Formation DBA1

PostgreSQL Administration

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Administration

Formation DBA1

TITRE : PostgreSQL Administration
SOUS-TITRE : Formation DBA1

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-00-5

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, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé :

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

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

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

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

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

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

Note: Ceci est un résumé de la licence.

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

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

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

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

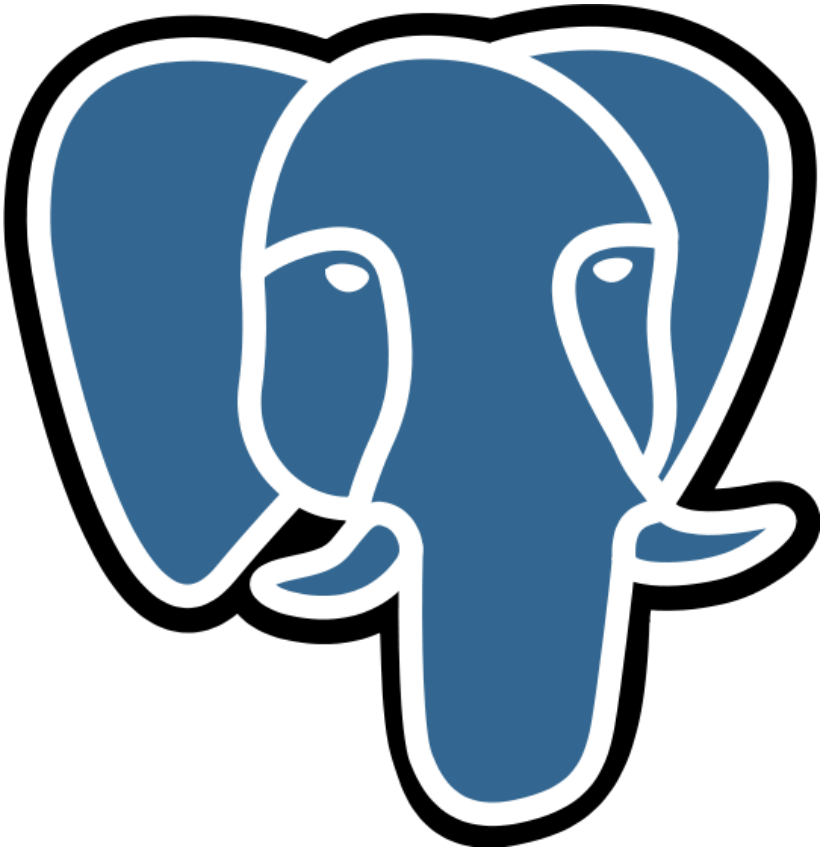
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 DÉCOUVRIR POSTGRESQL



2.1 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 utilisé 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.

2.1.1 AU MENU

1. Origines et historique du projet
2. Versions et feuille de route
3. Concepts de base
4. Fonctionnalités
5. Sponsors et références

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

Les deux premières parties expliquent la genèse du projet et détaillent les différences entres les versions successives du logiciel. Puis nous ferons un rappel théorique sur les principes fondateurs de PostgreSQL (ACID, MVCC, transactions, journaux de transactions) ainsi que sur les fonctionnalités essentielles (schémas, index, tablespaces, triggers).

Nous terminerons par un panorama d'utilisateurs renommés et de cas d' utilisations remarquables.

2.1.2 OBJECTIFS

- Comprendre les origines du projet

- Revoir les principes fondamentaux
- Découvrir des exemples concrets

PostgreSQL est un des plus vieux logiciels Open-Source ! Comprendre son histoire permet de mieux réaliser le chemin parcouru et les raisons de son succès. Par ailleurs, un rappel des concepts de base permet d'avancer plus facilement lors des modules suivants. Enfin, une série de cas d'utilisation et de références sont toujours utiles pour faire le choix de PostgreSQL en ayant des repères concrets.

2.2 UN PEU D'HISTOIRE...

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

2.2.1 LICENCE

- Licence PostgreSQL
  - BSD / MIT
  - http://www.postgresql.org/about/licence/
- Droit de
  - utiliser, copier, modifier, distribuer sans coût de licence
- Reconnu par l'Open Source Initiative
  - http://opensource.org/licenses/PostgreSQL
- Utilisé 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.

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

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

2.2.2 POSTGRESQL ?!?!

- Michael Stonebraker recode Ingres
- post « ingres » => post~~in~~gres => postgres
- postgres => PostgreSQL

L'origine du nom PostgreSQL remonte à la base de données Ingres, développée 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 » (prononcez « post - gresse - Q - L »). Cependant, le nom « Postgres » est accepté comme alias.

Pour aller plus loin :

- Fil de discussion sur les listes de discussion[1]
- Article sur le wiki officiel[2]

2.2.3 PRINCIPES FONDATEURS

- Sécurité des données (ACID)
- Respect des normes (ISO SQL)
- Fonctionnalités
- Performances
- Simplicité du code

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

Ceci explique en partie la réputation de relative lenteur et de complexité face aux autres SGBD du marché. Cette image est désormais totalement obsolète, notamment grâce aux avancées réalisées depuis les versions 8.x.

[1] http://archives.postgresql.org/pgsql-advocacy/2007-11/msg00109.php
[2] http://wiki.postgresql.org/wiki/Postgres

2.2.4 ORIGINES

- Années 1970 : **Ingres** est développé à Berkeley
- 1985 : **Postgres** succède à Ingres
- 1995 : Ajout du langage **SQL**.
- 1996 : Postgres devient **PostgreSQL**
- 1996 : Création du **PostgreSQL Global Development Group**

L'histoire de PostgreSQL remonte à la base de données Ingres, développée à Berkeley par Michael Stonebraker. Lorsque ce dernier décida en 1985 de recommencer le développement de zéro, il nomma le logiciel Postgres, comme raccourci de post-Ingres. Lors de l'ajout des fonctionnalités SQL en 1995 par deux étudiants chinois de Berkeley, Postgres fut renommé Postgres95. Ce nom fut changé à la fin de 1996 en PostgreSQL lors de la libération du code source de PostgreSQL.

De longs débats enflammés animent toujours la communauté pour savoir s'il faut revenir au nom initial Postgres.

À l'heure actuelle, le nom Postgres est accepté comme un alias du nom officiel PostgreSQL.

Plus d'informations :

- Page associée sur le site officiel[3]

2.2.5 ORIGINES (ANNÉES 2000)

Apparitions de la communauté internationale

- ~ 2000: Communauté japonaise
- 2004 : Communauté francophone
- 2006 : SPI
- 2007 : Communauté italienne
- 2008 : PostgreSQL Europe et US
- 2009 : Boom des PGDay

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.

[3] http://www.postgresql.org/about/history

En 2000 apparaît la communauté japonaise. Elle dispose d'un grand groupe, capable de réaliser des conférences chaque année. Elle compte au dernier recensement connu, plus de 3000 membres.

En 2004 nait l'association française (loi 1901) appelée PostgreSQLfr. Cette association a pour but de fournir un cadre légal pour pouvoir participer à certains événements comme Solutions Linux, les RMLL ou le pgDay 2008 à Toulouse. Elle permet aussi de récolter des fonds pour aider à la promotion de PostgreSQL.

En 2006, le PGDG intègre le « Software in the Public Interest », Inc. (SPI), une organisation à but non lucratif chargée de collecter et redistribuer des financements. Ce n'est pas une organisation spécifique à PostgreSQL. Elle a été créée à l'initiative de Debian et dispose aussi de membres comme OpenOffice.org.

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. PGUS apparaît pour consolider les différents groupes américains d'utilisateurs PostgreSQL. Ces derniers étaient plutôt créés géographiquement, par état ou grosse ville. Ils peuvent rejoindre et être aidés par cette organisation. De même en Europe arrive PostgreSQL Europe, une 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. Cet événement a eu lieu la première fois en France (sous le nom pgday.eu) à Paris, en 2009, puis en Allemagne à Stuttgart en 2010, en 2011 à Amsterdam, à Prague en 2012, à Dublin en 2013 et à Madrid en 2014. Cependant, elle aide aussi les communautés allemandes et suédoise à monter leur propre événément (respectivement pgconf.de et nordic pgday).

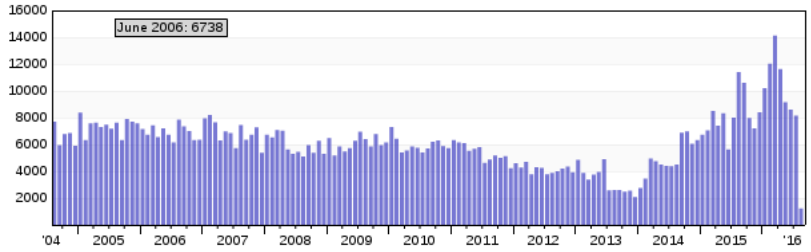
En 2010, on dénombre plus d'une conférence par mois consacrée uniquement à PostgreSQL dans le monde.

- Communauté japonaise[4]
- Communauté francophone[5]
- Communauté italienne[6]
- Communauté européenne[7]
- Communauté US[8]

[4] http://www.postgresql.jp
[5] http://www.postgresql.fr
[6] http://www.itpug.org
[7] http://www.postgresql.eu
[8] http://www.postgresql.us

17.12

2.2.6 PROGRESSION DU PROJET - ÉCHANGES DE MAIL



Ce graphe représente l'évolution du trafic des listes de diffusion du projet qui est corollaire du nombre d'utilisateurs du logiciel.

On remarque une augmentation très importante jusqu'en 2005 puis une petite chute en 2008, un peu récupérée en 2009, un nouveau creux fin 2012 jusqu'en début 2013, un trafic stable autour de 4500 mails par mois en 2014 et depuis une progression constante pour arriver en 2016 à des pics de 12000 mails par mois.

La moyenne actuelle est d'environ 250 messages par jour sur les 23 listes actives.

Source du graphe[9]

On peut voir l'importance de ces chiffres en comparant le trafic des listes PostgreSQL et MySQL (datant de février 2008) sur ce lien[10] .

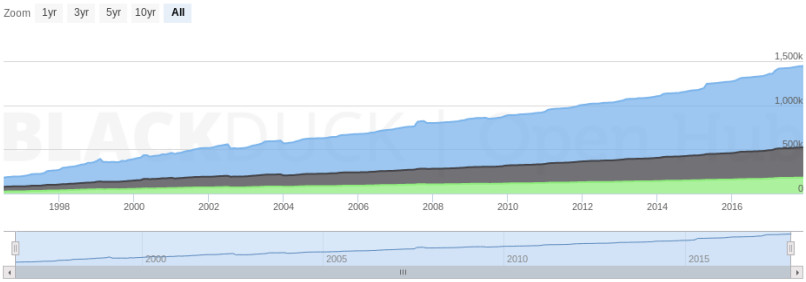
Début 2014, il y a moins de 1 message par jour sur les listes MySQL tel que mesuré sur Markmail.

Pour aller plus loin : On peut également visualiser l'évolution des contributions de la communauté PostgreSQL grâce au projet Code Swarm[11] .

---

[9] http://postgresql.markmail.org/
[10] http://markmail.blogspot.com/2008/02/postgresql-more-traffic-than-mysql-and.html
[11] http://www.vimeo.com/1081680

2.2.7 PROGRESSION DU CODE



Ce graphe représente l'évolution du nombre de lignes de code dans les sources de PostgreSQL. Cela permet de bien visualiser l'évolution du projet en terme de développement.

On note une augmentation constante depuis 2000 avec une croissance régulière d 'environ 25000 lignes de code C par an. Le plus intéressant est certainement de noter que l'évolution est constante.

Actuellement, PostgreSQL est composé de plus de 1.000.000 de lignes (dont 270.000 lignes de commentaires), pour environ 200 développeurs actifs.

Source[12] .

2.3 LES VERSIONS

- Versions obsolètes : 9.2 et antérieures
- Versions actuelles : de 9.3 à 10
- Version en cours de développement : 11
- Versions dérivées

2.3.1 HISTORIQUE

- 1996 : v1.0 -> première version publiée
- 2003 : v7.4 -> première version *réellement* stable
- 2005 : v8.0 -> arrivée sur Windows

[12] https://www.openhub.net/p/postgres

17.12

- 2008 : v8.3 -> performance et fonctionnalités
- 2010 : v9.0 -> réplication intégrée
- 2016 : v9.6 -> parallélisation
- 2017 : v10 -> réplication logique

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 procédures stockées en Java, le Point In Time Recovery, la réplication asynchrone 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 : gestion des commit fests, création de l 'outil web commitfest, 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 ajoute quelques améliorations supplémentaires. La version 9.4 apporte également un certain nombre d'améliorations, ainsi que 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, mais surtout l'ajout de la réplication logique.

Il est toujours possible de télécharger les sources depuis la version 1.0 jusqu'à la version courante sur postgresql.org[13] .

2.3.2 NUMÉROTATION

- Avant la version 10
  - X.Y : version majeure (8.4, 9.6)
  - X.Y.Z : version mineure (9.6.4)
- Après la version 10
  - X : version majeure (10, 11)

[13] http://www.postgresql.org/ftp/source/

  - X.Y : version mineure (10.1)

Une version majeure apporte de nouvelles fonctionnalités, des changements de comportement, etc. Une version majeure sort généralement tous les 12/15 mois.

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Elles sont plus fréquentes que les versions majeures, avec un rythme de sortie de l'ordre des trois mois, sauf bugs majeurs ou failles de sécurité. Chaque bug est corrigé dans toutes les versions stables actuellement maintenues par le projet.

2.3.3 VERSIONS COURANTES

- Dernières releases (9 novembre 2017) :
  - version 9.3.20
  - version 9.4.15
  - version 9.5.10
  - version 9.6.6
  - version 10.1
- Prochaine sortie, 8 février 2018

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, depuis quelques années, les choses évoluent et la tendance actuelle est de livrer une version stable majeure tous les 12 à 15 mois, tout en conservant la qualité des versions. De ce fait, toute fonctionnalité supposée pas suffisamment stable est repoussée à la version suivante.

Le support de la version 7.3 a été arrêté au début de l'année 2008. La même chose est arrivée aux versions 7.4 et 8.0 milieu 2010, à la 8.1 en décembre 2010, à la 8.2 en décembre 2011, à la 8.3 en février 2013, à la 8.4 en juillet 2014 et la 9.0 en septembre 2015, la 9.1 en septembre 2016. La prochaine version qui subira ce sort est la 9.2, en septembre 2017.

La tendance actuelle est de garantir un support pour chaque version courante pendant une durée minimale de 5 ans.

Pour plus de détails : Politique de versionnement[14] .

---

[14] http://www.postgresql.org/support/versioning/

17.12

2.3.4 VERSION 8.4

- Juillet 2009 - juillet 2014 (cette version n'est plus maintenue)
- Fonctions Window (clause `OVER`)
- CTE (vues non persistantes) et requêtes récursives
- Infrastructure SQL/MED (données externes)
- Paramètres par défaut et nombre variant de paramètres pour les fonctions
- Restauration parallélisée d'une sauvegarde
- Droits sur les colonnes

De plus, cette version apporte des améliorations moins visibles telles que :

- locale configurable par base de données ;
- refonte du FSM ;
- VACUUM sélectif grâce au « visibility map » ;
- support des certificats SSL ;
- statistiques sur les fonctions ;
- fonction `pg_terminate_backend()` ;
- nouveaux modules contrib : `pg_stat_statements`, `auto_explain`, ...

Exemple de la volonté d'intégrer des fonctionnalités totalement matures, le Hot Standby, fonctionnalité très attendue par les utilisateurs, a finalement été repoussé pour la version suivante car les développeurs estimaient qu' elle n'était pas assez fiable.

Pour plus de détails :

- Release notes[15]
- Article GLMF[16]

Cette version n'est plus maintenue depuis juillet 2014.

2.3.5 VERSION 9.0

- Septembre 2010 - septembre 2015
- Hot Standby + Streaming Replication
- Contraintes d'exclusion
- Améliorations pour l'`EXPLAIN`
- Contrainte `UNIQUE` différable
- Droits par défaut, `GRANT ALL`

[15] http://www.postgresql.org/docs/current/interactive/release-8-4.html
[16] http://www.dalibo.org/hs44_postgresql_8.4

- Triggers sur colonne, et clause `WHEN`

Mais aussi :

- droit d'accès aux « Large Objects » ;
- configurations par utilisateurs, par bases de données mais aussi par couple utilisateur/base ;
- bloc de code anonyme.

Pour plus de détails :

- Traduction annonce 9.0[17]
- Présentation sur la version 9.0[18]
- Article GLMF sur la réplication, partie 1[19]
- Article GLMF sur la réplication, partie 2[20]
- Article GLMF sur les autres nouveautés de la versino 9.0[21]

L'arrêt du support de cette version surviendra en septembre 2015.

2.3.6 VERSION 9.1

- Septembre 2011 - septembre 2016
- Réplication synchrone
- Supervision et administration plus aisée de la réplication
- Gestion des extensions
- Support des tables distantes via SQL/MED
- Support des labels de sécurité
- Support des tables non journalisées

Et beaucoup d'autres :

- moins de verrous pour les `DDL` (`ALTER TABLE`, `TRIGGER`) ;
- réduction de la taille des champs de type `NUMERIC` sur disque ;
- compteurs du nombre de `VACUUM` et `ANALYZE` pour les tables `pg_stat_*_tables` ;
- le support des triggers sur les vues.

Pour plus de détails :

[17] http://www.dalibo.org/annonce_9.0
[18] http://www.dalibo.org/quoi_de_neuf_dans_postgresql_9
[19] http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_1
[20] http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_2
[21] http://www.dalibo.org/glmf134_postgresql_les_autres_nouveautes

17.12

- Page officielle des nouveautés de la version 9.1[22]
- Article GLMF sur la version 9.1, partie 1[23]
- Article GLMF sur la version 9.1, partie 2[24]

2.3.7 VERSION 9.2

- Septembre 2012 - septembre 2017
- Réplication en cascade
  - `pg_basebackup` utilisable sur un esclave
  - réplication synchrone en mémoire seulement sur l'esclave
- Axé performances
  - amélioration de la scalabilité (lecture et écriture)
  - parcours d'index seuls
- Support de la méthode d'accès SP-GiST pour les index
- Support des types d'intervalles de valeurs
- Support du type de données JSON

Et beaucoup d'autres :

- amélioration des `EXPLAIN` ;
- nouveau processus `checkpointer` ;
- utilisation d'algorithmes spécialisés plus rapides pour le tri ;
- nouveau paramètre pour limiter la taille des fichiers temporaires ;
- nouvel outil `pg_receivexlog` (à présent `pg_receivewal`) ;
- annulation de ses propres requêtes avec `pg_cancel_backend()` par un utilisateur de base ;
- possibilité de déplacer plus facilement un tablespace, moteur éteint ;
- plus de traces pour l'activité disque d'autovacuum ;
- nouveaux champs dans les vues statistiques `pg_stat_activity`, `pg_stat_database` et `pg_stat_bgwriter` ;
- ajout des vues avec « barrière de sécurité » ;
- ajout des fonctions `LEAKPROOF` ;
- support des labels de sécurité sur les objets partagés.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.2[25]

[22] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.1/fr
[23] http://www.dalibo.org/glmf145_nouveautes_de_postgresql_9.1_partie_1
[24] http://www.dalibo.org/glmf145_nouveautes_de_postgresql_9.1_partie_2
[25] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.2

- Workshop Dalibo sur la version 9.2[26]

2.3.8 VERSION 9.3

- Septembre 2013 - septembre 2018 (?)
- Meilleure gestion de la mémoire partagée
- Support de la clause `LATERAL` dans un `SELECT`
- 4 To maxi au lieu de 2 Go pour les Large Objects
- `COPY FREEZE`
- Vues en mise à jour
- Vues matérialisées

Et d'autres encore :

- le FDW PostgreSQL (`postgres_fdw`) ;
- failover rapide ;
- configuration du `recovery.conf` par pg_basebackup ;
- `pg_dump` parallélisé ;
- background workers ;
- etc.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.3[27]
- Workshop Dalibo sur la version 9.3[28]

2.3.9 VERSION 9.4

- décembre 2014 - décembre 2019 (?)
- amélioration des index GIN (taille réduite et meilleures performances)
- nouveau type JSONB
- rafraîchissement sans verrou des vues matérialisées
- possibilité pour une instance répliquée de mémoriser la position des instances secondaires (*replication slots*)
- décodage logique (première briques pour la réplication logique intégrée)

Et beaucoup d'autres :

[26] https://kb.dalibo.com/conferences/postgresql_9.2/presentation
[27] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.3
[28] https://kb.dalibo.com/conferences/workshop_9.3/presentation

17.12

- clause `WITH CHECK OPTION` pour les vues automatiquement modifiables ;
- clauses `FILTER` et `WITHIN GROUP` pour les agrégats ;
- commande `SQL ALTER SYSTEM` pour modifier `postgresql.conf` ;
- améliorations des fonctions d'agrégat ;
- nouvelle contrib : `pg_prewarm` ;
- nouvelles options de formatage pour `log_line_prefix` ;
- background workers dynamiques...

