe failed -(9) Bad file descriptor

[3] (from line 152341)
PANIC:  could not locate a valid checkpoint record
```

8.6 STATISTIQUES D'ACTIVITÉ

- Configuration
- Liste des vues statistiques
- Outils externes de classement

Les statistiques sont certainement les informations les plus simples à récupérer par un système de supervision. Il faut dans un premier temps s'assurer que la configuration est adéquate. Ceci fait, il est possible de lire les statistiques disponibles dans les vues proposées par défaut. Enfin, il existe quelques outils capables de récupérer des informations provenant des tables statistiques de PostgreSQL. Leur mise en place permettra une supervision facilitée.

8.6.1 Statistiques d'activité - configuration 1

- Tracer l'activité :
  - `track_activities` = `on`
- S'assurer que les requêtes ne sont pas tronquées :
  - `track_activity_query_size` = `10000` ou +
- Récupérer l'identifiant de requête
  - `compute_query_id` = on

Il est important d'avoir des informations sur les sessions en cours d'exécution sur le serveur. Cela se fait grâce au paramètre `track_activities`. Il est à `on` par défaut. Ainsi, dans la vue `pg_stat_activity` qui liste les sessions, les colonnes `xact_start`, `query_start`, `state_change`, `wait_event_type`, `wait_event`, `state` et `query` sont renseignées.

```
bench=# SELECT * FROM pg_stat_activity
        WHERE backend_type = 'client backend' \gx

-[ RECORD 1 ]----+--------------------------------------
datid            | 16425
datname          | bench
pid              | 3006
```

```
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | test
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2021-02-05 14:29:42.833102+00
xact_start       | 2021-02-05 17:28:12.157115+00
query_start      | 2021-02-05 17:28:12.157115+00
state_change     | 2021-02-05 17:28:12.157117+00
wait_event_type  |
wait_event       |
state            | active
backend_xid      |
backend_xmin     | 186938
query_id         |
query            | select * from pg_stat_activity where backend_type = 'client
 ↪  backend'
backend_type     | client backend
```

Il est à noter que la requête indiquée dans la colonne `query` est tronquée à 1024 caractères par défaut. En pratique, cette limite est vite atteinte par de longues requêtes. Il est conseillé de l'augmenter à quelques kilooctets (paramètre `track_activity_query_size`).

Depuis la version 14, il est possible d'avoir en plus l'identifiant de la requête. Pour cela, il faut activer le paramètre `compute_query_id`.

8.6.2 Statistiques d'activité - configuration 2

- `track_counts` = `on`
- `track_io_timing` = `on`
- `track_functions` = `off` / `pl` / `all`

Par défaut, le paramètre `track_counts` est à `on`. Le collecteur de statistiques est alors capable de récupérer des informations sur des nombres de lignes lues, insérées, mises à jour, supprimées, vivantes, mortes, etc., et de blocs lus dans le cache de PostgreSQL (*hit*) ou en dehors (*read*).

`track_io_timing` réalise un chronométrage des opérations de lecture et écriture disque. Il complète les champs `blk_read_time` et `blk_write_time` dans les tables `pg_stat_database` et `pg_stat_statements` (si cette extension[9] est installée). Il ajoute des traces suite à un `VACUUM` ou un `ANALYZE` exécutés par le processus `autovacuum` Dans les plans d'exécutions (avec

[9] https://dali.bo/x2_html#pg_stat_statements

`EXPLAIN (ANALYZE,BUFFERS)`), il permet l'affichage du temps passé à lire hors du cache de PostgreSQL(sur disque ou dans le cache de l'OS) :

```
I/O Timings: read=2.062
```

Avant d'activer `track_io_timing` sur une machine peu performante, vérifiez avec l'outil `pg_test_timing`[10] que la quasi-totalité des appels dure moins d'une nanoseconde.

PostgreSQL sait aussi récupérer des statistiques sur les routines stockées. Il faut activer le paramètre `track_functions` qui a trois valeurs : `off` pour ne rien récupérer, `pl` pour récupérer les statistiques sur les procédures stockées en `PL/*` et `all` pour récupérer les statistiques des fonctions quelque soit leur langage (notamment celles en C). Les résultats (nombre et durée d'exécution) peuvent se suivre dans la vue `pg_stat_user_functions`.