Il y a eu des soucis détectés pendant la phase de bêta sur la partie JSONB. La correction de ceux-ci a nécessité de repousser la sortie de cette version de trois mois le temps d'effectuer les tests nécessaires. La version stable est sortie le 18 décembre 2014.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.4[29]
- Workshop Dalibo sur la version 9.4[30]

2.3.10 VERSION 9.5

- Janvier 2016 - Janvier 2021 (?)
- Row Level Security
- Index `BRIN`
- `INSERT ... ON CONFLICT { UPDATE | IGNORE }`
- `SKIP LOCKED`
- `SQL/MED`
  - import de schéma, héritage
- Supervision
  - amélioration de `pg_stat_statements`, ajout de `pg_stat_ssl`
- fonctions OLAP (`GROUPING SETS`, `CUBE` et `ROLLUP`)

Pour plus de détails :

- Page officielle des nouveautés de la version 9.5[31]
- Workshop Dalibo sur la version 9.5[32]

---

[29] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.4
[30] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.4
[31] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.5
[32] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.5

2.3.11 VERSION 9.6

- Septembre 2016 - Septembre 2021 (?)
- Parallélisation
  - parcours séquentiel, jointure, agrégation
- SQL/MED
  - tri distant, jointures impliquant deux tables distantes
- Réplication synchrone
- MVCC
  - `VACUUM FREEZE`, `CHECKPOINT`, ancien snapshot
- Maintenance

Le développement de cette version a commencé en mai 2015. La première bêta est sortie en mai 2016. La version stable est sortie le 29 septembre 2016.

La fonctionnalité majeure sera certainement l'intégration du parallélisme de certaines parties de l'exécution d'une requête.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.6[33]
- Workshop Dalibo sur la version 9.6[34]

2.3.12 VERSION 10

- Septembre 2017 - Septembre 2022 (?)
- Meilleure parallélisation
  - parcours d'index, jointure MergeJoin, sous-requêtes corrélées
- Réplication logique
- Partitionnement

La fonctionnalité majeure est de loin l'intégration de la réplication logique. Cependant d'autres améliorations devraient attirer les utilisateurs comme celles concernant le partitionnement, les tables de transition ou encore les améliorations sur la parallélisation.

Pour plus de détails : * Page officielle des nouveautés de la version 10[35] * Workshop Dalibo sur la version 10[36]

---

[33] https://wiki.postgresql.org/wiki/NewIn96
[34] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.6
[35] https://wiki.postgresql.org/wiki/New_in_postgres_10
[36] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_10

17.12

2.3.13 PETIT RÉSUMÉ

- Versions 7
  - fondations
  - durabilité
- Versions 8
  - fonctionnalités
  - performances
- Versions 9
  - réplication physique
  - extensibilité
- Versions 10
  - réplication logique
  - parallélisation

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 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).

2.3.14 QUELLE VERSION UTILISER ?

- 9.2 et inférieures
  - **Danger !**
- 9.3
  - planifier une migration rapidement
- 9.4, 9.5 et 9.6
  - mise à jour uniquement
- 10
  - nouvelles installations et nouveaux développements

Si vous avez une version 9.2 ou inférieure, planifiez le plus rapidement possible une migration vers une version plus récente, comme la 9.3 ou la 9.4.

La 9.2 n'est plus maintenue à compter de septembre 2017. Si vous utilisez cette version, il serait bon de commencer à étudier une migration de version dès que possible.

Pour les versions 9.3, 9.4, 9.5 et 9.6, le plus important est d'appliquer les mises à jour correctives.

La version 10 est officiellement stable depuis septembre 2017. Cette version peut être utilisée pour les nouvelles installations en production et les nouveaux développements. Son support est assuré jusqu'en septembre 2022.

Tableau comparatif des versions[37] .

2.3.15 VERSIONS DÉRIVÉES / FORKS

- Compatibilité Oracle
  - EnterpriseDB Postgres Plus
- Data warehouse
  - Greenplum
  - Netezza
  - Amazon RedShift

Il existe de nombreuses versions dérivées de PostgreSQL. Elles sont en général destinées à des cas d'utilisation très spécifiques. Leur code est souvent fermé et nécessite l'acquisition d'une licence payante.

Liste exhaustive des « forks »[38] .

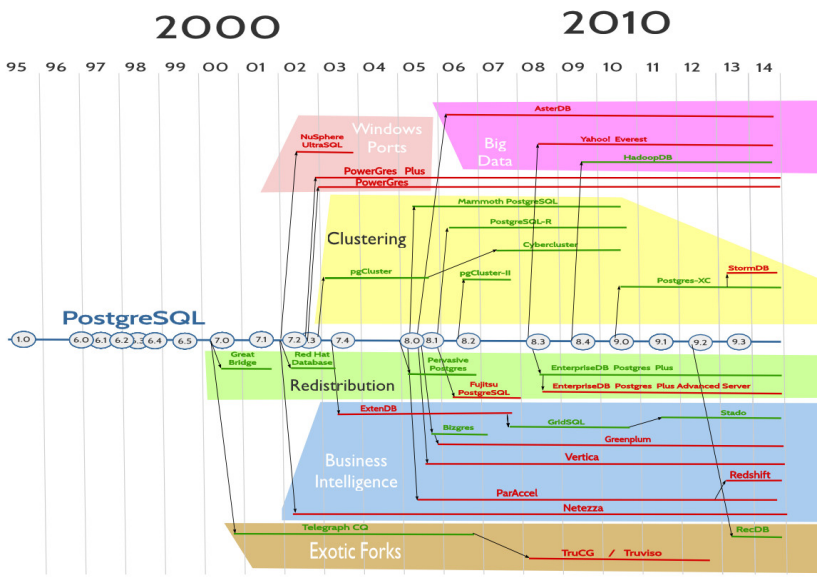
Sauf cas très précis, il est recommandé d'utiliser la version officielle, libre et gratuite.

---

[37] http://www.postgresql.org/about/featurematrix
[38] http://wiki.postgresql.org/wiki/PostgreSQL_derived_databases

17.12

2.3.16 HISTORIQUE DES VERSIONS DÉRIVÉES



Voici un schéma des différentes versions de PostgreSQL ainsi que des versions dérivées. Cela montre principalement l'arrivée annuelle d'une nouvelle version majeure, ainsi que de la faible résistance des versions dérivées. La majorité n'a pas survécu à la vitalité du développement de PostgreSQL.

2.4 CONCEPTS DE BASE

- ACID
- MVCC
- Transactions
- Journaux de transactions

2.4.1 ACID

- **Atomicité** (Atomic)

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

Les propriétés ACID sont le fondement même de tout système transactionnel. Il s'agit de quatre règles fondamentales :

- A : Une transaction est entière : « tout ou rien ».
- C : Une transaction amène le système d'un état stable à un autre.
- I : Les transactions n'agissent pas les unes sur les autres.
- D : Une transaction validée provoque des changements permanents.

Les propriétés ACID sont quatre propriétés essentielles d'un sous-système de traitement de transactions d'un système de gestion de base de données. Certains SGBD ne fournissent pas les garanties ACID. C'est le cas de la plupart des SGBD non-relationnels (« NoSQL »). Cependant, la plupart des applications ont besoin de telles garanties et la décision d'utiliser un système ne garantissant pas ces propriétés ne doit pas être prise à la légère.

2.4.2 MULTIVERSION CONCURRENCY CONTROL (MVCC)

- Le « noyau » de PostgreSQL
- Garantit ACID
- Permet les écritures concurrentes sur la même table

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.

C'est notamment MVCC qui permet de sauvegarder facilement une base *à chaud* et d'obtenir une sauvegarde cohérente 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, un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

2.4.3 MVCC ET LES VERROUS

- 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**.

MVCC maintient toutes les versions nécessaires de chaque tuple, 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.

Voici un exemple concret :

```
# SELECT now();
              now
-------------------------------
 2017-08-23 16:28:13.679663+02
(1 row)

# BEGIN;
BEGIN
# SELECT now();
             now
-------------------------------
 2017-08-23 16:28:34.888728+02
(1 row)

# SELECT pg_sleep(2);
 pg_sleep
----------

(1 row)

# SELECT now();
             now
-------------------------------
 2017-08-23 16:28:34.888728+02
(1 row)
```

2.4.4 TRANSACTIONS

- Intimement liées à ACID et MVCC :
- Une transaction est un ensemble d'opérations atomique
- Le résultat d'une transaction est « tout ou rien »
- SAVEPOINT disponible pour sauvegarde des modifications d'une transaction à un instant t

Voici un exemple de transaction:

```
=> BEGIN;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1, 'Haddock', 35);
INSERT 0 1
=> SELECT age FROM capitaines;
 age
-----
  35
(1 ligne)

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

On voit que la table capitaine 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.

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;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1,'Haddock',35);
INSERT 0 1
=> SAVEPOINT insert_sp;
SAVEPOINT
=> UPDATE capitaines SET age=45 WHERE nom='Haddock';
```

```
UPDATE 1
=> ROLLBACK TO SAVEPOINT insert_sp;
ROLLBACK
=> COMMIT;
COMMIT
=> SELECT age FROM capitaines WHERE nom='Haddock';
 age
-----
  35
(1 row)
```

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`.

2.4.5 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
- On peut spécifier le niveau d'isolation au démarrage d'une transaction:
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited`
  - `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.

2.4.6 WRITE AHEAD LOGS, AKA WAL

- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Comparable à la journalisation des systèmes de fichiers

Les journaux de transactions (appelés parfois WAL ou 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 à l'emplacement final 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'information[39] .

2.4.7 AVANTAGES DES WAL

- Un seul *sync* sur le fichier de transactions
- Le fichier de transactions est écrit de manière séquentielle
- Les fichiers de données sont écrits de façon asynchrone
- Point In Time Recovery
- Réplication (*WAL shipping*)

Les écritures se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture. Généralement, le déplacement des têtes d'un disque est l'opération la plus coûteuse. L'éviter est un énorme avantage.

De plus, comme on n'écrit que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, à condition que le système de fichiers le supporte.

L'écriture asynchrone dans les fichiers de données 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.

---

[39] http://www.dalibo.org/glmf108_postgresql_et_ses_journaux_de_transactions

2.5 FONCTIONNALITÉS

- Développement
- Sécurité
- Le « catalogue » d'objets SQL

Depuis toujours, PostgreSQL se distingue par sa licence libre (BSD) et sa robustesse prouvée sur de nombreuses années. Mais sa grande force réside également dans le grand nombre de fonctionnalités intégrées dans le moteur du SGBD :

- L'extensibilité, avec des API très bien documentées
- Une configuration fine et solide de la sécurité des accès
- Un support excellent du standard SQL

2.5.1 FONCTIONNALITÉS : DÉVELOPPEMENT

- PostgreSQL est une plate-forme de développement !
- 15 langages de procédures stockées
- Interfaces natives pour ODBC, JDBC, C, PHP, Perl, etc.
- API ouverte et documentée
- Un nouveau langage peut être ajouté **sans recompilation** de PostgreSQL

Voici la liste non exhaustive des langages procéduraux supportés :

- PL/pgSQL
- PL/Perl
- PL/Python
- PL/Tcl
- PL/sh
- PL/R
- PL/Java
- PL/lolcode
- PL/Scheme
- PL/PHP
- PL/Ruby
- PL/Lua
- PL/pgPSM
- PL/v8 (Javascript)

PostgreSQL peut donc être utilisé comme un serveur d'applications ! Vous pouvez ainsi placer votre code au plus près des données.

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 serait préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une procédure en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir. Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage.

Les applications externes peuvent accéder aux données du serveur PostgreSQL grâce à des connecteurs. Ils peuvent passer par l'interface native, la **libpq**. C'est le cas du connecteur PHP et du connecteur Perl par exemple. Ils peuvent aussi ré-implémenter cette interface, ce que fait le pilote ODBC (psqlODBC) ou le driver JDBC.

Tableau des langages supportés[40] .

2.5.2 FONCTIONNALITÉS : EXTENSIBILITÉ

Création de types de données et

- de leurs fonctions
- de leurs opérateurs
- de leurs règles
- de leurs aggrégats
- de leurs méthodes d'indexations

Il est possible de définir de nouveaux 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 va pouvoir être utilisé dans tous les objets SQL habituels : table, procédure stockée, opérateur (pour redéfinir l'opérateur + par exemple), procédure d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
```

[40] http://wiki.postgresql.org/wiki/PL_Matrix

```
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

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

Conférence de Heikki Linakangas sur la création d'un type color[41] .

2.5.3 SÉCURITÉ

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

Le support des annuaires LDAP est disponible à partir de la version 8.2.

Le support de GSSAPI/SSPI est disponible à partir de la version 8.3. L' interface de programmation GSS API est un standard de l'IETF qui permet de sécuriser les services informatiques. La principale implémentation de GSSAPI est Kerberos. SSPI permet le Single Sign On sous MS Windows, de façon transparente, qu'on soit dans un domaine AD ou NTLM.

La gestion des certificats SSL est disponible à partir de la version 8.4.

Le support de Radius est disponible à partir de la version 9.0.

Le support de `SCRAM-SHA-256` est disponible à partir de la version 10.

2.5.4 RESPECT DU STANDARD SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.
- Unicode et plus de 50 encodages

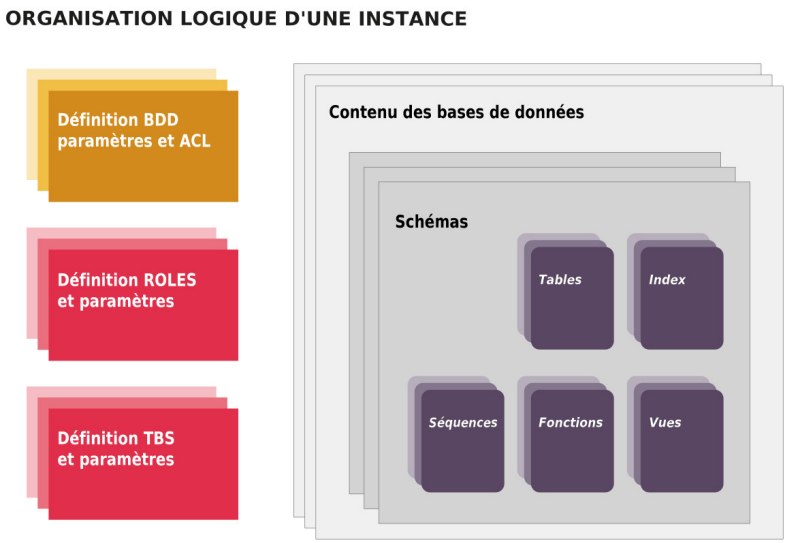
[41] http://wiki.postgresql.org/images/1/11/FOSDEM2011-Writing_a_User_defined_type.pdf

La dernière version du standard SQL est `SQL:2011`.

À ce jour, aucun SGBD ne supporte complètement `SQL:2011` *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de `SQL:2011` est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

2.5.5 ORGANISATION LOGIQUE



2.5.6 SCHÉMAS

- Espace de noms
- Concept différent des schémas d'Oracle
- Sous-ensemble de la base

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble. Tout dépend des droits dont il dispose. PostgreSQL vérifie la présence des objets par rapport au

paramètre `search_path` valable pour la session en cours lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête.

À la création d'un utilisateur, un schéma n'est pas forcément associé.

Le comportement et l'utilisation des schémas diffèrent donc d'avec Oracle.

Les schémas sont des espaces de noms dans une base de données permettant :

- de grouper 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.

Les schémas sont très utiles pour les systèmes de réplication (Slony, bucardo).

Exemple d'utilisation de schéma :

```
=> CREATE SCHEMA pirates;
CREATE SCHEMA
=> SET search_path TO pirates,public;
SET
=> CREATE TABLE capitaines (id serial, nom text);
CREATE TABLE

=> INSERT INTO capitaines (nom)
 VALUES ('Anne Bonny'), ('Francis Drake');
INSERT 0 2

-- Sélections des capitaines... pirates
=> SELECT * FROM capitaines;
 id |      nom
----+---------------
  1 | Anne Bonny
  2 | Francis Drake
(2 rows)

=> CREATE SCHEMA corsaires;
CREATE SCHEMA
=> SET search_path TO corsaires,pirates,public;
SET
=> CREATE TABLE capitaines (id serial, nom text);
CREATE TABLE
=> INSERT INTO corsaires.capitaines (nom)
 VALUES ('Robert Surcouf'), ('Francis Drake');
INSERT 0 2
-- Sélections des capitaines... français
=> SELECT * FROM capitaines;
```

```
 id |      nom
----+----------------
  1 | Robert Surcouf
  2 | Francis Drake
(2 rows)


-- Quels capitaine portant un nom identique
-- fut à la fois pirate et corsaire ?
=> SELECT pirates.capitaines.nom
  FROM pirates.capitaines,corsaires.capitaines
  WHERE pirates.capitaines.nom=corsaires.capitaines.nom;
      nom
---------------
 Francis Drake
(1 row)
```

2.5.7 VUES

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Améliorations en 9.3 et 9.4
  - vues matérialisées
  - vues automatiquement modifiables

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 en tant que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Exemple:

```
=# SET search_path TO public;
SET

-- 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;
CREATE ROLE
```

```
-- création de la table, et ajout de données
=# ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
ALTER TABLE
=# INSERT INTO capitaines (nom,age,num_cartecredit)
           VALUES ('Robert Surcouf',20,'1234567890123456');
INSERT 0 1

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

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

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

-- 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******
(1 ligne)

=> SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

À partir de la 8.4, 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 pouvait être fastidieux).

Par exemple :

```
=> SET ROLE postgres;
SET
=# 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;
CREATE VIEW
=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
      nom       | age |    num_cc_anon    |            num_md5_cc
```

```
----------------+-----+-----------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
(1 row)
```

La version 9.3 améliore encore les choses en permettant la modification des vues simples, sans ajout de code et de trigger. Ce n'est pas le cas ici étant donné que la vue retourne des colonnes calculées :

```
=# UPDATE capitaines_anon SET nom='Nicolas Surcouf' WHERE nom='Robert Surcouf';
ERROR:  cannot update view "capitaines_anon"
DETAIL:  Views that return columns that are not columns of their base relation
         are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an
       unconditional ON UPDATE DO INSTEAD rule.
```

À partir de la version 9.4, ce type de modification est possible, à condition que seules les colonnes identiques à la table sous-jacente soient mises à jour :

```
=# UPDATE capitaines_anon SET nom='Nicolas Surcouf' WHERE nom='Robert Surcouf';
UPDATE 1

=# SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';
       nom       | age |   num_cc_anon   |            num_md5_cc
-----------------+-----+-----------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
(1 row)

=# UPDATE capitaines_anon SET num_cc_anon='123456789xxxxxx'
  WHERE nom='Nicolas Surcouf';
ERREUR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

La version 9.3 ajoute aussi le support natif des vues matérialisées. Elles ne sont pas mises à jour automatiquement, il faut utiliser une instruction spécifique quand le rafraîchissement s'avère nécessaire. Ce rafraîchissement bloque les écritures et les lectures, et est un rafraîchissement complet de la vue matérialisée. Ces quelques inconvénients sont corrigés en 9.4, qui apporte la possibilité d'effectuer le rafraîchissement de la vue matérialisée de façon concurrente, c'est-à-dire sans bloquer les autres sessions désirant lire le contenu de la vue.

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

-- 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;
SELECT 2

-- 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******
(1 row)

-- 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';
UPDATE 1
=# SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';
 id |      nom        | age | num_cartecredit
----+-----------------+-----+------------------
  1 | Robert Surcouf |  20 | 1234567890123456
(1 row)

=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
       nom        | age |   num_cc_anon
------------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******
(1 row)

-- Après un rafraichissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
=# REFRESH MATERIALIZED VIEW capitaines_anon;
REFRESH MATERIALIZED VIEW
=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
      nom        | age |   num_cc_anon
-----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******
(1 row)

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions
-- ( >= 9.4 ) :
=# 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 unique pour faire un rafraichissement
```

```
-- sans bloquer les autres sessions.
=# CREATE UNIQUE INDEX ON capitaines_anon(nom);
CREATE INDEX
=# REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
REFRESH MATERIALIZED VIEW
```

Avant la version 9.3, il était possible de simuler une vue matérialisée via une table standard, une procédure stockée de type trigger et un trigger. Pour plus de détails, voir cet article de Jonathan Gardner[42] .

2.5.8 INDEX

Les algorithmes suivants sont supportés :

- `B-tree` (par défaut)
- `GiST` / `SP-GiST`
- `Hash`
- `GIN` (version `8.2`)
- `BRIN` (version `9.5`)

Attention aux index `hash` ! Avant la version 10, leur modification n'est pas enregistrée dans les journaux de transactions, ce qui amène deux problèmes. En cas de crash du serveur, il est fréquemment nécessaire de les reconstruire (`REINDEX`). De plus, ils ne sont pas restaurés avec PITR et donc avec le *Log Shipping* et le *Streaming Replication*. Par ailleurs, toujours avant la version 10, ils ne sont que rarement plus performants que les index B-Tree.

Pour une indexation standard , on utilise en général un index Btree.

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 peuvent être utiles pour les grands volumes de données fortement corrélées par rapport à leur emplacement physique sur les disques.

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

Plus d'informations :

- Article Wikipédia sur les arbres B[43]

[42] http://jonathangardner.net/PostgreSQL/materialized_views/matviews.html
[43] http://fr.wikipedia.org/wiki/Arbre_B

17.12

- Article Wikipédia sur les tables de hachage[44]
- Documentation officielle française[45]

2.5.9 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, au delà du type de données.

Elles donnent des informations au planificateur qui lui permettent d'optimiser les requêtes. Par exemple, le planificateur de la version 9.0 sait ne pas prendre en compte une jointure dans certains cas, notamment grâce à l' existence d'une contrainte unique.

Les contraintes d'exclusion ont été ajoutées en 9.0. Elles permettent un test sur plusieurs colonnes avec différents opérateurs (et non pas que l'égalité dans le cas d'une contrainte unique, qui est après tout une contrainte d' exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

2.5.10 DOMAINES

- Types créés par les utilisateurs
- Permettent de créer un nouveau type à partir d'un type de base
- En lui ajoutant des contraintes supplémentaires.

Exemple:

```
=> CREATE DOMAIN code_postal_francais AS text check (value ~ '^\d{5}$');
CREATE DOMAIN
=> ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
ALTER TABLE
=> UPDATE capitaines SET cp='35400' WHERE nom LIKE '%Surcouf';
INSERT 0 1
```

[44] http://fr.wikipedia.org/wiki/Table_de_hachage
[45] http://docs.postgresql.fr/current/textsearch-indexes.html

```
=> UPDATE capitaines SET cp='1420' WHERE nom LIKE 'Haddock';
ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"
```

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, on peut 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.

2.5.11 ENUMS

- Types créés par les utilisateurs
- Permettent de définir une liste ordonnée de valeurs de type chaîne de caractère pour ce type

Exemple :

```
=> CREATE TYPE jour_semaine
  AS ENUM ('Lundi','Mardi','Mercredi','Jeudi','Vendredi',
  'Samedi','Dimanche');
CREATE TYPE
=> ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;
CREATE TABLE
=> UPDATE capitaines SET jour_sortie='Mardi' WHERE nom LIKE '%Surcouf';
UPDATE 1
=> UPDATE capitaines SET jour_sortie='Samedi' WHERE nom LIKE 'Haddock';
UPDATE 1
=> SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
 id |   nom    | age | num_cartecredit | cp | jour_sortie
----+----------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
(1 rows)
```

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, on aurait pu créer une table `jour_de_la_semaine`, et stocker la clé

associée dans `planning`. On aurait pu tout aussi bien positionner une contrainte `CHECK`, mais on n'aurait plus eu une liste ordonnée.

2.5.12 TRIGGERS

- Opérations: `INSERT`, `COPY`, `UPDATE`, `DELETE`
- 8.4, trigger `TRUNCATE`
- 9.0, trigger pour une colonne, et/ou avec condition
- 9.1, trigger sur vue
- 9.3, trigger DDL
- 10, tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)

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 nous donnent une vision des lignes avant et après modification.

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

Exemple :

```
=> ALTER TABLE capitaines ADD COLUMN salaire integer;
ALTER TABLE

=> CREATE FUNCTION verif_salaire()
  RETURNS trigger AS $verif_salaire$
  BEGIN
  -- On verifie 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 FUNCTION

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

=> UPDATE capitaines SET salaire=2000 WHERE nom='Haddock';
UPDATE 1
=> UPDATE capitaines SET salaire=3000 WHERE nom='Haddock';
UPDATE 1
=> UPDATE capitaines SET salaire=2000 WHERE nom='Haddock';
ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

2.6 SPONSORS & RÉFÉRENCES

- Sponsors
- 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 ?

2.6.1 SPONSORS

- NTT (Streaming Replication)
- Crunchy Data Solutions (Tom Lane, Stephen Frost, Joe Conway, Greg Smith)
- Microsoft Skype Division (projet skytools)
- EnterpriseDB (Bruce Momjian, Dave Page...)
- 2nd Quadrant (Simon Riggs...)
- VMWare (Heikki Linnakangas)

- Dalibo
- Fujitsu
- Red Hat
- Sun Microsystems (avant le rachat par Oracle)

À partir de juin 2006, le système d'exploitation Unix Solaris 10 embarque PostgreSQL dans sa distribution de base, comme base de données de référence pour ce système d'exploitation. Ce n'est plus le cas avec la sortie en 2011 de Oracle Solaris 11.

Le rachat de MySQL par Sun Microsystems ne constitue pas un danger pour PostgreSQL. Au contraire, Sun a rappelé son attachement et son implication dans le projet.

- News SUN[46]
- Article[47]

Le rachat de Sun Microsystems par Oracle ne constitue pas non plus un danger, bien qu'évidemment les ressources consacrées à PostgreSQL, autant humaines que matérielles, ont toutes été annulées.

NTT finance un groupe de développeurs sur PostgreSQL, ce qui lui a permis de fournir de nombreux patchs pour PostgreSQL, le dernier en date concernant un système de réplication interne au moteur. Ce système a été inclus dans la version de la communauté depuis la 9.0. Plus d'informations[48] .

Ils travaillent à un outil de surveillance de bases PostgreSQL assez poussé qu'ils ont présenté lors de PGCon 2010.

Fujitsu a participé à de nombreux développements aux débuts de PostgreSQL.

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. Il maintient quelques paquets RPM, dont ceux du SGBD PostgreSQL. Il assure une maintenance sur leur anciennes versions pour les distributions Red Hat à grande durée de vie. Tom Lane a travaillé également chez SalesForce, ensuite il a rejoint Crunchy Data Solutions fin 2015.

Skype est apparu il y a plusieurs années maintenant. Ils proposent un certain nombre d'outils très intéressants : PgBouncer (pooler de connexion), Londiste (réplication par trigger), etc. Ce sont des outils qu'ils utilisent en interne et qu'ils publient sous licence BSD comme retour à la communauté. Le rachat par Microsoft n'a pas affecté le développement de ces outils.

[46] http://www.news.com/Sun-backs-open-source-database-PostgreSQL/2100-1014_3-5958850.html
[47] http://www.lemondeinformatique.fr/actualites/lire-sun-encourage-a-essayer-la-version-83-de-postgresql-25253.html
[48] http://wiki.postgresql.org/wiki/Streaming_Replication

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

Dalibo participe pleinement à la communauté. La société est sponsor platinum du projet PostgreSQL[49] . Elle développe et maintient plusieurs outils plébiscités par la communauté, comme par exemple pgBadger ou Ora2Pg, avec de nombreux autres projets en cours, et une participation active au développement de patchs pour PostgreSQL. Elle sponsorise également des événements comme les PGDay français et européens, ainsi que la communauté francophone. Plus d'informations[50] .

2.6.2 RÉFÉRENCES

- Yahoo
- Météo France
- RATP
- CNAF
- Le Bon Coin
- Instagram
- Zalando
- TripAdvisor

Le DBA de TripAdvisor témoigne de leur utilisation de PostgreSQL dans l'interview suivante[51] .

2.6.3 LE BON COIN

Site de petites annonces :

- Base transactionnelle de 6 To
- 4è site le plus consulté en France (2017)
- 800 000 nouvelles annonces par jour

[49] http://www.postgresql.org/about/sponsors/
[50] http://www.dalibo.org/contributions
[51] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley

17.12

- 4 serveurs PostgreSQL en réplication
  - 160 cœurs par serveur
  - 2 To de RAM
  - 10 To de stockage flash

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

Voir les témoignages de ses directeurs technique[52] (témoignage de juin 2012) et infrastructure[53] (juin 2017) pour plus de détails sur la configuration.

2.7 CONCLUSION

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

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.

2.7.1 BIBLIOGRAPHIE

- Documentation officielle (préface)
- Articles fondateurs de M. Stonebracker
- Présentation du projet PostgreSQL

[52] http://www.postgresqlfr.org/temoignages:le_bon_coin
[53] https://www.kissmyfrogs.com/jean-louis-bergamo-leboncoin-ce-qui-a-ete-fait-maison-est-ultra-performant/

« Préface : 2. Bref historique de PostgreSQL[54] ». PGDG, 2013

« The POSTGRES™ data model[55] ». Rowe and Stonebraker, 1987

« Présentation du projet PostgreSQL[56] ». Guillaume Lelarge, 2008

Iconographie :

La photo initiale est le logo officiel de PostgreSQL[57] .

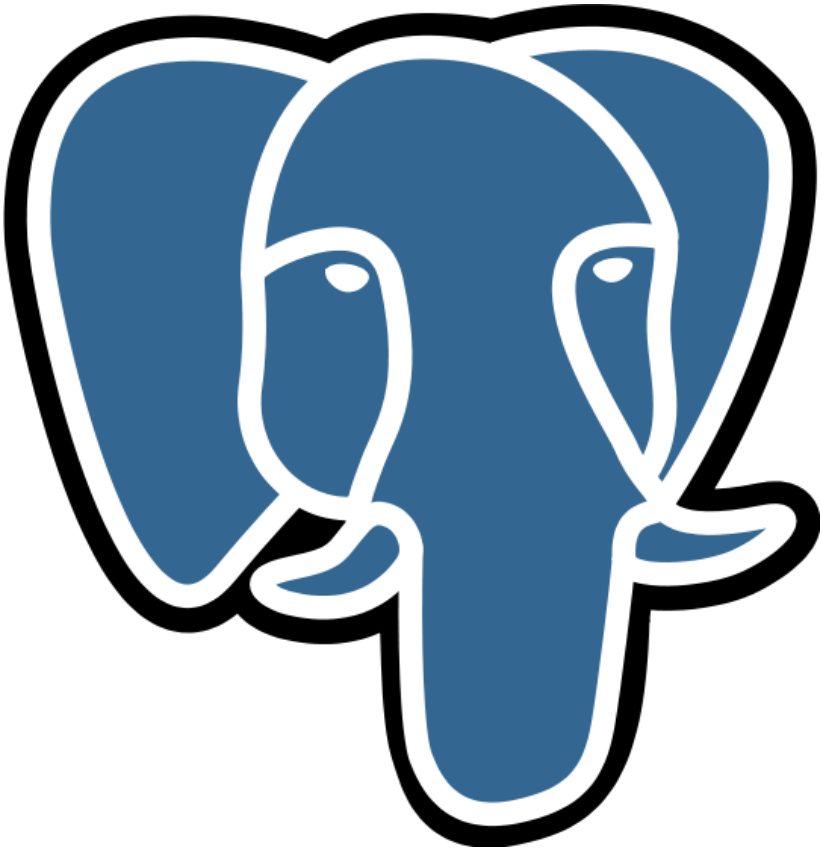
2.7.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[54] http://docs.postgresqlfr.org/9.4/history.html
[55] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[56] http://www.dalibo.org/presentation_du_projet_postgresql
[57] http://wiki.postgresql.org/wiki/Trademark_Policy

3 INSTALLATION DE POSTGRESQL



3.1 INTRODUCTION

- Installation depuis les sources
- Installation depuis les binaires
  - installation à partir de packages
  - installation sous windows
- Premiers réglages

- Mises à jours

Il existe trois façons d'installer PostgreSQL :

- les installeurs graphiques
  - 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
  - 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 INSTALLATION À PARTIR DES SOURCES

Étapes :

- Téléchargement
- Vérification des pré-requis
- Compilation
- Installation

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.

3.2.1 TÉLÉCHARGEMENT

- Disponible via :
  - HTTP
  - FTP
- Télécharger le fichier postgresql-X.Y.Z.tar.bz2

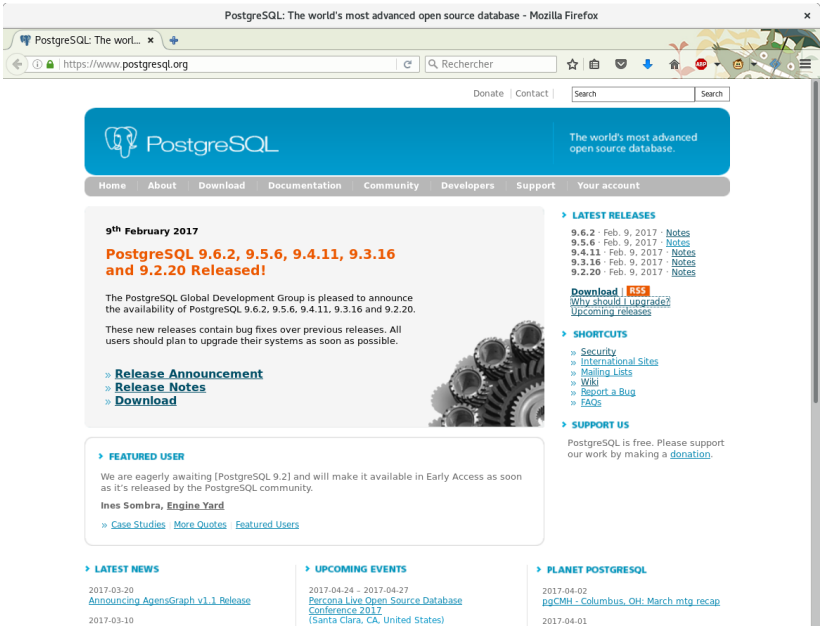
Les fichiers sources et les instructions de compilation sont disponibles sur le site officiel du projet[58] . Ceci étant dit, un serveur FTP anonyme[59] est également mis à la disposition des internautes. Le nom du fichier à télécharger se présente toujours sous la forme « postgresql-X.Y.Z.tar.bz2 » où X.Y.Z représente la version de PostgreSQL.

Lorsque la future version du logiciel est en phase de démonstration (alpha) ou de test (versions bêta), les sources sont accessibles à ces adresses :

- http://www.postgresql.org/developer/alpha
- http://www.postgresql.org/developer/beta

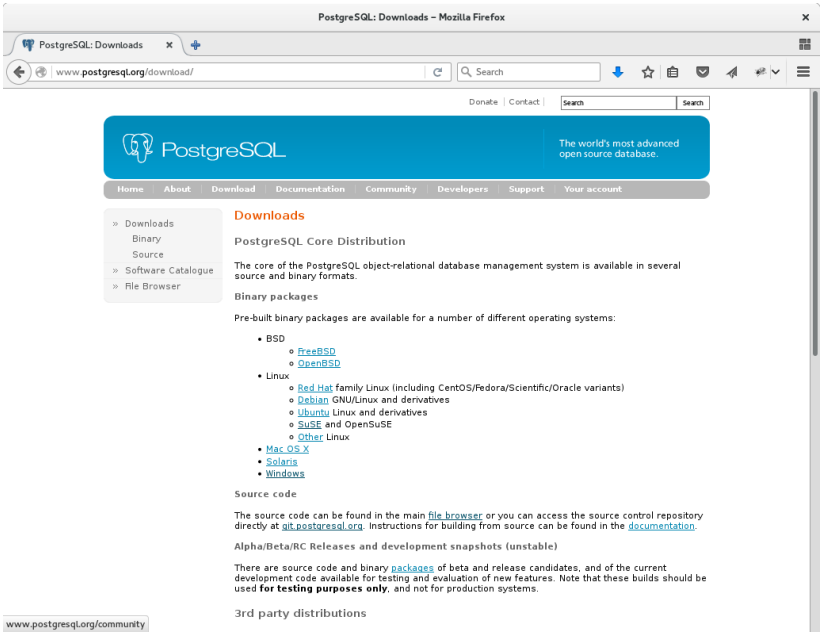
Voici comment récupérer la dernière version des sources de PostgreSQL :

- Aller sur la page d'accueil du projet PostgreSQL



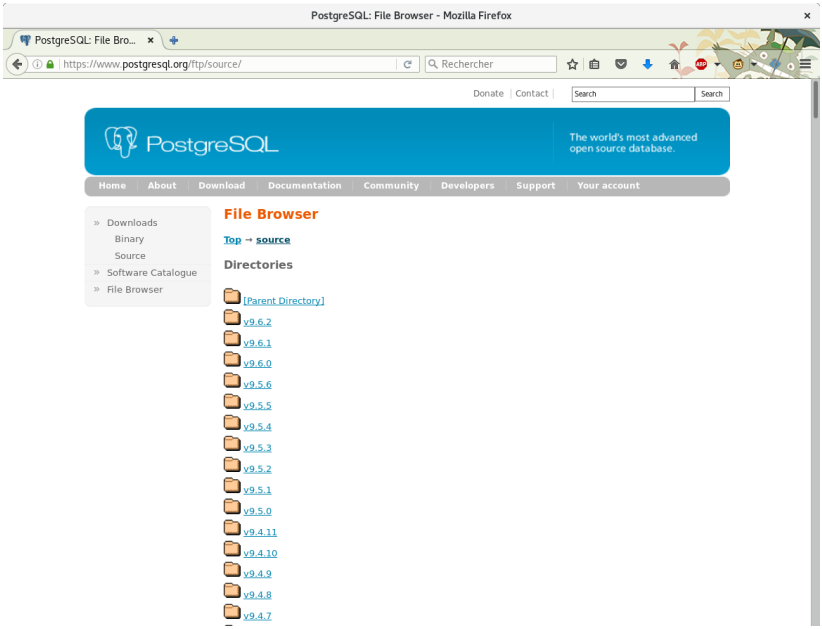
[58] http://www.postgresql.org/download/
[59] ftp://ftp.postgresql.org