8.6.3 Statistiques d'activité - configuration 3

- `stats_temp_directory` (<v15)
  - répertoire contenant les fichiers temporaires des statistiques
  - copié vers `pg_stat` lors d'un arrêt propre
  - à monter sur du tmpfs

Avant la version 15, il existe un processus collecteur de statistiques, qui fonctionne ainsi :

- il est lancé au démarrage de PostgreSQL (il est d'ailleurs impossible de le désactiver complètement) ;
- il collecte ses statistiques dans le répertoire ciblé par le paramètre `stats_temp_directory` ;
- il met à jour les fichiers dès que les autres processus lui fournissent des statistiques ;
- à l'arrêt de PostgreSQL, il recopie les fichiers du répertoire temporaire dans le répertoire `$PGDATA/pg_stat`.

L'intérêt de ce fonctionnement est de pouvoir copier le fichier de statistiques sur un disque très rapide (comme un disque SSD), voire dans de la mémoire montée en disque comme `tmpfs` (c'est d'ailleurs le défaut sur Debian).

À partir de la version 15, les statistiques sont stockés en mémoire partagée et il n'y a plus de collecteur.

[10] https://docs.postgresql.fr/current/pgtesttiming.html

8.6.4 Statistiques d'activité : perte

- Les statistiques d'activité sont perdues en cas de crash ou restauration
  - `ANALYZE` (voire `VACUUM`)

En cas d'arrêt brutal de PostgreSQL ou de restauration physique, les statisiques d'activité sont perdues (pas les statistiques sur les données). Lancer un `ANALYZE` est préférable pour éviter que l'autovacuum tarde à nettoyer les tables ou rafraîchir les statistiques sur les données.

8.6.5 Informations intéressantes à récupérer

Sur :

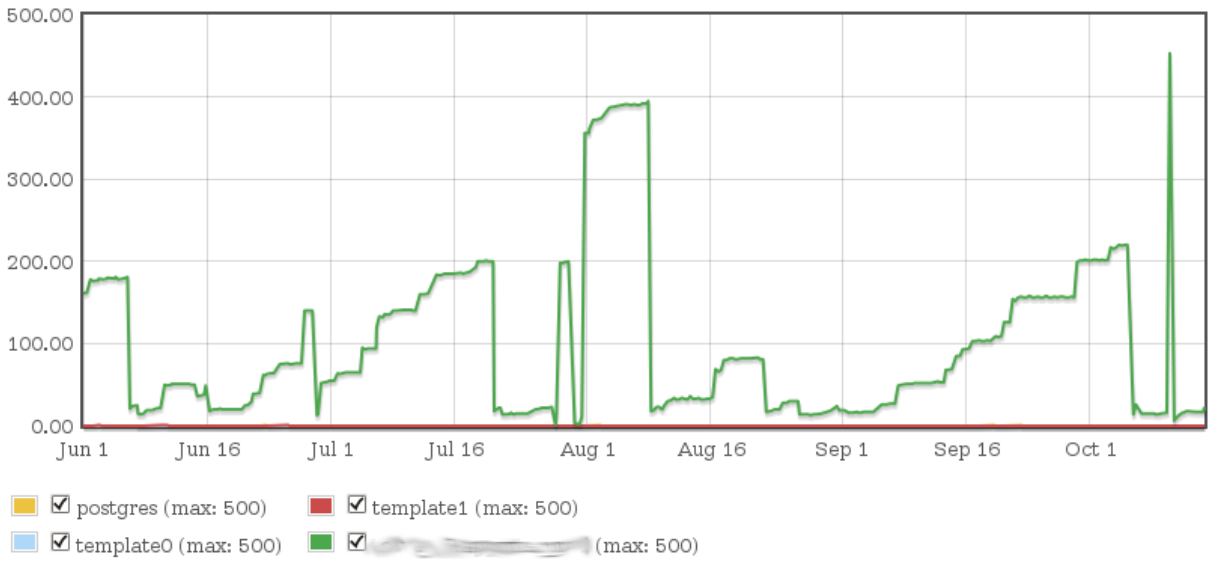
- l'activité
- l'instance
- les bases
- les tables
- les index
- les fonctions

Au niveau des statistiques, il existe un grand nombre d'informations intéressantes à récupérer. Cela va des informations sur l'activité en cours (nombre de connexions, noms des utilisateurs connectés, requêtes en cours d'exécution), à celles sur les bases de données (nombre d'écritures, nombre de lectures dans le cache), à celles sur les tables, index et fonctions.

Les connaître toutes présente peu d'intérêt car seulement certaines sont vraiment intéressantes à superviser. Nous allons voir ici quelques exemples.