* Cliquer sur le lien *Downloads*

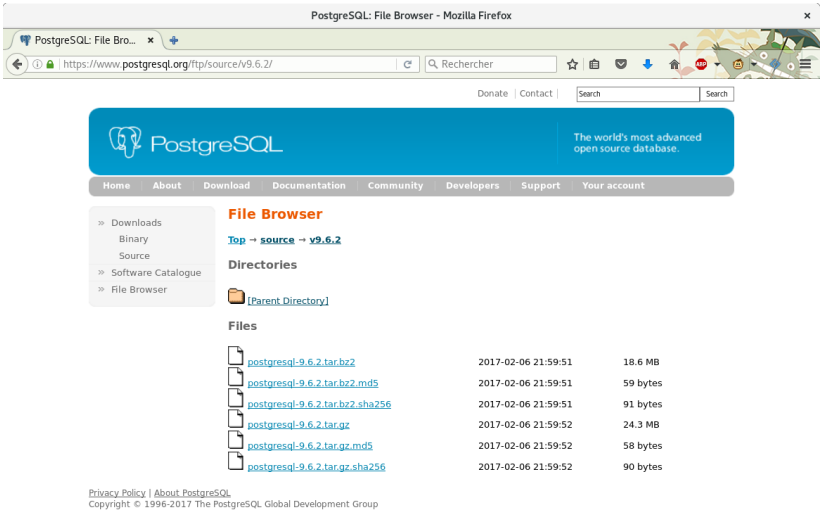


* Cliquer sur le lien *file browser* de la partie *Source code*

17.12

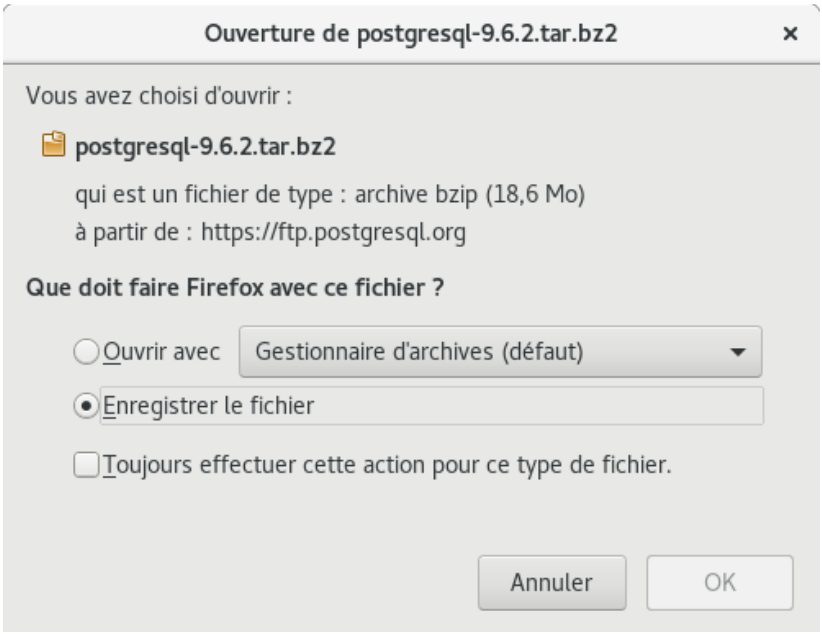


* Cliquer sur le lien « v9.6.2 » (la 9.6.2 était la dernière version disponible lors de la mise à jour de cette présentation ; utilisez la version la plus récente possible)



* Cliquer sur le lien « postgresql-9.6.2.tar.bz2 »

17.12



3.2.2 PHASES DE COMPILATION/INSTALLATION

- Processus standard :

```
$ tar xvfj postgresql-<version>.tar.bz2
$ cd postgresql-<version>
$ ./configure
$ make
$ make install
```

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. **Attention aux options de configuration !**

```
$ cd postgresql-<version>
$ ./configure [OPTIONS]
```

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/dynloader/linux.c
               to src/backend/port/dynloader.c
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/backend/port/dynloader/linux.h to src/include/dynloader.h
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
```

On passe alors à la phase de compilation des sources de PostgreSQL pour en construire les différents exécutables :

```
$ make
```

Cette phase est la plus longue. Cependant, cela reste assez rapide sur du matériel récent. Il est possible de le rendre plus rapide 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`.

L'opération de compilation doit s'arrêter avec le message suivant :

```
All of PostgreSQL successfully made. Ready to install.
```

Dans le cas contraire, une erreur s'est produite et il est nécessaire de la corriger avant de continuer.

Si le message de réussite de compilation est affiché, il est possible d' installer le résultat de la compilation :

```
$ make install
```

Là-aussi, un message spécifique doit être affiché pour indiquer le résultat de l'installation :

```
PostgreSQL installation complete.
```

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`.

3.2.3 OPTIONS POUR ./CONFIGURE

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

```
$ pg_config --configure
```

Le script de configuration de la compilation possède un certain nombre de paramètre optionnels. Parmi toutes ces options, citons les suivantes qui sont probablement les plus intéressantes:

- `--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`: permet d'installer l'interface Perl ainsi que les extensions Perl de PostgreSQL.

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. La personne qui a procédé à cette compilation n'est pas forcément là, n'a pas forcément conservé cette information, et il faut donc un autre moyen pour récupérer cette information. L'outil pg_config le permet ainsi :

```
$ pg_config --configure
  '--prefix=/usr'
  '--mandir=/usr/share/postgresql/8.4/man'
  '--with-docdir=/usr/share/doc/postgresql-doc-8.4'
  '--datadir=/usr/share/postgresql/8.4'
  '--bindir=/usr/lib/postgresql/8.4/bin'
  '--libdir=/usr/lib/postgresql/8.4/lib'
  '--includedir=/usr/include/postgresql/8.4'
  '--enable-nls'
  '--enable-integer-datetimes'
  '--enable-thread-safety'
  '--enable-debug' '--disable-rpath' '--with-tcl'
  '--with-perl' '--with-python' '--with-pam'
  '--with-krb5' '--with-openssl' '--with-gnu-ld'
  '--with-tclconfig=/usr/lib/tcl8.5'
  '--with-tkconfig=/usr/lib/tk8.5'
  '--with-includes=/usr/include/tcl8.5'
  '--with-system-tzdata=/usr/share/zoneinfo'
  '--sysconfdir=/etc/postgresql-common'
  '--with-gssapi' '--with-libxml'
  '--with-libxslt' '--with-ldap' '--with-ossp-uuid'
  'CFLAGS= -fPIC' 'LDFLAGS= -Wl,--as-needed'
```

3.2.4 TESTS DE 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/postgresql-10beta3/tmp_install/home/dalibo/pg/bin:$PATH"
LD_LIBRARY_PATH="/home/dalibo/postgresql-10beta3/tmp_install/home/dalibo/pg/lib"
../../../src/test/regress/pg_regress --temp-instance=./tmp_check --inputdir=.
--bindir=     --dlpath=.  --schedule=./parallel_schedule
============== creating temporary instance            ==============
```

17.12

```
============== initializing database system           ==============
============== starting postmaster                    ==============
running on port 50848 with PID 13844
============== creating database "regression"         ==============
CREATE DATABASE
ALTER DATABASE
============== running regression test queries        ==============
test tablespace               ... ok
parallel group (20 tests):  name money text regproc txid float4 varchar char
   float8 boolean uuid oid int2 enum pg_lsn int8 bit int4 numeric rangetypes
     boolean                  ... ok
     char                     ... ok
     name                     ... ok
     varchar                  ... ok
     text                     ... ok
...
     largeobject              ... ok
     with                     ... ok
     xml                      ... ok
test identity                 ... ok
test event_trigger            ... ok
test stats                    ... ok
============== shutting down postmaster               ==============
============== removing temporary instance            ==============

=======================
 All 178 tests passed.
=======================
...
```

Les tests de régression sont une suite de tests qui vérifient que PostgreSQL fonctionne correctement sur la machine cible. Ces tests ne doivent 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.

3.2.5 CRÉATION DE L'UTILISATEUR

- Ajout d'un utilisateur

  - lancera PostgreSQL
  - sera le propriétaire des répertoires et fichiers
- 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
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. La version 9.5 rend cette pratique moins intéressante grâce à la possibilité de nommer les instances avec le paramètre `cluster_name`, information affichée au niveau des processus gérant une instance.

Il peut aussi se révéler nécessaire de positionner un certain nombre de variables d'environnement.

Afin de rendre l'exécution de PostgreSQL possible à partir de n'importe quel répertoire, il est très pratique (essentiel ?) d'ajouter le répertoire d' installation des exécutables (`/usr/local/pgsql/bin` par défaut ou le chemin indiqué à l'option `--prefix` lors de l'étape `configure`) aux chemins de recherche. Pour cela, nous modifions la variable d'environnement `PATH`. En complément, la variable `LD_LIBRARY_PATH` indique au système où trouver les différentes bibliothèques nécessaires à l'exécution des programmes. Voici un exemple avec ces deux variables :

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
```

D'autres variables d'environnement peuvent éventuellement être utiles:

- `MANPATH` pour ajouter aux chemins de recherche des pages de manuel celui de votre installation de PostgreSQL ;
- `PGDATA` qui sera consulté par certains exécutables de PostgreSQL pour déterminer où se trouve le répertoire de travail de l'instance.

Voici un exemple avec ces deux variables :

```
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
export PGDATA=/usr/local/pgsql/data
```

17.12

Afin de vous éviter d'avoir à redéfinir ces variables d'environnement après chaque redémarrage du système, il est conseillé d'inclure ces lignes dans votre fichier `${HOME}/.profile` ou dans `/etc/profile`.

3.2.6 CRÉATION DU RÉPERTOIRE DE DONNÉES

- Outil initdb
- Création du répertoire

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

- Options d'emplacement
  - `--data` pour les fichiers de données
  - `--waldir` pour les journaux de transactions
- Option sécurité
  - `--pwprompt` pour configurer immédiatement le mot de passe de l'utilisateur `postgres`
  - `--data-checksums` pour ajouter des sommes de contrôle sur les fichier de données

Notez que vous devez exécuter cette commande en étant connecté sous le compte de l'utilisateur PostgreSQL décrit dans la section précédente (généralement l'utilisateur `postgres`).

Si le répertoire n'existe pas, `initdb` tentera de le créer. Toutefois, vous devez veiller à ce qu'il ait les droits pour le faire. S'il existe, il doit être vide.

Voici ce qu'affiche cette commande :

```
$ initdb -D /usr/local/pgsql/data
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 locale "en_US.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 disabled.

creating directory /usr/local/pgsql/data ... ok
creating subdirectories ... ok
```

```
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

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.

De plus, 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`.

Il est aussi possible d'activer les sommes de contrôle des fichiers de données avec l'option `--data-checksums`. Cette option est apparue avec PostgreSQL 9.3 mais impose une dégradation sensible des performances. La version 9.5 amène une amélioration qui réduit l'impact sur les performances.

L'option `--waldir` avait pour nom `--xlogdir` avant la version 10. Cependant, l'option courte n'a pas été modifiée.

3.2.7 LANCEMENT ET ARRÊT

- Script de démarrage

17.12

```
# /etc/init.d/postgresql [action]
```

- Outil pg_ctl

```
$ pg_ctl --pgdata /usr/local/pgsql/data --log logfile [action]
```

- `[action]` dépend du besoin
  - start pour démarrer
  - stop pour arrêter
  - reload pour recharger la configuration
  - restart pour redémarrer

La méthode recommandée est d'utiliser le script de démarrage. 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.

L'autre méthode est fonctionnelle mais moins intéressante. Elle est à exécuter avec l'utilisateur qui a été créé précédemment.

Les deux méthodes partagent la majorité des actions présentées ci-dessus. Cependant, `pg_ctl` permet de préciser plus facilement le mode d'arrêt parmi les trois disponibles :

- smart : pour vider le cache de PostgreSQL sur disque, et attendre la fin de l' exécution des clients (pgAdmin, pg_dump, psql, libpq, etc) ;
- fast : pour vider le cache sur disque et déconnecter les clients sans attendre (de ce fait, les transactions en cours sont annulées) ;
- immediate : équivalent à un arrêt brutal, tous les processus serveur sont tués (de ce fait, au redémarrage, le serveur devra rejouer ses journaux de transactions).

Par défaut, le mode `fast` est utilisé. Avant la version 9.5, le mode par défaut était `smart`. Il est possible de forcer le mode avec l'option `-m`.

3.3 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.

3.3.1 PAQUETS DEBIAN OFFICIELS

- Différents paquets disponibles
  - serveur, client, contrib, docs
  - et les extensions, outils
- `apt-get install postgresql-<version majeure>`
  - installe les binaires
  - crée l'utilisateur « postgres »
  - exécute initdb
  - démarre le serveur
- Particularités
  - wrappers/scripts pour la gestion des différentes instances
  - plusieurs versions majeures installables
  - respect de la FHS

Sous Debian et les versions dérivées (Ubuntu par exemple), l'installation de PostgreSQL a été découpée en plusieurs paquets :

- le serveur : postgresql-
- les clients : postgresql-client-
- les modules contrib : postgresql-contrib-
- la documentation : postgresql-doc-

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-plpython-
- PL/perl dans postgresql-plperl-
- PL/tcl dans postgresql-pltcl-
- 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-.

Le `<version majeure>` correspond à la version majeure souhaitée (`9.6` par exemple). Cela permet d'installer plusieurs versions majeures sur le même serveur physique ou virtuel. Les exécutables sont installés dans :

`/usr/lib/postgresql/<version majeure>/bin`

les fichiers de configuration dans :

`/etc/postgresql/<version majeure>/instance`

les traces dans :

`/var/log/postgresql/postgresql-<version majeure>.log`

et les données dans :

`/var/lib/postgresql/<version majeure>/instance`

`instance` correspond au nom de l'instance (`main` par défaut). Ceci est fait pour respecter le plus possible la norme FHS[60] (*Filesystem Hierarchy Standard*).

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

Quand le paquet serveur est installé, plusieurs opérations sont exécutées : téléchargement du paquet, installation des binaires contenus dans le paquet, création de l'utilisateur `postgres` (s'il n'existe pas déjà), création du répertoire des données, lancement du serveur. En cas de mise à jour d'un paquet, le serveur PostgreSQL est redémarré après mise à jour des binaires. Tout ceci explique le grand intérêt de passer par les paquets Debian sur ce type de distribution.

3.3.2 PAQUETS DEBIAN COMMUNAUTAIRES

- La communauté met des paquets Debian à disposition :
  - http://wiki.postgresql.org/wiki/Apt
- Synchrone avec le projet PostgreSQL
- Ajout du dépôt dans /etc/apt/sources.list.d/pgdg.list :
  - deb http://apt.postgresql.org/pub/repos/apt/ jessie-pgdg main

Les paquets de la communauté ont le même contenu que les paquets officiels Debian. La seule différence est qu'ils sont mis à jour plus fréquemment, en liaison directe avec la communauté PostgreSQL.

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. Il est donc parfois difficile de mettre à jour avec la dernière version de PostgreSQL. De ce fait, la communauté PostgreSQL met à disposition son propre dépôt de paquets Debian. Elle en assure le maintien et le support. L'équipe de mainteneurs est généralement prévenue trois/quatre jours avant la sortie d'une version pour qu'elle puisse préparer des paquets qui seront disponibles le jour de la sortie officielle.

Les dépôts sont situés sur le serveur `apt.postgresql.org`. On ajoutera la déclaration suivante dans les fichiers de configuration du gestionnaire `apt`, par exemple pour une

[60] http://www.pathname.com/fhs/

distribution Debian 8 (nom de code Jessie) :

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

Le nom de code de la distribution peut être obtenu avec la commande :

```
$ lsb_release -c
```

Enfin, pour finaliser la déclaration, il est nécessaire de récupérer la clé publique du dépôt communautaire :

```
# wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC4CF8.asc \
    | sudo apt-key add -
# apt-get update
```

3.3.3 PAQUETS REDHAT OFFICIELS

- Différents paquets disponibles
  - serveur, client, contrib, docs
  - et les extensions, outils
- `yum install postgresqlxx-server`
  - installe les binaires
  - crée l'utilisateur « postgres »
- Opérations manuelles
  - initdb
  - lancement du serveur
- Particularités
  - fichiers de configuration des instances
  - plusieurs versions majeures installables
  - respect du stockage PostgreSQL

Sous RedHat et les versions dérivées (CentOS, Scientific Linux par exemple), l 'installation de PostgreSQL a été découpée en plusieurs paquets :

- le serveur : postgresql-server
- les clients : postgresql
- les modules contrib : postgresql-contrib
- la documentation : postgresql-docs

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-plpython
- PL/perl dans postgresql-plperl

- PL/tcl dans postgresql-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.

Le `xx` correspond à la version majeure souhaitée (95 pour une version 9.5 par exemple). Cela 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-<version majeure>/bin`, les traces dans `/var/lib/pgsql/<version majeure>/data/log` (utilisation du `logger process` de PostgreSQL), les données dans `/var/lib/pgsql/<version majeure>/data`. `data` est le répertoire par défaut des données, mais il est possible de le surcharger. Plutôt que de respecter la norme FHS (`Filesystem Hierarchy Standard`), RedHat a fait le choix de respecter l'emplacement des fichiers utilisé par défaut par les développeurs PostgreSQL.

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. Cela se fait de deux façons, suivant que le système où est installé PostgreSQL utilise `systemd` ou non. Dans le cas où il est utilisé, un script, nommé `/usr/psql-<version majeure>/bin/postgresqlxx-setup`, est mis à disposition pour lancer `initdb` avec les bonnes options. Dans le cas contraire, il faut passer par le script de démarrage, auquel on fournit l'action `initdb`.

Pour installer plusieurs instances, il est préférable de créer des fichiers de configuration dans le répertoire `/etc/sysconfig/pgsql`. Le nom du fichier de configuration doit correspondre au nom du script de démarrage. La première chose à faire est donc de faire un lien entre le script de démarrage avec un nom permettant de désigner correctement l'instance :

```
# ln -s /etc/init.d/postgresql-<version majeure> /etc/init.d/postgresql-serv2
```

Puis, de créer le fichier de configuration avec les paramètres essentiels (généralement PGDATA, PORT) :

```
# echo >>/etc/sysconfig/pgsql/postgresql-serv2 <<_EOF_
PGDATA=/var/lib/pgsql/<version majeure>/serv2
PORT=5433
```

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

3.3.4 PAQUETS REDHAT COMMUNAUTAIRES

- La communauté met des paquets RedHat à disposition :
  - http://yum.postgresql.org
- Synchrone avec le projet PostgreSQL
- Ajout du dépôt grâce aux paquets RPM disponibles

La différence entre les paquets officiels et ceux de la communauté tient principalement en une disponibilité immédiate des mises à jour.

L'installation de la configuration du dépôt est très simple. Il suffit de télécharger le paquet RPM de définition du dépôt, puis de l'installer avec la commande `rpm`.

3.4 INSTALLATION SOUS WINDOWS

- Disponible pour les différentes versions NT de Windows (2003 server, 2008 server, 2012 server, etc).
- Deux installeurs graphiques disponibles
  - EnterpriseDB
  - BigSQL
- Ou archive des binaires

Le portage de PostgreSQL sous Windows date de la version 8.0. Lors du développement de cette version, le travail qu'il a nécessité fût suffisamment important pour justifier à lui seul le passage de la branche 7 à la branche 8 du projet. La version Windows a été considérée comme une version beta pendant les versions 8.0 et 8.1. La version 8.2 est donc la première version stable de PostgreSQL sous Windows. Seules les versions NT sont supportés car seules ces versions disposent du système de fichiers NTFS. Ce système de fichiers est obligatoire car, contrairement à la VFAT, il gère les liens symboliques (appelés jonctions sous Windows). Une version 64 bits n'est disponible que depuis la version 9.0.

Dans un premier temps, la communauté proposait son installeur graphique, et EnterpriseDB en proposait un autre. Étant donné la quantité de travail nécessaire pour le développement et la maintenance de l'installeur graphique, la communauté a abandonné son installeur graphique. EntrepriseDB a continué de proposer le sien. Il contient le serveur PostgreSQL avec les modules contrib ainsi que pgAdmin3. Il contient aussi un outil appelé StackBuilder permettant d'installer d'autres outils (comme PostGIS ou phpPgAdmin).

Lorsque l'équipe de pgAdmin a décidé d'abandonner pgAdmin3 au profit de pgAdmin4 (et donc quand EnterpriseDB a mis à jour son installeur pour installer pgAdmin4), BigSQL a

décidé de créer son propre installeur, fournissant le serveur PostgreSQL avec les modules contribs, ainsi qu'une version améliorée de pgAdmin3.

En soit, la grosse différence entre ces deux installeurs est la version de pgAdmin. Les versions de PostgreSQL proposées sont les versions communautaires à notre connaissance.

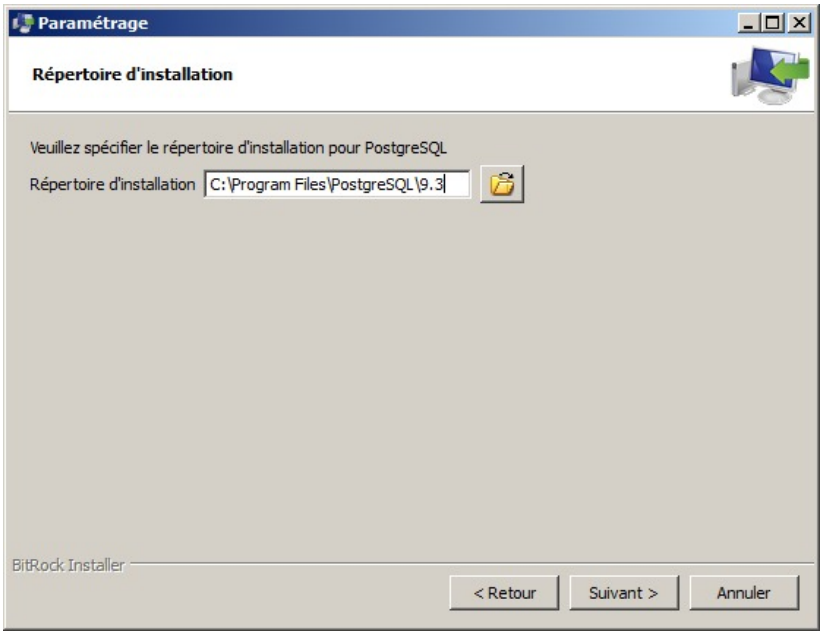
Les slides suivantes ne concernent que l'installeur d'EntrepriseDB.

3.4.1 INSTALLEUR GRAPHIQUE - BIENVENUE

C'est l'écran d'accueil de l'installeur, il suffit de cliquer sur le bouton Suivant.

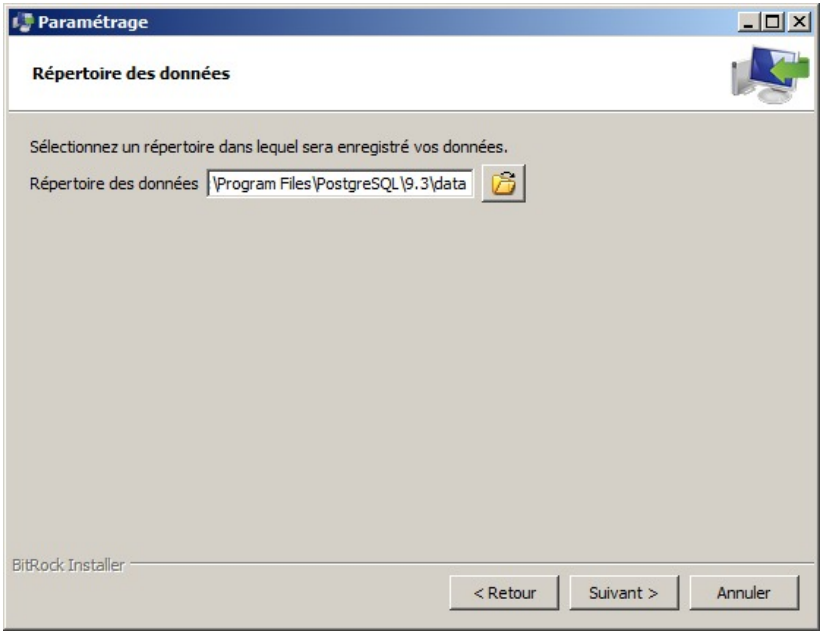
L'indication « Packaged by EnterpriseDB » signifie que l'installeur a été développé et proposé au téléchargement par la société EnterpriseDB. En effet, maintenir un installeur Windows est un travail conséquent et personne dans la communauté ne souhaite actuellement faire ce travail. Il n'empêche que cet installeur est disponible gratuitement et propose la version communautaire de PostgreSQL.

3.4.2 INSTALLEUR GRAPHIQUE - RÉPERTOIRE D'INSTALLATION



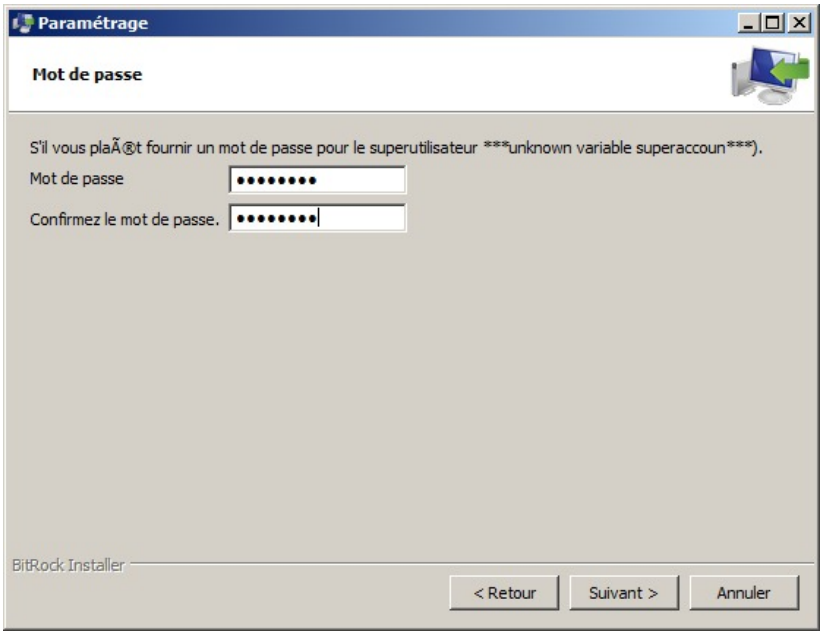
Cet écran sert à saisir le répertoire d'installation. Ce dernier a une valeur par défaut généralement convenable car il n'existe pas vraiment de raison d' installer les binaires PostgreSQL en dehors du répertoire Programmes.

3.4.3 INSTALLEUR GRAPHIQUE - RÉPERTOIRE DONNÉES



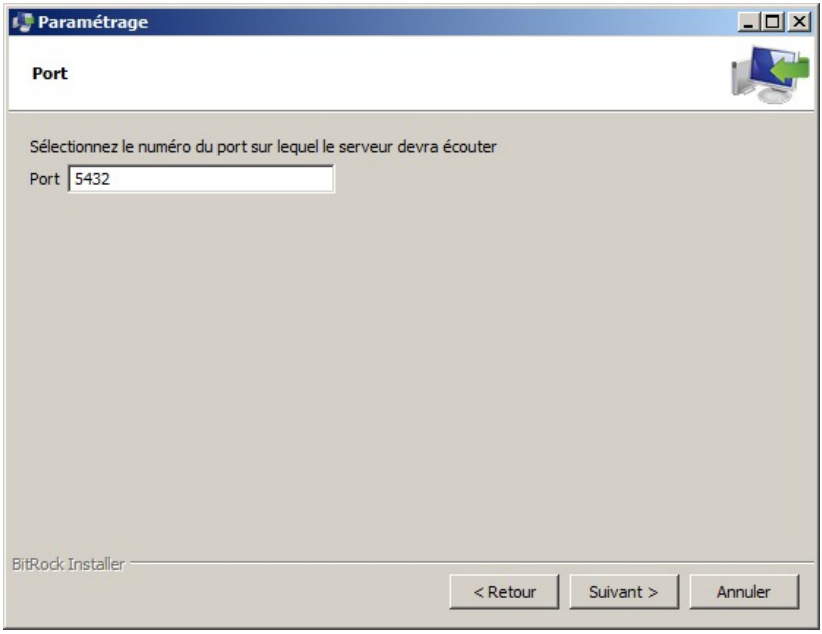
Cet écran sert à saisir le répertoire des données de l'instance PostgreSQL. La valeur par défaut de ce dernier ne convient pas forcément à toutes les installations. Il est par exemple souvent modifié pour que le répertoire des données se trouve sur un autre disque que le disque système.

3.4.4 INSTALLEUR GRAPHIQUE - MOT DE PASSE



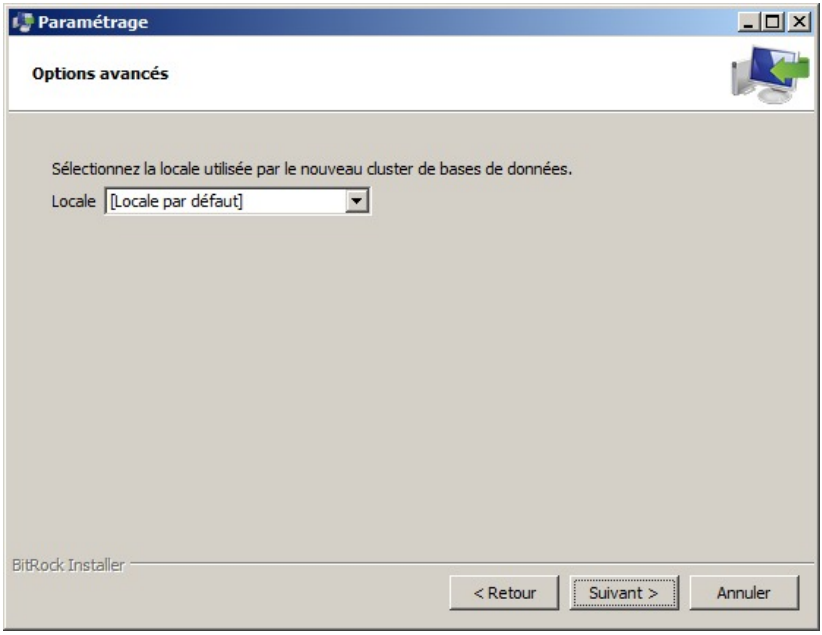
L'installeur graphique crée un compte utilisateur système. Le mot de passe saisi ici est le mot de passe de ce compte système. Il sera aussi utilisé par le compte de l'utilisateur `postgres` sous PostgreSQL. En cas de mise à jour, il faut saisir l'ancien mot de passe.

3.4.5 INSTALLEUR GRAPHIQUE - PORT



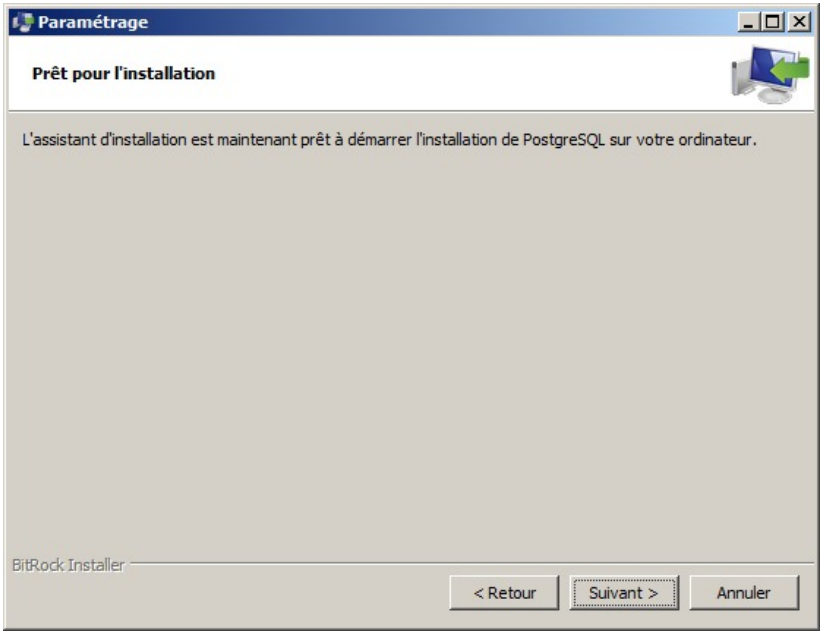
Le numéro de port indiqué est par défaut le 5432, sauf si d'autres instances sont déjà installés. Dans ce cas, l'installeur propose un numéro de port non utilisé.

3.4.6 INSTALLEUR GRAPHIQUE - AUTRES



L'installeur propose d'utiliser la locale par défaut mais il est possible de la modifier.

3.4.7 INSTALLEUR GRAPHIQUE - PRÊT



3.4.8 INSTALLEUR GRAPHIQUE - TERMINÉ

À ce moment-là, un nouvel utilisateur a été créé avec le nom `postgres`. La commande `initdb` a été exécutée pour créer le répertoire des données. Un service a été 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. Il est à noter que cet installeur a aussi installé pgAdmin, dans une version compatible avec la version du serveur PostgreSQL.

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

3.5 PREMIERS RÉGLAGES

- Sécurité
- Configuration minimale

- Démarrage
- Test de connexion

3.5.1 SÉCURITÉ

- Politique de mot de passe
  - 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.

Autant il est possible de ne pas fournir de mot de passe pour l'utilisateur système, autant il faut absolument en donner un pour le rôle `postgres`. Cela se fait avec cette requête :

```
ALTER ROLE postgres ENCRYPTED PASSWORD 'mot de passe';
```

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 si applicable, en utilisant quel rôle, pour quelle base de données et avec quel méthode d'authentification.

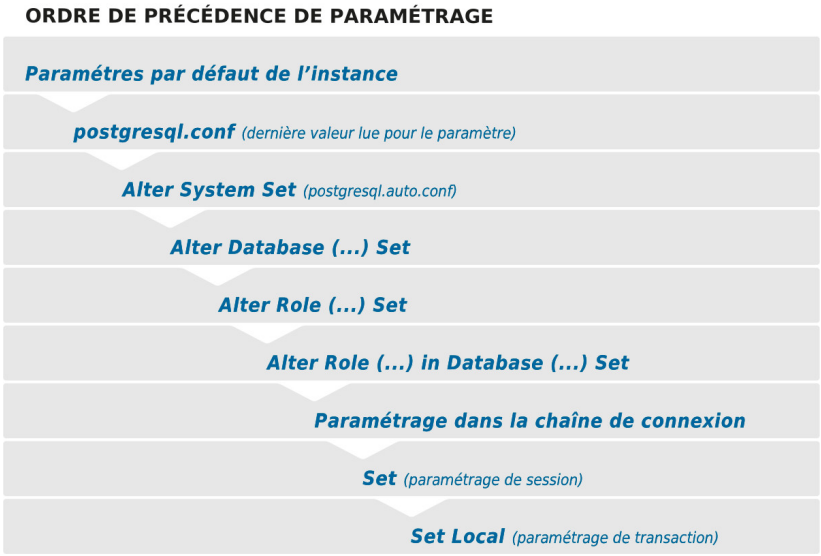
3.5.2 CONFIGURATION MINIMALE

- Fichier `postgresql.conf`
- Configuration du moteur
- Plus de 200 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 un sous-répertoire du répertoire `/etc` .

Ce fichier contient beaucoup de paramètres, plus de 200, mais seuls quelques-uns sont essentiels à connaître pour avoir une instance fiable et performante.

3.5.3 CONFIGURATION 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`.

3.5.4 CONFIGURATION DES CONNEXIONS

- listen_addresses = '*'
- port = 5432
- max_connections = 100

Par défaut, le serveur installé n'écoute pas sur les interfaces réseaux externes. Pour autoriser les clients externes à se connecter à PostgreSQL, il faut modifier le paramètre `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 séparant avec des virgules.

Le port par défaut des connexions TCP/IP est le 5432. Il est possible de le modifier. Cela n'a réellement un intérêt que si vous voulez exécuter plusieurs serveurs PostgreSQL sur le même serveur (physique ou virtuel). En effet, plusieurs serveurs sur une même machine ne peuvent pas écouter sur le même port.

Le nombre de connexions simultanées est limité par le paramètre max_connections. Dès que ce nombre est atteint, les connexions suivantes sont refusées jusqu'à ce qu'un utilisateur connecté se déconnecte. La valeur par défaut de 100 est généralement suffisante. Il peut être intéressant de la diminuer (pour que chaque processus ait accès à plus de mémoire) ou de l' augmenter (pour qu'un plus grand nombre d'utilisateurs puisse se connecter en même temps). Attention toutefois à ne pas dépasser 1000 car cela serait source de contre-performance.

3.5.5 CONFIGURATION DES RESSOURCES MÉMOIRE

- shared_buffers
- work_mem
- maintenance_work_mem

Chaque fois que PostgreSQL a besoin de lire ou écrire des données, il les place 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`. Sur un serveur dédié à PostgreSQL, une valeur d'un quart de la mémoire vive totale est généralement suffisante. Suivant les cas, une valeur inférieure ou supérieure sera encore meilleure pour les performances. À vous de tester ce qui vous convient le mieux. Attention, il faut noter que l'augmentation de cette valeur peut nécessiter la modification du paramètre système SHMMAX pour que PostgreSQL ait le droit de

réclamer autant de mémoire au démarrage. Cette restriction disparaît avec PostgreSQL 9.3.

`work_mem` et `maintenance_work_mem` sont des paramètres mémoires utilisés par chaque processus. `work_mem` est la taille de la mémoire allouée aux opérations de tri et hachage, alors que `maintenance_work_mem` est la taille de la mémoire pour les `VACUUM`, `CREATE INDEX` et ajout de clé étrangère. Cette mémoire est distincte de celle allouée par le paramètre `shared_buffers`, et elle sera allouée pour chaque nouvelle connexion PostgreSQL. Donc, si votre `max_connexions` est important, ces valeurs (en fait, surtout `work_mem`) devront être faibles. Alors que si votre `max_connections` est faible, ces valeurs pourront être augmentées.

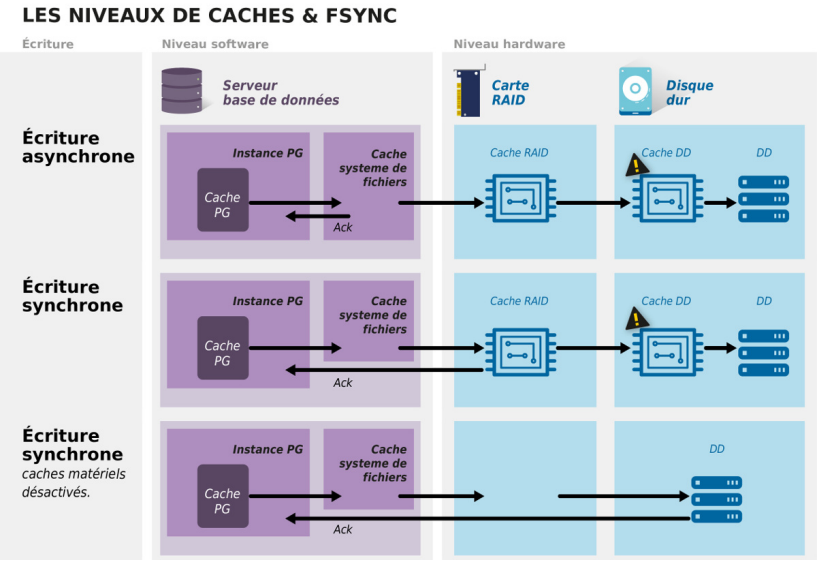
3.5.6 CONFIGURATION DES JOURNAUX DE TRANSACTIONS 1/2

- Paramètre : fsync

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. Le paramètre `fsync` permet de désactiver l'envoi de l'ordre de synchronisation au système d'exploitation. Ce paramètre ne doit jamais être désactivé en production.