8.6.6 Nombre de connexions par base

```
SELECT datname, numbackends FROM pg_stat_database;
SELECT datname, count(*) FROM pg_stat_activity
  WHERE datname IS NOT NULL
  GROUP BY datname;
```



Il est souvent intéressant de connaître le nombre de personnes connectées. Cela permet notamment de s'assurer que la configuration du paramètre `max_connections` est suffisamment élevée pour ne pas voir des demandes de connexion être refusées.

Il est aussi intéressant de pouvoir dénombrer le nombre de connexions par bases. Cela permet d'avoir une idée de la charge sur chaque base. Une base ayant de plus en plus d'utilisateurs et souffrant de performances devra peut-être être placée seule sur un serveur.

Il est aussi possible d'avoir le nombre de connexions par :

- utilisateur :

```
SELECT usename, count(*) FROM pg_stat_activity WHERE usename IS NOT NULL GROUP BY
↪  usename;
```

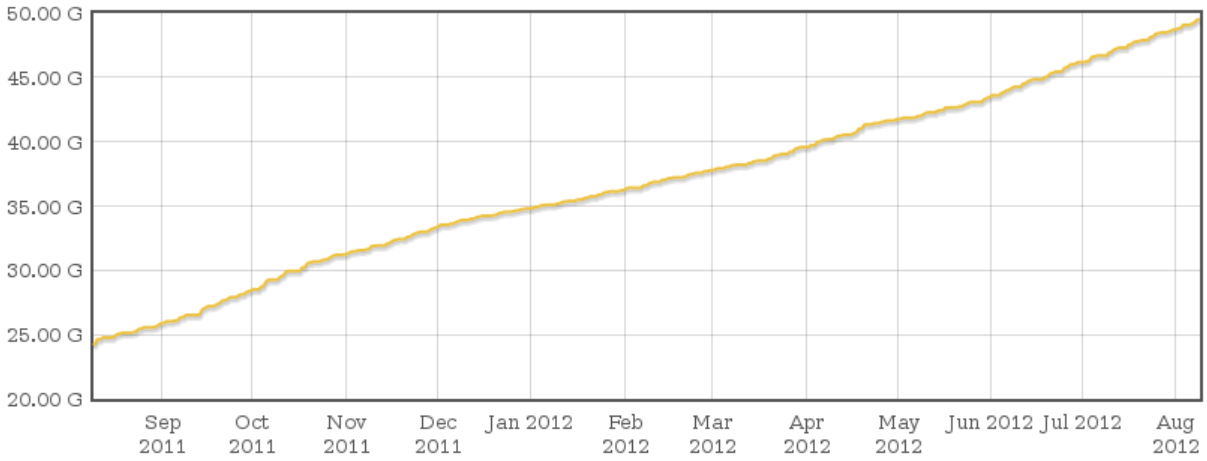
- application cliente :

```
SELECT application_name, count(*) FROM pg_stat_activity GROUP BY application_name;
```

Le graphe affiché ci-dessus montre l'utilisation des connexions sur différentes bases. Les bases systèmes ne sont pas du tout utilisées. Seule la base utilisateur reçoit un grand nombre de connexions. Il y a même eu deux pics, un à environ 400 connexions et un autre à 450 connexions.

8.6.7 Taille des bases

```
SELECT datname, pg_database_size(oid) FROM pg_database;
```

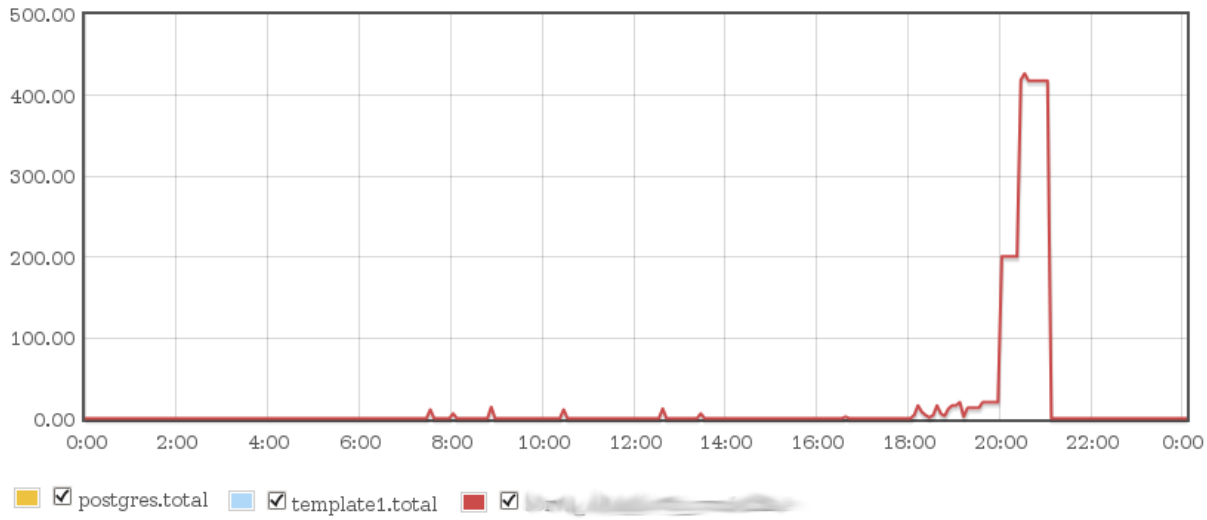


La volumétrie des bases est une autre information fréquemment demandée. L'exécution de cette requête toutes les cinq minutes permettra de suivre l'évolution de la taille des bases.