3.5.7 CONFIGURATION DES JOURNAUX DE TRANSACTIONS 2/2



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 écrit dans le cache "système de fichier" du système d'exploitation (OS) situé en RAM (mémoire volatile). Celui-ci confirme au processus que l'écriture a bien été réalisée. Le bloc sera écrit sur disque "plus tard" afin de grouper les demandes d'écritures des autres processus et réduire les déplacements des têtes de lecture/écriture des disques qui sont des opérations coûteuses en temps. Entre la confirmation et l'écriture réelle sur les disques il peut se passer un certain délai, si une panne arrivait durant celui-ci les données n' étant pas encore présentes physiquement sur les disques elles seraient donc perdues.

- Dans le cas d'une écriture synchrone : Un processus écrit dans le cache du système d'exploitation et demande à l'OS d'effectuer une synchronisation sur disque. Le bloc est donc écrit physiquement sur les disques et le processus a la confirmation de l'écriture à la fin de l'opération. Il a donc la garantie que la donnée est bien présente physiquement sur les disques. Cette opération est très coûteuse et lente (déplacement des têtes des disques). Pour gagner en performance, les constructeurs ont donc mis en place un système de cache dans les cartes RAID, le processus

a la confirmation de l'écriture une fois la donnée présente dans ce cache même si 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.

3.5.8 CONFIGURATION DES TRACES

- log_destination
- logging_collector
- log_line_prefix
- lc_messages

PostgreSQL dispose de plusieurs moyens pour enregistrer les traces : soit il les envoie sur la sortie des erreurs (`stderr` et `csvlog`), 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 « log collector » qui va enregistrer les messages dans des fichiers. Ce démon s'active en configurant le paramètre `logging_collector` à `on`. Si vous faites cela, n'oubliez pas de changer le préfixe des messages pour ajouter au moins l'horodatage (`log_line_prefix = '%t'`). Des traces sans date et heure ne servent à rien.

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 les outils pgFouine et pgBadger, pourtant écrit par des Français, ne savent pas interpréter des traces en français. De plus, la moindre recherche sur Internet ramènera plus de liens avec des traces en anglais.

3.5.9 CONFIGURATION DES DÉMONS

- autovacuum
- stats collector

En dehors du « logger process », PostgreSQL dispose d'autres démons. L' autovacuum joue un rôle important pour de bonnes performances : il empêche une fragmentation excessive des tables et index, et met à jour les statistiques sur les données (qui servent à l'optimiseur de requêtes). Le collecteur de 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 démons devraient toujours être activés.

3.6 MISE À JOUR

- Recommandations
- Mise à jour mineure
- Mise à jour majeure

3.6.1 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 on trouve des conseils essentiels.

Les « Releases Notes » sont présentes dans l'annexe E de la documentation officielle[61] .

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 donnée, sauf cas exceptionnel qui serait alors précisé dans les notes de version (par exemple, de la 9.6.1 à la 9.6.2, il est nécessaire de reconstruire les index construits en mode concurrent). Pensez éventuellement à faire une sauvegarde préalable par sécurité.

A contrario, si la mise à jour consiste en un changement de version majeure (par exemple, de la 9.4 à la 9.6), alors 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.

Dans tous les cas, pensez à bien redémarrer le serveur. Mettre à jour les binaires ne suffit pas.

[61] http://docs.postgresql.fr/current/release.html

3.6.2 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-get` sur Debian et affiliés, `yum` sur RedHat 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.

3.6.3 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
- Pour mettre à jour
  - mise à jour des binaires
  - et mise à jour/traitement des fichiers de données
- 3 méthodes
  - dump/restore
  - Slony
  - pg_upgrade

17.12

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.

Une fois les tests 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.

3.6.4 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_dump (ou pg_dumpall) et 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 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.

3.6.5 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 qu'il est acté que le nouveau serveur donne pleine satisfaction, il suffit de désinstaller Slony des deux côtés.

3.6.6 MISE À JOUR MAJEURE PAR PG_UPGRADE

- Outil développé par la communauté depuis la version 8.4
  - et fourni avec PostgreSQL
- Prend comme prérequis que le format de stockage des fichiers de données utilisateurs ne change pas entre versions
- Nécessite les deux versions sur le même serveur

pg_upgrade est certainement l'outil le plus rapide pour une mise à jour majeure. 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 plus rapidement, il est aussi 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.

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.

17.12

3.7 CONCLUSION

- L'installation est simple....
- ... mais elle doit être soigneusement préparée
- Préférer les paquets officiels.
- Attention aux données lors d'une mise à jour !

3.7.1 POUR ALLER PLUS LOIN

- Documentation officielle, chapitre `Installation`
- Documentation Dalibo, pour l'installation sur Windows

Vous pouvez retrouver la documentation en ligne sur : http://docs.postgresql.fr/current/installation.html

ainsi que la partie spécifique à l'installation sur : http://docs.postgresql.fr/current/INSTALL.html

Les documentations de Dalibo pour l'installation de PostgreSQL sur Windows sont disponibles sur :

- http://www.dalibo.org/installation_de_postgresql_8.3_sous_windows
- http://www.dalibo.org/installation_de_postgresql_8.4_sous_windows
- http://www.dalibo.org/installation_de_postgresql_9.0_sous_windows

---

3.7.2 QUESTIONS

N'hésitez pas, c'est le moment !

3.8 TRAVAUX PRATIQUES

3.8.1 ÉNONCÉS

Installation à partir des sources

Note : pour éviter tout problème lié au positionnement des variables d'environnement dans les exercices suivants, l'installation depuis les sources se fera avec un utilisateur dédié, différent de l'utilisateur utilisé par l'installation depuis les paquets de la distribution.

Environnement

Vérifier que les logiciels de compilation sont installés

Créer l'utilisateur système `srcpostgres` avec pour `HOME` `/opt/pgsql`.

Se connecter en tant qu'utilisateur `srcpostgres`.

Téléchargement

Consulter le site officiel du projet et relevez la dernière version de PostgreSQL.

Télécharger les fichiers sources de la dernière version et les placer dans `/opt/pgsql/src`.

Compilation et installation

Configurer puis compiler les sources. L'installation devra être réalisée dans `/opt/pgsql/9.6/`

Installer les fichiers obtenus.

Configurer le système

Où se trouvent les binaires installés de PostgreSQL ?

Positionnez correctement les variables d'environnement `PATH` et `LD_LIBRARY_PATH` de l'utilisateur `srcpostgres` pour accéder facilement à ces binaires.

Lancement

Initialiser une instance dans `/opt/pgsql/9.6/data` en spécifiant `postgres` comme nom de super-utilisateur, et la démarrer. Positionnez la variable d'environnement `PGDATA` pour l'utilisateur `srcpostgres`.

Créer une base de données `test` et s'y connecter avec le rôle `postgres`.

Jeu de données

Récupérer le jeu de données permettant d'installer la base de données `cave`.

Installer ce jeu de données à partir de l'utilisateur système `srcpostgres`, en modifiant la variable d'environnement `PGUSER` pour que la connexion se fasse par défaut avec le rôle `postgres`.

Créer un script d'initialisation

Trouver un exemple de script d'initialisation dans les sources téléchargées.

Adapter le script trouvé et l'ajouter au démarrage.

Installation de paquets binaires

Sauvegarde

17.12

La priorité est d'assurer la sécurité des données. Faire une copie de sauvegarde des données insérées lors de l'installation précédente, puis arrêter l'instance installée précédemment.

Pré-installation

Quelle commande permet d'installer les paquets binaires de PostgreSQL ? Quelle version est packagée ? Quels paquets devront également être installés ?

Installation

Exécuter la commande d'installation

Démarrage

Comment démarrer le serveur ?

Emplacement des fichiers

Où ont été placés les fichiers installés? Où se trouvent les fichiers de configuration? Où est le répertoire `$PGDATA` ?

Configuration

Vérifier la configuration par défaut de PostgreSQL. Est-ce que le serveur écoute sur le réseau ?

Migration des données

Vérification

Vérifier que les données de la version compilée peuvent être injectées sans risque dans la version packagée

Restauration

Restaurer les données

3.8.2 SOLUTIONS

Installation à partir des sources

Environnement

Vérifier que les logiciels de compilation et autres dépendances sont installés, et si ce n'est pas le cas les installer. Ces actions doivent être effectuées en tant qu'utilisateur privilégié (soit directement en tant que `root`, soit en utilisant la commande `sudo`).

Sous Debian ou Ubuntu :

```
apt-get install build-essential libreadline-dev zlib1g-dev flex bison \
    libxml2-dev libxslt-dev libssl-dev
```

Sous Fedora, RedHat ou CentOS, il faudra utiliser `yum` ou `rpm`, par exemple :

```
yum install -y bison-devel readline-devel zlib-devel openssl-devel wget
yum groupinstall -y 'Development Tools'
```

En tant que `root`, créer l'utilisateur système `srcpostgres` avec pour `HOME` `/opt/pgsql` :

```
useradd -d /opt/pgsql -m -r -s /bin/bash srcpostgres
```

Se connecter en tant qu'utilisateur `srcpostgres` :

```
su - srcpostgres
```

Téléchargement

Sur le site officiel du projet, télécharger les fichiers sources de la dernière version (PostgreSQL 9.6).

```
mkdir ~srcpostgres/src
cd ~/src
wget http://ftp.postgresql.org/pub/source/v9.6.5/postgresql-9.6.5.tar.bz2
# commande suivante est équivalente à :
# tar --gunzip --extract --verbose --file postgresql-9.6.5.tar.bz2
tar xjvf postgresql-9.6.5.tar.bz2
cd postgresql-9.6.5
```

Compilation et installation

Configurer puis compiler les sources de PostgreSQL :

```
./configure --prefix /opt/pgsql/9.6
make
```

(cette opération peut prendre du temps)

Ensuite installer les fichiers obtenus:

```
make install
```

Dans ce TP, nous nous sommes attachés à changer le moins possible d'utilisateur système. Il se peut que vous ayez à installer les fichiers obtenus en tant qu'utilisateur `root` dans d'autres environnements en fonction de la politique de sécurité adoptée.

Configurer le système

Les binaires installés sont situés dans le répertoire `/opt/pgsql/9.6/bin`.

Adapter les variables d'environnement `PATH` et `LD_LIBRARY_PATH`. Exécuter les commandes suivantes pour ajouter la modification des variables d'environnement à la fin

du fichier ~srcpostgres/.bash_profile (le nom de ce fichier peut être différent selon l'environnement utilisé) :

```
echo 'export PATH=/opt/pgsql/9.6/bin:$PATH' >> ~srcpostgres/.bash_profile
echo 'export LD_LIBRARY_PATH=/opt/pgsql/9.6/lib:$LD_LIBRARY_PATH' >> \
    ~srcpostgres/.bash_profile
```

Il faut ensuite recharger le fichier profile à l'aide de la commande suivante (ne pas oublier le point et l'espace au début de la commande) :

```
. ~srcpostgres/.bash_profile
```

Lancement

Initialiser et démarrer le serveur. Toujours en tant qu'utilisateur srcpostgres :

```
initdb -D /opt/pgsql/9.6/data -U postgres
pg_ctl -D /opt/pgsql/9.6/data start
createdb -U postgres test
psql -U postgres test
```

Positionner la variable d'environnement PGDATA. Exécuter la commande suivante pour ajouter la modification de la variable d'environnement PGDATA à la fin du fichier ~srcpostgres/.bash_profile :

```
echo "export PGDATA=/opt/pgsql/9.6/data" >> ~srcpostgres/.bash_profile
```

Il faut ensuite recharger le fichier profile à l'aide de la commande suivante (ne pas oublier le point et l'espace au début de la commande) :

```
. ~srcpostgres/.bash_profile
```

Jeu de données

En tant que root, récupérer les scripts de création de la base de données cave et copier les dans le répertoire home de l'utilisateur srcpostgres :

```
cp /opt/formation/tp/cave.tgz ~srcpostgres/
chown srcpostgres: ~srcpostgres/cave.tgz
su - srcpostgres
```

Puis, en tant qu'utilisateur srcpostgres décompresser l'archive et exécuter le script de création de la base de données :

```
cd ~
tar zxf cave.tgz
cd cave/
export PGUSER=postgres
./create_db.sh
```

(cette opération peut prendre un peu de temps)

Créer un script d'initialisation

Un script d'initialisation exemple pour linux est disponible dans les sources à l'emplacement suivant : `contrib/start-scripts/linux`. Depuis une session `root`, il faut le copier dans le répertoire des scripts d'initialisation, et lui donner lesdroits d'exécution :

```
cp ~srcpostgres/src/postgresql-9.6.5/contrib/start-scripts/linux \
    /etc/init.d/srcpostgresql
chmod +x /etc/init.d/srcpostgresql
```

Pour adapter ce script, éditez le et modifiez les paramètres suivants: prefix=/opt/pgsql/9.6 PGDATA="/opt/pgsql/9.6/data" PGLOG="$PGDATA/serverlog" PGUSER="srcpostgres"

Pour activer PostgreSQL au démarrage, sous Debian ou Ubuntu exécuter la commande suivante:

```
update-rc.d srcpostgresql defaults
```

Sous RedHat, CentOS ou Fedora exécuter la commande suivante:

```
chkconfig --add srcpostgresql
```

Installation de paquets binaires

Sauvegarde

En tant que `srcpostgres`, sauvegarde de toutes les bases de données dans le fichier `/opt/pgsql/backups/tp.partie1.sql` :

```
su - srcpostgres
mkdir /opt/pgsql/backups
pg_dumpall -U postgres > /opt/pgsql/backups/tp.partie1.sql
```

Arrêt de l'instance :

```
pg_ctl -D $PGDATA stop
```

`PGDATA` étant positionné, la commande suivante est aussi possible:

```
pg_ctl stop
```

On aurait aussi pu faire, en tant que `root` :

```
service srcpostgresql stop
```

Pré-installation

Tout dépend de votre distribution. Le gestionnaire de package de Mandriva est `urpmi` alors que celui de Debian et Ubuntu est `aptitude` ou `apt-get`. Celui de la Fedora, CentOS et autres distributions compatibles RedHat est `yum`, sur Suse on utilisera plutôt `yast` ou `zypper`.

17.12

La dernière version stable disponible de PostgreSQL est la 9.6.5. La dernière version binaire dépend là-aussi de votre distribution.

Le paquet `libpq` devra également être installé.

Installation sous Debian et Ubuntu

Préparer le dépôt :

```
echo \
   "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" \
   > /etc/apt/sources.list.d/pgdg.list
apt-get install wget ca-certificates
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc \
   | sudo apt-key add -
```

Installer les paquets :

```
apt-get update
apt-get install postgresql-9.6 postgresql-client-9.6 postgresql-contrib-9.6
```

Installation sous RedHat, CentOS et Fedora

Préparer le dépôt (par exemple ici pour RHEL 6) :

```
yum install https://download.postgresql.org/pub/repos/yum/9.6/redhat/\
rhel-6-x86_64/pgdg-redhat96-9.6-3.noarch.rpm
```

Installer les paquets :

```
yum install postgresql96 postgresql96-server postgresql96-contrib
service postgresql-9.6 initdb
```

Démarrage

L'instance est démarrée en tant que `root` grâce à la commande suivante sous Debian et Ubuntu :

```
service postgresql start
```

Sous RedHat, CentOS et Fedora :

```
service postgresql-9.6 start
```

Attention, l'instance créée précédemment à partir des sources doit impérativement être arrêtée pour pouvoir démarrer la nouvelle instance !

En effet, comme on n'a pas modifié le port par défaut (5432), les deux instances ne peuvent pas être démarrées en même temps.

Emplacement des fichiers

Les fichiers de configuration de PostgreSQL se trouvent dans le répertoire `PGDATA` (par défaut `/var/lib/pgsql/9.6/data/`) pour une version 9.6 sous RedHat, CentOS et Fedora.

Sous Debian et Ubuntu, ils se trouvent dans le répertoire `/etc/postgresql/9.6/main` pour une version 9.6.

Configuration

On peut vérifier dans le fichier `postgresql.conf` que par défaut, le serveur écoute uniquement l'interface réseau `localhost`. On peut également le constater en utilisant la commande `netstat`.

```
netstat -anp | grep post
```

Migration des données

Vérification

Lire la page de « Release Notes »:

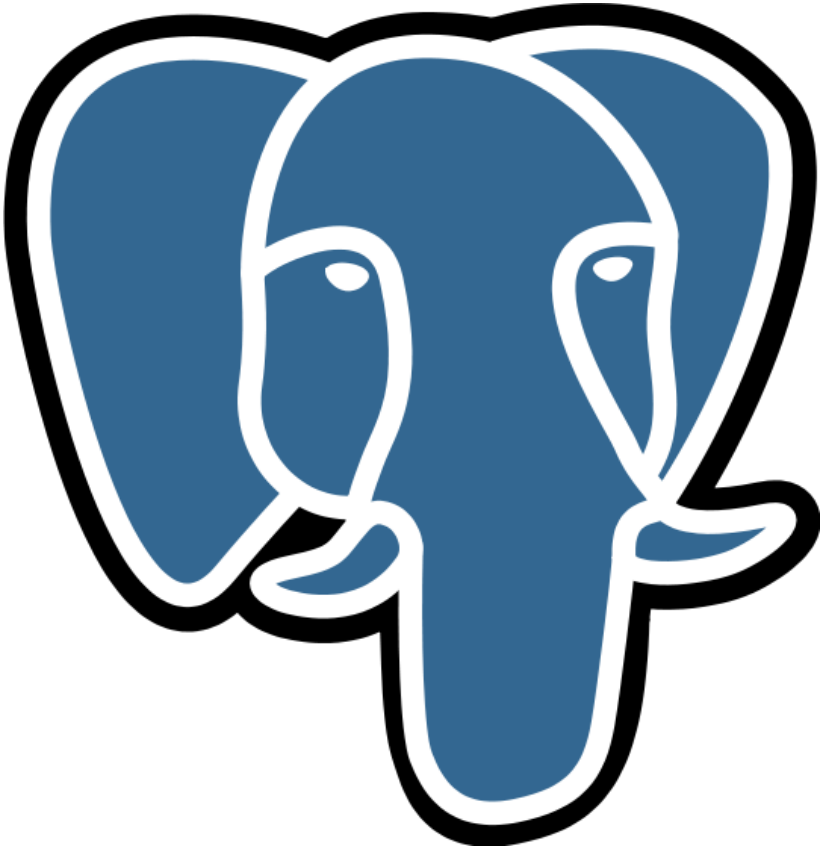
http://www.postgresql.org/docs/current/interactive/release-9-6.html

Restauration

Récupérer en tant que `root` le fichier de sauvegarde réalisé précédemment et restaurer les données dans la nouvelle instance :

```
cp /opt/pgsql/backups/tp.partie1.sql ~postgres/
chown postgres: ~postgres/tp.partie1.sql
su - postgres
psql -f tp.partie1.sql
```

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 ;

- le précompilateur ;
- 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.

L'outil le plus important est la console en mode texte, `psql`. Cet outil permet d'effectuer l'ensemble des activités `DDL` (**Data Definition Language**, instructions **create**, **drop**, **alter**...) et `DML` (**Data Modification Language**, instructions **insert**, **update**, **delete**, **select**...).

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
  - outils de maintenance
  - des outils de sauvegardes/restauration
  - 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
- `createlang`: ajouter un langage de procédures à une base (<v10)
- `createuser`: ajouter un nouveau compte utilisateur
- `dropdb`: supprimer une base de données
- `droplang`: supprimer un langage de procédures (<v10)
- `dropuser`: supprimer un compte utilisateur

Chacune de ces commandes est un « alias », un raccourci qui permet d'exécuter des commandes SQL de manière plus simple.

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 nouvelle base se fait en utilisant l'outil `createdb` et en lui indiquant le nom de la nouvele base. Par exemple, pour créer une base b1, il faudrait faire ceci :

```
$ createdb b1
```

Il est à noter que `createlang` et `droplang` ont été supprimés de la distribution lors de la sortie de la version 10. Un langage s'installe à présent grâce aux extensions.

4.2.2 OUTILS : SAUVEGARDE / RESTAURATION

- Pour une instance
  - `pg_dumpall`: sauvegarder l'instance PostgreSQL
- Pour une base de données
  - `pg_dump`: sauvegarder une base de données
  - `pg_restore`: restaurer une base de données PostgreSQL

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. Pour ne sauvegarder qu'une seule base, il est préférable de passer par l'outil `pg_dump`. Il faut évidemment lui fournir le nom de la base à sauvegarder. L'option `-f` permet de préciser le fichier en sortie. Sans cette option, le résultat va sur la sortie standard. Pour sauvegarder notre base b1, il suffirait de lancer la commande suivante :

```
$ pg_dump -f b1.sql b1
```

Pour la restauration, l'outil habituel est `pg_restore`. Notez aussi que `psql` peut être utilisé pour la restauration si la sauvegarde est en mode texte.

4.2.3 OUTILS : MAINTENANCE

- Maintenance des bases

  - `vacuumdb`: récupérer l'espace inutilisé (`VACUUM FULL`) et/ou mettre à jour les statistiques de l'optimiseur (`ANALYZE`)
  - `reindexdb`: réindexer une base de données PostgreSQL
  - `clusterdb`: réorganiser une table en fonction d'un index
- Maintenance de l'instance
  - `pg_ctl`: lancer, arrêter, relancer, promouvoir le serveur PostgreSQL

Les commandes `reindexdb` et `vacuumdb` doivent être lancées de manière régulière. La fréquence est à déterminer selon l'activité et le volume de chaque base de données. L'autovacuum rend l'outil `vacuumdb` moins important, mais il ne le rend pas obsolète pour autant.

La commande `clusterdb` est très spécifique à l'activité des applications en relation avec les bases.

Pour lancer une réindexation de la base `b1` en affichant la commande exécutée, il suffit de saisir la commande suivante :

```
$ reindexdb -e b1
REINDEX DATABASE b1;
```

4.2.4 OPTIONS CONNEXION

- Ces outils s'appuient sur:
  - des options en ligne de commande
  - des variables d'environnement
  - des valeurs par défaut

Les options de connexion permettent de préciser l'utilisateur et la base de données utilisés pour la connexion initiale.

Lorsque l'une des options n'est pas précisée, la bibliothèque cliente PostgreSQL vérifie qu'il n'existe pas une variable shell correspondante et prend sa valeur. S'il ne trouve pas de variable, il utilise une valeur par défaut.

`-h HOTE` ou `$PGHOST` permet de préciser l'alias ou l'adresse `IP` de la machine qui héberge le serveur. Sans indication, le client se connecte sur la socket Unix dans `/tmp`.

`-p PORT` ou `$PGPORT` permet de préciser le port sur lequel le serveur écoute les connexions entrantes. Sans indication, le port par défaut est le 5432.

`-U NOM` ou `$PGUSER` permet de préciser le nom de l'utilisateur utilisé pour la connexion. L'option `-U` n'est toutefois pas nécessaire, sous réserve que les arguments de la ligne

de commande respectent l'ordre `NOM_BASE NOM_UTILISATEUR`. Sans indication, le nom d'utilisateur PostgreSQL est le nom de l'utilisateur utilisé au niveau système.

`-d base` ou `$PGDATABASE` permet de préciser le nom de la base de données utilisée pour la connexion. Le drapeau `-d` n'est pas nécessaire sous réserve que le nom de la base de données soit le dernier argument de la ligne de commande. Sans indication le nom de la base de données de connexion correspondra au nom de l'utilisateur utilisé au niveau PostgreSQL.

Le tableau suivant résume ce qui est écrit plus haut :

option	variable	défaut
-h HOTE	$PGHOST	/tmp
-p PORT	$PGPORT	5432
-U NOM	$PGUSER	nom de l'utilisateur OS
-d base	$PGDATABASE	nom de l'utilisateur PG

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 se connectera ainsi au premier serveur de la liste qui soit ouvert en écriture et disponible parmi les 3 précisés :

```
psql -h serveur1,serveur2,serveur3 -p 5432,5433,5434
     target_session_attrs=read-write
     -U dupont mabase
```

ou ainsi

```
psql "host=serveur1,serveur2,serveur3 -p 5432,5433,5434
      target_session_attrs=read-write
      user=dupont" mabase
```

Toutes ces variables, ainsi que de nombreuses autres, sont documentées ici[62] .

4.2.5 AUTHENTIFICATION D'UN CLIENT

- En interactif

[62] http://docs.postgresql.fr/current/libpq-envars.html

  - `-W | --password`
  - `-w | --no-password`
- Variable `$PGPASSWORD`
- Fichier `.pgpass`
  - chmod 600 .pgpass
  - `nom_hote:port:database:nomutilisateur:motdepasse`

Options -W et -w

`-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.

`-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. Cette option apparaît la première fois avec la version 8.4.

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`. Si les droits du fichier sont moins stricts, le

fichier sera ignoré.

Les droits du fichier ne sont actuellement pas vérifiés sur *Microsoft Windows* .

4.3 LA CONSOLE PSQL

- Un outil simple pour
  - les opérations courantes,
  - les tâches de maintenance,
  - 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.

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…

4.3.1 OBTENIR DE L'AIDE ET QUITTER

- Obtenir de l'aide sur les commandes internes psql
  - `\? [motif]`
- Obtenir de l'aide sur les ordres SQL
  - `\h [motif]`
- Quitter
  - `\q`

`\h [NOM]` affiche l'aide en ligne des commandes SQL. Sans argument, la liste des commandes disponibles est affichée.

Exemple :

```
postgres=# \h savepoint
Command:     SAVEPOINT
Description: define a new savepoint within the current transaction
Syntax:
SAVEPOINT savepoint_name
```

`\q` permet de quitter la console.

4.3.2 GESTION DE LA CONNEXION

- Spécifier le jeu de caractère
  - `\encoding [ENCODING]`
- Modifier le mot de passe d'un utilisateur
  - `\password [USERNAME]`
- Obtenir des informations sur la connexion courante:
  - `\conninfo`
- Se connecter à une autre base
  - `\connect [DBNAME|- USER|- HOST|- PORT|-]`
  - `\c [DBNAME|- USER|- HOST|- PORT|-]`

`\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
```

`\c` permet de changer d'utilisateur et/ou de base de données sans quitter le client.

Exemple :

```
postgres@serveur_pg:~$ psql -q postgres
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".
```

17.12

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 USER` ou un `ALTER USER` (à moins de chiffrer soit-même le mot de passe).

4.3.3 GESTION DE L'ENVIRONNEMENT SYSTÈME

- Chronométrer les requêtes
  - `\timing`
- Exécuter une commande OS
  - `\! [COMMAND]`
- Changer de répertoire courant
  - `\cd [DIR]`

`\timing` active le chronométrage des commandes. Il accepte un argument indiquant la nouvelle valeur (soit on, soit off). Sans argument, la valeur actuelle est basculée.

`\! [COMMANDE]` ouvre un shell interactif en l'absence d'argument ou exécute la commande indiquée.

`\cd` permet de changer de répertoire courant. Cela peut s'avérer utile lors de lectures ou écritures sur disque.

4.3.4 CATALOGUE SYSTÈME: OBJETS UTILISATEURS

- Lister les bases de données
  - `\l`
- Lister les schémas
  - `\dn`
- Lister uniquement les tables|index|séquences|vues|vues matérialisées [systèmes]
  - `\d{t|i|s|v|m}[S][+] [motif]`
- Lister les fonctions
  - `\df[+] [motif]`
- Lister des fonctions d'agrégats
  - `\da [motif]`

Ces commandes permettent d'obtenir des informations sur les objets utilisateurs: tables, index, vues, séquences, fonctions, agrégats, etc. stockés dans la base de données. Ces

commandes listent à la fois les objets utilisateur et les objets système pour les versions antérieures à la 8.4.

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.

`\l[+]` dresse la liste des bases de données sur le serveur. Avec `+`, les commentaires et les tailles des bases sont également affichés.

`\d[+] [motif]` permet d'afficher la liste des tables de la base lorsqu'aucun motif n'est indiqué. Dans le cas contraire, la table précisée est décrite. Le `+` permet d'afficher également les commentaires associés aux tables ou aux lignes de la table, ainsi que la taille de chaque table.

`\db [motif]` dresse la liste des tablespaces actifs sur le serveur.

`\d{t|i|s|v}[S] [motif]` permet d'afficher respectivement :

- la liste des tables de la base active ;
- la liste des index ;
- la liste des séquences ;
- la liste des vues ;
- la liste des tables systèmes.

`\da` dresse la liste des fonctions d'agrégats.

`\df` dresse la liste des fonctions.

Exemple :

```
postgres=# \c cave
You are now connected to database "cave" as user "postgres".
cave=# \d appel*
                              Table "public.appellation"
  Column   |  Type   | Collation | Nullable |            Default
-----------+---------+-----------+----------+-------------------------------
 id        | integer |           | not null | nextval(
           |         |           |          |  'appellation_id_seq'::regclass)
 libelle   | text    |           | not null |
 region_id | integer |           |          |
Indexes:
    "appellation_pkey" PRIMARY KEY, btree (id)
    "appellation_libelle_key" UNIQUE CONSTRAINT, btree (libelle)
Foreign-key constraints:
    "appellation_region_id_fkey" FOREIGN KEY (region_id)
```

```
     REFERENCES region(id) ON DELETE CASCADE
Referenced by:
    TABLE "vin" CONSTRAINT "vin_appellation_id_fkey"
    FOREIGN KEY (appellation_id) REFERENCES appellation(id) ON DELETE CASCADE

Sequence "public.appellation_id_seq"
   Column    |  Type   | Value
-------------+---------+-------
 last_value | bigint  | 1
 log_cnt    | bigint  | 0
 is_called  | boolean | f
Owned by: public.appellation.id

Index "public.appellation_libelle_key"
 Column  | Type | Definition
---------+------+------------
 libelle | text | libelle
unique, btree, for table "public.appellation"

Index "public.appellation_pkey"
 Column |  Type   | Definition
--------+---------+------------
 id     | integer | id
primary key, btree, for table "public.appellation"

cave=# \da sum
                            List of aggregate functions
  Schema   |Name| Result     | Argument     |      Description
           |    | data type  | data types   |
-----------+----+------------+--------------+-------------------------------------
pg_catalog |sum | numeric    | bigint       | sum as numeric
           |    |            |              | across all bigint input values
pg_catalog |sum | double     | double       | sum as float8
           |    | precision  | precision    | across all float8 input values
pg_catalog |sum | bigint     | integer      | sum as bigint
           |    |            |              | across all integer input values
pg_catalog |sum | interval   | interval     | sum as interval
           |    |            |              | across all interval input values
pg_catalog |sum | money      | money        | sum as money
```

```
           |    |           |             | across all money input values
pg_catalog |sum | numeric   | numeric     | sum as numeric
           |    |           |             | across all numeric input values
pg_catalog |sum | real      | real        | sum as float4
           |    |           |             | across all float4 input values
pg_catalog |sum | bigint    | smallint    | sum as bigint
           |    |           |             | across all smallint input values
(8 rows)

cave=# \df public.*
                                List of functions
 Schema|     Name     | Result    | Argument data types   |  Type
       |              | data type |                       |
-------+--------------+-----------+-----------------------+--------
public | peuple_stock | bigint    | annee_debut integer,  | normal
       |              |           | annee_fin integer     |
public | peuple_vin   | bigint    |                       | normal
public | trous_stock  | bigint    |                       | normal
public | trous_vin    | bigint    |                       | normal
(4 rows)
```

4.3.5 CATALOGUE SYSTÈME: RÔLES ET ACCÈS

- Lister les rôles
  - `\du[+]`
- Lister les droits d'accès
  - `\dp`
- Lister les droits d'accès par défaut
  - `\ddp`
- Lister les configurations par rôle et par base
  - `\drds`

L'ensemble des informations concernant les objets, les utilisateurs, les procédures... sont accessibles par des commandes internes débutant par `\d`.

Pour connaître les rôles stockés en base, cette commande est `\du` (u pour user) ou `\dg` (g pour group). Dans les versions antérieures à la 8.0, les groupes et les utilisateurs étaient deux notions distinctes. Elles sont aujourd 'hui regroupées dans une notion plus générale, les rôles.

Les droits sont accessibles par les commandes \dp (p pour permissions) ou \z .

Exemple :

```
cave=# \du
                   List of roles
 Role name  |         Attributes                   |     Member of
------------+--------------------------------------+--------------------
 admin      |                                      | {}
 caviste    |                                      | {}
 postgres   | Superuser, Create role, Create DB,   | {}
            | Replication, Bypass RLS              |
 stagiaire2 |                                      | {}
 u1         |                                      | {pg_signal_backend}
 u2         |                                      | {}
```

```
cave=# \z
                              Access privileges
Schema|        Name        |  Type   | Access privileges  |  Column privileges    |
------+--------------------+---------+--------------------+-----------------------+
public| appellation        | table   |                    |                       |
public| appellation_id_seq | sequence|                    |                       |
public| contenant          | table   |                    |                       |
public| contenant_id_seq   | sequence|                    |                       |
public| recoltant          | table   |                    |                       |
public| recoltant_id_seq   | sequence|                    |                       |
public| region             | table   |caviste=            |                       |
      |                    |         |arwdDxt/caviste+    |                       |
      |                    |         |stagiaire2=r/caviste|                       |
public| region_id_seq      | sequence|                    |                       |
public| stock              | table   |                    |vin_id:               +|
      |                    |         |                    |  stagiaire2=r/caviste+|
      |                    |         |                    |contenant_id:         +|
      |                    |         |                    |  stagiaire2=r/caviste+|
      |                    |         |                    |annee:                +|
      |                    |         |                    |  stagiaire2=r/caviste |
public| type_vin           | table   |                    |                       |
public| type_vin_id_seq    | sequence|                    |                       |
public| vin                | table   |                    |                       |
public| vin_id_seq         | sequence|                    |                       |
```

```
(13 rows)
```

La commande \ddp permet de connaître les droits accordés par défaut à un utilisateur sur les nouveaux objets avec l'ordre ALTER DEFAULT PRIVILEGES.

```
cave=# \ddp
             Default access privileges
  Owner   | Schema | Type  |    Access privileges
----------+--------+-------+-------------------------
 caviste |        | table | caviste=arwdDxt/caviste+
         |        |       | u1=r/caviste
(1 row)
```

Enfin, la commande \drds permet d'obtenir la liste des paramètres appliqués spécifiquement à un utilisateur ou une base de données.

```
cave=# \drds
                List of settings
  Role   | Database |           Settings
---------+----------+----------------------------
 caviste |          | maintenance_work_mem=256MB
         | cave     | work_mem=32MB
(2 rows)
```

4.3.6 CATALOGUE SYSTÈME: TABLESPACES ET EXTENSIONS

- Lister les tablespaces
  - \db
- Lister les extensions
  - \dx

\db [motif] dresse la liste des tablespaces actifs sur le serveur.

```
postgres=# \db
              List of tablespaces
    Name    |  Owner   |      Location
------------+----------+---------------------
 pg_default | postgres |
 pg_global  | postgres |
 ts1        | postgres | /tmp/tmp.fbdHJIa3jP
(3 rows)
```

17.12

`\dx [motif]` dresse la liste des extensions installées dans la base courante :

```
postgres=# \dx
                          List of installed extensions
        Name         | Version |   Schema   |                Description
---------------------+---------+------------+-----------------------------------
 pg_stat_statements | 1.5     | public     | track execution statistics
                    |         |            | of all SQL statements executed
 plpgsql            | 1.0     | pg_catalog | PL/pgSQL procedural language
 unaccent           | 1.1     | public     | text search dictionary
                    |         |            | that removes accents
(3 rows)
```

4.3.7 CATALOGUE SYSTÈME: AUTRES OBJETS

- Type, domaines et opérateurs
  - `\dD[S+]`
  - `\dT[S+]`
  - `\do [motif]`
- Lister les objets FTS
  - `\dF [motif]`
  - `\dFd [motif]`
  - `\dFt [motif]`
  - `\dFp [motif]`
- Conversions
  - `\dc [motif]`
  - `\dC [motif]`

Types, domaines et opérateurs

`\dD [motif]` dresse la liste de tous les domaines disponibles.

`\dT[+] [motif]` dresse la liste des types de données disponibles.

`\do [motif]` dresse la liste des opérateurs disponibles.

Exemple :

```
postgres=# \dD
                  List of domains
Schema|    Name      |Type|Col.|Null.|Def.|           Check
------+--------------+----+----+-----+----+-----------------------------
```

```
public|code_postal_fr|text|    |     |    |CHECK (VALUE ~ '^\d{5}$'::text)
(1 row)

postgres=# \x on
Expanded display is on.
postgres=# \dT+ bigint
List of data types
-[ RECORD 1 ]-----+----------------------------------
Schema            | pg_catalog
Name              | bigint
Internal name     | int8
Size              | 8
Elements          |
Owner             | postgres
Access privileges |
Description       | ~18 digit integer, 8-byte storage
```

Objets FTS

`\dF [motif]` dresse la liste des configurations de la recherche plein texte.

`\dFd[+] [motif]` dresse la liste des dictionnaires de la recherche plein texte. `+` permet d'afficher également le modèle et les options d'initialisation des dictionnaires.

`\dFt[+] [motif]` dresse la liste des modèles de la recherche plein texte. `+` permet d'afficher également la fonction d'initialisation et celle de récupération des lexemes.

`\dFp [motif]` dresse la liste des analyseurs de la recherche plein texte.

Conversions

`\dc` dresse la liste des conversions de jeux de caractères.

'dC' dresse la liste des conversions de type de données disponibles.

Exemple

```
postgres=# \dc iso_8859_15_to_utf8
                        List of conversions
   Schema   |        Name         | Source | Destination | Default?
------------+---------------------+--------+-------------+----------
 pg_catalog | iso_8859_15_to_utf8 | LATIN9 | UTF8        | yes
(1 row)

postgres=# \dC
```

17.12

```
                              List of casts
 Source type |         Target type          |      Function      |   Implicit?
-------------+------------------------------+--------------------+---------------
 "char"      | character                    | bpchar             | in assignment
 "char"      | character varying            | text               | in assignment
 "char"      | integer                      | int4               | no
 "char"      | text                         | text               | yes
 abstime     | date                         | date               | in assignment
 abstime     | integer                      | (binary coercible) | no
[...]
```

4.3.8 VISUALISER LE CODE DES OBJETS

- 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 :

```
cave=# \sf trous_vin
CREATE OR REPLACE FUNCTION public.trous_vin()
 RETURNS bigint
 LANGUAGE plpgsql
AS $function$
DECLARE
  vin_total integer;
  echantillon integer;
  v_vin_id integer;
  v_tuples bigint := 0;
  v_annee integer;
BEGIN

  -- on compte le nombre de tuples dans vin
  select count(*) from vin into vin_total;
  raise NOTICE '% vins disponibles', vin_total;

  -- on calcule la taille de l'echantillon a
```

```
  -- supprimer de la table vin
  select round(vin_total/10) into echantillon;
  raise NOTICE 'taille de l''echantillon %', echantillon;

  -- on fait une boucle correspondant a 10% des tuples
  -- de la table vin
  for v_tuples in 1 .. echantillon loop

     -- selection d'identifiant, au hasard
     v_vin_id := round(random()*vin_total);

     -- si le tuple est deja efface, ce n'est pas grave..