Le graphe ci-dessus montre une montée en volumétrie assez importante, la base ayant doublé en un an.

8.6.8 Nombre de verrous

```
SELECT d.datname, count(*) FROM pg_locks l
JOIN pg_database d ON l.database=d.oid
GROUP BY d.datname ORDER BY d.datname;
```



Autre demande fréquente : pouvoir suivre le nombre de verrous. La requête ci-dessus récupère le nombre de verrous posés par base. Il est aussi possible de récupérer les types de verrous, ou de ne prendre en compte que certains types de verrous.

Le graphe montre une utilisation très limitée des verrous. En fait, on observe surtout une grosse utilisation des verrous entre 20h et 21h30. Si le graphe montrait plusieurs jours d'affilé, on pourrait s'apercevoir que le pic est présent tous les jours à ce moment-là. En fait, il s'agit de la sauvegarde. La sauvegarde pose un grand nombre de verrous, des verrous très légers qui vont bloquer très peu de monde, mais néanmoins, ils sont présents.

8.6.9 Et un grand nombre d'autres informations

- Ratio de lecture du cache (souvent appelé *hit ratio*)
- Retard de réplication
- Nombre de transactions par seconde

Les trois exemples proposés ci-dessus ne sont que les exemples les plus marquants. Beaucoup d'autres informations sont récupérables. On peut citer par exemple :

- le ratio de lecture dans le cache de PostgreSQL ;
- le retard de la réplication interne de PostgreSQL (envoi, écriture, application) ;
- le nombre de transactions par seconde ;
- l'activité d'une table (en nombre de lectures, insertions, suppressions, modifications) ;
- le nombre de parcours séquentiels et de parcours d'index ;
- etc.

8.6.10 Outils

- Beaucoup d'outils existent pour exploiter les statistiques :
  - Munin, Nagios, Zabbix + sondes PG
- Dédiés à PostgreSQL :
  - pg_stat_statements[11]
  - PoWA[12]

Il existe de nombreux outils utilisables avec les statistiques. Les plus connus sont Munin (sondes déjà intégrées), Nagios et Zabbix. Pour ces deux derniers, il est nécessaire d'ajouter des sondes comme `check_postgres` et `check_pgactivity` évoquées plus haut.

`pg_stat_statements` est un module contrib de PostgreSQL, donc non installé par défaut. Il collecte des statistiques sur toutes les requêtes exécutées. Son contenu est souvent extrêmement instructif. Pour plus de détails sur les métriques relevées, voir https://dali.bo/h2_html#pg_stat_statements, et pour l'installation et des exemples de requêtes, voir https://dali.bo/x2_html#pg_stat_statements, ou encore la documentation officielle[13].

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire historisant les informations collectées par `pg_stat_statements`, et fournissant une interface graphique permettant d'observer en temps réel les requêtes normalisées les plus consommatrices d'une instance selon plusieurs critères.

8.6.11 munin

- Scripts Perl
- Sondes PostgreSQL incluses
- Récupère les statistiques toutes les 5 min
- Crée des pages HTML statiques et des fichiers PNG
    - donc des graphes

Munin est à la base un outil de métrologie utilisé par les administrateurs systèmes. Il comprend un certain nombre de sondes capables de récupérer des informations telles que la charge système, l'utilisation de la mémoire et des interfaces réseau, l'espace libre d'un disque, etc. Des sondes lui ont été ajoutées pour pouvoir surveiller certains services comme Sendmail, Postfix, Apache, et même PostgreSQL.

Munin est composé de deux parties : les sondes et le générateur de rapports. Les sondes sont exécutées toutes les cinq minutes. Elles sont généralement écrites en Perl. Elles récupèrent les informations et les stockent dans des bases BerkeleyDB. Ensuite, le générateur de rapports lit les bases en question pour générer des graphes au format PNG. Des pages HTML sont créées pour faciliter l'accès aux graphes.

Munin est inclus dans les distributions habituelles.

[13] https://docs.postgresql.fr/current/pgstatstatements.html

8.6.12 Nagios

- Outil GPL, sur https://www.nagios.org/
- Nombreux concurrents et équivalents
- Sondes dédiées à PostgreSQL : `check_postgres` et `check_pgactivity`

Nagios est un outil très connu de supervision. Il dispose par défaut de quelques sondes, principalement système. Il existe aussi des outils concurrents (Naemon, Icinga 2…) ou des surcouches compatibles au niveau des sondes.

Pour PostgreSQL, il est possible de le coupler à des sondes dédiées, comme `check_postgres` ou `check_pgactivity`, déjà évoquées, pour qu'il puisse récupérer un certain nombre d'informations statistiques de PostgreSQL. Ne pas oublier de compléter par des sondes plus classiques pour les I/O, le CPU, la RAM, etc.

8.6.13 Outils - Zabbix

- Outil GPL, sur https://www.zabbix.com/
- Sonde `check_postgres.pl`
- Template pg-monz
  - https://pg-monz.github.io/pg_monz/index-en.html

Zabbix est certainement le deuxième outil opensource le plus utilisé pour la supervision. Son avantage par rapport à Nagios est qu'il est capable de faire des graphes directement et qu'il dispose d'une interface plus simple d'approche.

Là-aussi, la sonde `check_postgres.pl` lui permet de récupérer des informations sur un serveur PostgreSQL.

8.6.14 Outils - pg_stat_statements

- Module contrib de PostgreSQL
- Récupère et stocke des statistiques d'exécution des requêtes
- Les requêtes sont normalisées
- Pas d'historisation

`pg_stat_statements` est une extension issue des « contrib » de PostgreSQL, donc livrée avec lui, mais non installée par défaut. Sa mise en place nécessite le préchargement de bibliothèques dans la mémoire partagée (paramètre `shared_preload_libraries`), et donc le redémarrage de l'instance.

Une fois installé et configuré, des mesures (nombre de blocs lus dans le cache, hors cache, …) sont collectées sur toutes les requêtes exécutées, et elles sont stockées avec les requêtes normalisées. Ces données sont ensuite exploitables en interrogeant la vue `pg_stat_statements`. À noter que ces statistiques sont cumulées sans être historisées, il est donc souvent difficile d'identifier quelle requête est la plus consommatrice à un instant donné, à moins de réinitialiser les statistiques.

Voir aussi la documentation officielle : https://docs.postgresql.fr/current/pgstatstatements.html

8.6.15 Outils - PoWA

- Site officiel : https://github.com/powa-team
- Licence : PostgreSQL
- Surveillance de l'activité SQL
- Captures des statistiques collectées par `pg_stat_statements`
  - et d'autres extensions
- Interface graphique : activité des requêtes en temps réel
- Dépôt GitHub
  - archiveur : https://github.com/powa-team/powa-archivist
  - UI web : https://github.com/powa-team/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire, sous licence PostgreSQL.

Tout comme pour l'extension standard `pg_stat_statements`, sa mise en place nécessite la modification du paramètre `shared_preload_libraries`, et donc le redémarrage de l'instance. Il faut également créer une nouvelle base de données dans l'instance. Par ailleurs, PoWA repose sur les statistiques collectées par `pg_stat_statements`, celui-ci doit donc être également installé.

Une fois installé et configuré, l'outil va récupérer à intervalle régulier les statistiques collectées par `pg_stat_statements`, les stocker et les historiser.

Il tire aussi partie d'autres extensions comme HypoPG, pg_qualstats, pg_stat_kcache…

L'outil fournit également une interface graphique permettant d'exploiter ces données, et donc d'observer en temps réel l'activité de l'instance. Cette activité est présentée sous forme de graphiques interactifs et de tableaux permettant de trier selon divers critères (nombre d'exécution, blocs lus hors cache…) les différentes requêtes normalisées sur l'intervalle de temps sélectionné.

8.7 CONCLUSION

- Un système est pérenne s'il est bien supervisé
- Supervision automatique
  - configuration des traces
  - configuration des statistiques
  - mise en place d'outils d'historisation

Une bonne politique de supervision est la clef de voûte d'un système pérenne. Pour cela, il faut tout d'abord s'assurer que les traces et les statistiques soient bien configurées. Ensuite, l'installation d'un outil d'historisation, de création de graphes et de génération d'alertes, est obligatoire pour pouvoir tirer profit des informations fournies par PostgreSQL.

8.7.1 Questions

N'hésitez pas, c'est le moment !

8.8 QUIZ

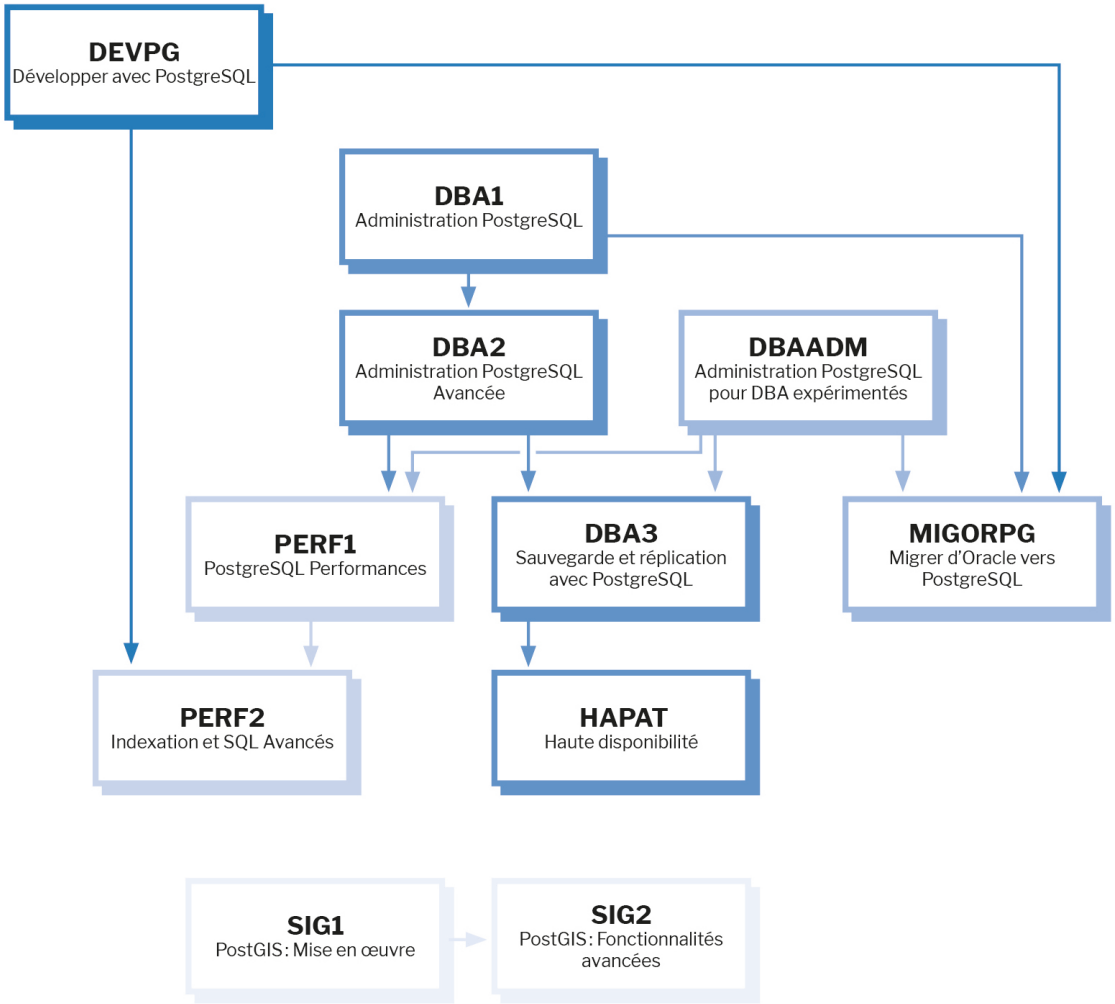
https://dali.bo/h1_quiz

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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