     -- TODO remplacer ce delete par un trigger on delete cascade
     --      voir dans druid le schema???
     delete
     from stock
     where vin_id = v_vin_id;

     delete
     from vin
     where id = v_vin_id;

     if (((v_tuples%100)=0) or (v_tuples=echantillon)) then
        raise notice 'vin : % sur % echantillon effaces',v_tuples, echantillon;
     end if;

  end loop; --fin boucle v_tuples

  RETURN echantillon;

END;
$function$
```

4.3.9 EXÉCUTER DES REQUÊTES

- Exécuter une requête
  - terminer une requête par ;

  - ou par \g
  - ou encore par \gx
- Rappel des requêtes:
  - flèche vers le haut
  - ctrl-R suivi d'un extrait de texte représentatif

Sauf si psql est exécuté avec l'option -S (mode *single-line*), toutes les requêtes SQL doivent se terminer par ; ou, pour marquer la parenté de PostgreSQL avec Ingres, \g.

En version 10, il est aussi possible d'utiliser \gx pour avoir l'action conjuguée de \g (ré-exécution de la requête) et de \x (affichage étendu).

La console psql, lorsqu'elle est compilée avec la bibliothèque libreadline ou la bibliothèque libedit, dispose des mêmes possibilités de rappel de commande que le shell bash.

Exemple

```
postgres=# SELECT * FROM pg_tablespace LIMIT 5;
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
 ts1        |       10 |        |
(3 rows)

postgres=# SELECT * FROM pg_tablespace LIMIT 5\g
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
 ts1        |       10 |        |
(3 rows)

postgres=# \g
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
 ts1        |       10 |        |
(3 rows)

postgres=# \gx
```

```
-[ RECORD 1 ]----------
spcname    | pg_default
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 2 ]----------
spcname    | pg_global
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 3 ]----------
spcname    | ts1
spcowner   | 10
spcacl     |
spcoptions |
```

4.3.10 EXÉCUTER LE RÉSULTAT D'UNE REQUÊTE

- Exécuter le résultat d'une requête
  - `\gexec`
- Apparaît en 9.6

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 :

```
cave=# 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  | type_vin
 public  | contenant
 public  | recoltant
 public  | region
 public  | appellation
 public  | vin
 public  | stock
```

```
(7 rows)
```

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 :

```
cave=# 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.type_vin
 VACUUM public.contenant
 VACUUM public.recoltant
 VACUUM public.region
 VACUUM public.appellation
 VACUUM public.vin
 VACUUM public.stock
(7 rows)
```

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` :

```
postgres=# \gexec
VACUUM
VACUUM
VACUUM
VACUUM
VACUUM
VACUUM
VACUUM
```

4.3.11 MANIPULER LE TAMPON DE REQUÊTES

- Éditer
  - dernière requête : `\e`
  - vue : `\ev nom_vue`
  - fonction PL/pgSQL : `\ef nom_fonction`
- Exécuter le contenu du tampon

  - `\g [FICHIER]`
- Afficher le tampon
  - `\p`
- Sauvegarder le contenu du tampon
  - `\w [FICHIER]`
- Supprimer le contenu du tampon
  - `\r`

`\e [FICHIER]` édite le tampon de requête courant ou le fichier indiqué à l' aide d'un éditeur externe.

`\g [FICHIER]` envoie le tampon de requête au serveur et, en présence d'un argument, le résultat au fichier indiqué.

`\p` affiche le contenu du tampon de requête.

`\r` supprime le contenu du tampon de requête.

`\w FICHIER` provoque l'écriture du tampon de requête dans le fichier indiqué.

`\ev [NOMVUE]` édite la vue indiquée. Sans argument, entre en mode édition avec la requête de déclaration d'une vue. Cette méta-commande est disponible à partir de la version 9.6.

`\ef [NOMFONCTION]` édite la fonction indiquée. Sans argument, entre en mode édition avec la requête de déclaration d'une fonction. Cette méta-commande est disponible à partir de la version 8.4.

`\s [FICHIER]` affiche l'historique des commandes effectuées lors de la session, en l'absence d'argument. Si un fichier est précisé, l'historique des commandes y est sauvegardé.

4.3.12 ENTRÉES/SORTIES

- Charger et exécuter un script SQL
  - `\i FICHIER`
- Rediriger la sortie dans un fichier
  - `\o FICHIER`
- Écrire un texte sur la sortie standard
  - `\echo texte...`
- Écrire un texte dans le fichier
  - `\qecho texte...`

`\i FICHIER` 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 [FICHIER | |COMMANDE]` envoie les résultats de la requête vers le fichier indiqué ou vers la commande UNIX au travers du tube.

Exemple :

```
postgres=# \o |a2ps
postgres=# SELECT ... ;
```

`\echo [TEXTE]` affiche le texte passé en argument sur la sortie standard.

`\qecho [TEXTE]` offre le même fonctionnement que `\echo [TEXTE]`, à ceci près que la sortie est dirigée sur la sortie des requêtes (fixée par `\o [ FICHIER]`) et non sur la sortie standard.

4.3.13 VARIABLES INTERNES PSQL

- Positionner des variables internes
  - `\set [NOM [VALEUR]]`
- Invalider une variable interne
  - `\unset NOM`
- Variables internes usuelles
  - `ON_ERROR_STOP`: `on` ou `off`
  - `ON_ERROR_ROLLBACK`: `on`, `off` ou `interactive`
  - `AUTOCOMMIT`: `on` ou `off`
- Liste des variables: http://docs.postgresql.fr/current/app-psql.html#app-psql-variables

`\set [NOM [VALEUR]]` affiche les variables internes lorsqu'il est utilisé sans argument. Avec un argument, il permet d'initialiser une variable interne.

Exemple :

```
postgres=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'cave'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
```

```
FETCH_COUNT = '0'
HISTCONTROL = 'none'
HISTSIZE = '500'
HOST = '/tmp'
IGNOREEOF = '0'
LASTOID = '0'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5436'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
QUIET = 'off'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 10beta3 on x86_64-pc-linux-gnu,
           compiled by gcc (GCC) 7.1.1 20170622 (Red Hat 7.1.1-3), 64-bit'
```

Les variables `ON_ERROR_ROLLBACK` et `ON_ERROR_STOP` sont discutées dans la partie relative à la gestion des erreurs.

4.3.14 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.

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
```

17.12

```
\if :est_standby
    \set PROMPT1 'standby %x$ '
\else
    \set PROMPT1 'primaire %x$ '
\endif
```

Puis, en lançant psql sur un serveur primaire, on obtient :

```
psql (10beta3)
Type "help" for help.

primaire $
```

alors qu'on obtient sur un serveur secondaire :

```
psql (10beta3)
Type "help" for help.

standby $
```

NB : Cette fonctionnalité étant liée au client `psql`, elle est disponible même si le serveur n'est pas en version 10.

4.3.15 PERSONNALISER PSQL

- psql est personnalisable
- Au démarrage, psql lit dans le `${HOME}`
  - `.psqlrc-X.Y`
  - `.psqlrc-X`
  - `.psqlrc`
- `.psqlrc` contient des méta-commandes `\set`
  - `\set ON_ERROR_ROLLBACK interactive`

La console psql est personnalisable par le biais de plusieurs variables internes. Il est possible de pérenniser ces personnalisations par le biais d'un fichier `.psqlrc`.

Exemple:

```
$ cat .psqlrc
\timing
\set ON_ERROR_ROLLBACK interactive
$ psql postgres
```

```
Timing is on.
psql (10beta3)
Type "help" for help.

postgres=# SELECT relname FROM pg_class LIMIT 1;
    relname
----------------
 pg_toast_16401
(1 row)

Time: 0.580 ms
```

4.4 ÉCRITURE DE SCRIPTS SHELL

- Script SQL
- Script Shell
- Exemple sauvegarde

4.4.1 EXÉCUTER UN SCRIPT SQL AVEC PSQL

- Exécuter un seul ordre SQL
  - `-c "ordre SQL"`
- Spécifier un script SQL en ligne de commande
  - `-f nom_fichier.sql`
- Possible de les spécifier plusieurs fois
  - exécutés dans l'ordre d'apparition
  - à partir de la version 9.6
- Charger et exécuter un script SQL depuis psql
  - `\i nom_fichier.sql`

L'option `-c` permet de spécifier la requête SQL en ligne de commande. Lorsque ce paramètre est utilisé, il implique automatiquement l'option `--no-psqlrc` jusqu'à la version 9.6.

Il est cependant souvent préférable de les enregistrer dans des fichiers si on veut les exécuter plusieurs fois sans se tromper. L'option `-f` est très utile dans ce cas.

4.4.2 GESTION DES TRANSACTIONS

- psql est en mode auto-commit par défaut
  - variable `AUTOCOMMIT`
- Ouvrir une transaction explicitement
  - `BEGIN;`
- Terminer une transaction
  - `COMMIT;`
- Ouvrir une transaction implicitement
  - option `-1` ou `--single-transaction`

Par défaut, `psql` est en mode auto-commit, 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;`.

Une autre possibilité est de demander à `psql` d'ouvrir une transaction avant le début de l'exécution du script et de faire un `COMMIT` explicite à la fin de l'exécution du script, ou un `ROLLBACK` explicite le cas échéant. La présence d'ordres `BEGIN`, `COMMIT` ou `ROLLBACK` modifiera le comportement de `psql` en conséquence.

4.4.3 ÉCRIRE UN SCRIPT SQL

- Attention à l'encodage des caractères
  - `\encoding`
  - SET client_encoding
- Écriture des requêtes
- Écrire du code procédural avec DO

`\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`.

Une requête se termine par le caractère ; (ou par \g mais qui est peu recommandé car non standard). 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.

Exemple :

```
INSERT INTO appellation (id, libelle, region_id) VALUES (1, 'Ajaccio', 1);
INSERT INTO appellation (id, libelle, region_id) VALUES (2, 'Aloxe-Corton', 2);
INSERT INTO appellation (id, libelle, region_id) VALUES
    (3, 'Alsace Chasselas ou Gutedel', 3);

SELECT v.id AS id_vin, a.libelle, r.nom, r.adresse
  FROM vin v
  JOIN appellation a
    ON (v.appellation_id = a.id)
  JOIN recoltant r
    ON (v.recoltant_id = r.id)
 WHERE libelle = :'appellation';
```

Écrire du code procédural avec DO permet l'exécution d'un bloc de code anonyme, autrement dit une fonction temporaire.

Le bloc de code est traité comme le corps d'une fonction sans paramètre et renvoyant void. Il est analysé et exécuté une seule fois.

Exemple :

Ajouter d'une colonne `status` dans des tables de traces.

```
DO $$DECLARE r record;
BEGIN
  FOR r IN SELECT table_schema, table_name FROM information_schema.tables
    WHERE table_type = 'BASE TABLE' AND table_schema = 'public'
    AND table_name ~ '^logtrace_20.*'
  LOOP
    EXECUTE 'ALTER TABLE ' || quote_ident(r.table_schema) || '.'
      || quote_ident(r.table_name) ||
      ' ADD COLUMN status text DEFAULT ''NON TRAITE''';
  END LOOP;
END$$;
```

4.4.4 UTILISER DES VARIABLES

- Positionner des variables

```
\set nom_table 'ma_table'
SELECT * FROM :"nom_table";
\set valeur_col1 'test'
SELECT * FROM :"nom_table" WHERE col1 = :'valeur_col1';
```

- Demander la valeur d'une variable à l'utilisateur
  - `\prompt 'invite' nom_variable`
- Retirer la référence à une variable
  - `\unset variable`

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`).

Exemple:

```
psql cave -U caviste

cave=# \set appellation 'Alsace Gewurztraminer'

cave=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'cave'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
FETCH_COUNT = '0'
HISTCONTROL = 'none'
HISTSIZE = '500'
HOST = '/tmp'
IGNOREEOF = '0'
LASTOID = '0'
ON_ERROR_ROLLBACK = 'interactive'
ON_ERROR_STOP = 'off'
PORT = '5436'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
QUIET = 'off'
```

```
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 10beta3 on x86_64-pc-linux-gnu,
           compiled by gcc (GCC) 7.1.1 20170622 (Red Hat 7.1.1-3), 64-bit'
appellation = 'Alsace Gewurztraminer'

cave=# SELECT v.id AS id_vin, a.libelle, r.nom, r.adresse
cave-#   FROM vin v
cave-#   JOIN appellation a
cave-#     ON (v.appellation_id = a.id)
cave-#   JOIN recoltant r
cave-#     ON (v.recoltant_id = r.id)
cave-#  WHERE libelle = :'appellation';

 id_vin |        libelle        |            nom            |    adresse
--------+-----------------------+---------------------------+--------------
     10 | Alsace Gewurztraminer | Mas Daumas Gassac         | 34150 Aniane
     11 | Alsace Gewurztraminer | Mas Daumas Gassac         | 34150 Aniane
     12 | Alsace Gewurztraminer | Mas Daumas Gassac         | 34150 Aniane
[...]
(20 rows)

cave=# \prompt appellation
Ajaccio
cave=# SELECT v.id AS id_vin, a.libelle, r.nom, r.adresse
  FROM vin v
  JOIN appellation a
    ON (v.appellation_id = a.id)
  JOIN recoltant r
    ON (v.recoltant_id = r.id)
 WHERE libelle = :'appellation';
 id_vin | libelle |            nom            |    adresse
--------+---------+---------------------------+-------------
      1 | Ajaccio | Mas Daumas Gassac         | 34150 Aniane
      2 | Ajaccio | Mas Daumas Gassac         | 34150 Aniane
      3 | Ajaccio | Mas Daumas Gassac         | 34150 Aniane
```

```
[...]
(20 rows)

cave=# \q

$ cat cave2.sql
SELECT v.id AS id_vin, a.libelle, r.nom, r.adresse
  FROM vin v
  JOIN appellation a
    ON (v.appellation_id = a.id)
  JOIN recoltant r
    ON (v.recoltant_id = r.id)
 WHERE libelle = :'appellation';

$ psql cave -f cave2.sql -v appellation='Ajaccio'
 id_vin | libelle |            nom            |   adresse
--------+---------+---------------------------+-------------
      1 | Ajaccio | Mas Daumas Gassac         | 34150 Aniane
(...)
```

4.4.5 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.

4.4.6 FORMATAGE DES RÉSULTATS

- Afficher des résultats non alignés
  - `-A | --no-align`
- Afficher uniquement les lignes
  - `-t | --tuples-only`
- Utiliser le format étendu
  - `-x | --expanded`
- Utiliser une sortie au format HTML
  - `-H | --html`
- Positionner un attribut de tableau HTML
  - `-T TEXT | --table-attr TEXT`

`-A` impose une sortie non alignée des données.

Exemple :

```
postgres@serveur_pg:~$ psql
postgres=# \l
                                  List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 b1        | postgres | UTF8     | C       | C     |
 cave      | caviste  | UTF8     | C       | C     |
 module_C1 | postgres | UTF8     | C       | C     |
 postgres  | postgres | UTF8     | C       | C     |
 prod      | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(7 rows)

postgres@serveur_pg:~$ psql -A
postgres=# \l
List of databases
Name|Owner|Encoding|Collate|Ctype|Access privileges
b1|postgres|UTF8|C|C|
cave|caviste|UTF8|C|C|
module_C1|postgres|UTF8|C|C|
postgres|postgres|UTF8|C|C|
```

```
prod|postgres|UTF8|C|C|
template0|postgres|UTF8|C|C|=c/postgres
postgres=CTc/postgres
template1|postgres|UTF8|C|C|=c/postgres
postgres=CTc/postgres
(7 rows)
```

`-H` impose un affichage HTML du contenu des tables.

Exemple :

```
postgres@serveur_pg:~$ psql -H
postgres=# \l
<table border="1">
  <caption>List of databases</caption>
  <tr>
    <th align="center">Name</th>
    <th align="center">Owner</th>
    <th align="center">Encoding</th>
    <th align="center">Collate</th>
    <th align="center">Ctype</th>
    <th align="center">Access privileges</th>
  </tr>
(...)
  <tr valign="top">
    <td align="left">template1</td>
    <td align="left">postgres</td>
    <td align="left">UTF8</td>
    <td align="left">C</td>
    <td align="left">C</td>
    <td align="left">=c/postgres<br />
postgres=CTc/postgres</td>
  </tr>
</table>
<p>(7 rows)<br />
</p>
```

`-T TEXT` permet de définir les attributs des balises HTML de table. L'argument passé est ajouté dans la ligne `<table border="1" ... >`. Cette option n'a d'intérêt qu'utilisée conjointement avec `-H`.

`-t` permet de n'afficher que les lignes, sans le nom des colonnes.

Exemple :

```
postgres@wanted:~$ psql -t
postgres=# \l
 b1         | postgres | UTF8     | C       | C     |
 cave       | caviste  | UTF8     | C       | C     |
 module_C1  | postgres | UTF8     | C       | C     |
 postgres   | postgres | UTF8     | C       | C     |
 prod       | postgres | UTF8     | C       | C     |
 template0  | postgres | UTF8     | C       | C     | =c/postgres          +
            |          |          |         |       | postgres=CTc/postgres
 template1  | postgres | UTF8     | C       | C     | =c/postgres          +
            |          |          |         |       | postgres=CTc/postgres
```

`-x` provoque un affichage étendu des informations. Chaque ligne de la table est affichée séparément sous la forme NOM - VALEUR.

Exemple :

```
$ psql -x
postgres=# \l
List of databases
-[ RECORD 1 ]-----+----------------------
Name              | b1
Owner             | postgres
Encoding          | UTF8
Collate           | C
Ctype             | C
Access privileges |
-[ RECORD 2 ]-----+----------------------
Name              | cave
Owner             | caviste
Encoding          | UTF8
Collate           | C
Ctype             | C
Access privileges |
-[ RECORD 3 ]-----+----------------------
Name              | module_C1
Owner             | postgres
Encoding          | UTF8
Collate           | C
```

```
Ctype             | C
Access privileges |
[...]
```

`-P VAR[=ARG]` permet de préciser différentes options de sortie. Chaque couple variable-valeur est regroupé par un signe égal (VAR=ARG).

4.4.7 SÉPARATEURS DE CHAMPS

- Modifier le séparateur de colonnes
  - `-F CHAINE | --field-separator CHAINE`
- Forcer un octet 0x00 comme séparateur de colonnes
  - `-z | --field-separator-zero`
- Modifier le séparateur de lignes
  - `-R CHAINE | --record-separator CHAINE`
- Forcer un octet 0x00 comme séparateur de lignes
  - `-0 | --record-separator-zero`

`-F CHAINE` permet de modifier le séparateur de champ. Par défaut, il s'agit du caractère '|'. Cette option ne fonctionne qu'utilisée conjointement au modificateur de non-alignement des champs.

Exemple :

```
postgres@serveur_pg:~$ psql -F';' -A
postgres=# \l
List of databases
Name;Owner;Encoding;Collate;Ctype;Access privileges
b1;postgres;UTF8;C;C;
cave;caviste;UTF8;C;C;
module_C1;postgres;UTF8;C;C;
postgres;postgres;UTF8;C;C;
prod;postgres;UTF8;C;C;
template0;postgres;UTF8;C;C;=c/postgres
postgres=CTc/postgres
template1;postgres;UTF8;C;C;=c/postgres
postgres=CTc/postgres
(7 rows)
```

`-R CHAINE` permet de modifier le séparateur d'enregistrements. il s'agit par défaut du

retour chariot. Ce commutateur ne fonctionne qu'utilisé conjointement au mode non-aligné de sortie des enregistrements.

Exemple :

```
postgres@serveur_pg:~$ psql -R' #Mon_séparateur# ' -A
postgres=# \l
List of databases #Mon_séparateur# Name|Owner|Encoding|Collate|Ctype|
Access privileges #Mon_séparateur# b1|postgres|UTF8|C|C| #Mon_séparateur# cave|
caviste|UTF8|C|C| #Mon_séparateur# module_C1|postgres|UTF8|C|C| #Mon_séparateur#
postgres|postgres|UTF8|C|C| #Mon_séparateur# template0|postgres|UTF8|C|C|
=c/postgres postgres=CTc/postgres #Mon_séparateur# template1|postgres|UTF8|C|C|
=c/postgres postgres=CTc/postgres #Mon_séparateur# (7 rows)
```

Les options `-z` et `-0` modifient le caractère nul comme séparateur de colonne et de ligne.

4.4.8 PIVOTAGE DES RÉSULTATS

- `\crosstabview [ colV [ colH [ colD [ sortcolH ] ] ] ]`
- Exécute le contenu du tampon de requête
  - la requête doit renvoyer au moins 3 colonnes
- Affiche le résultat dans une grille croisée
  - colV, en-tête vertical
  - colH, en-tête horizontal
  - colD, contenu à afficher dans le tableau
  - sortColH, colonne de tri pour l'en-tête horizontal

Disons que nous voulons récupérer le nombre de bouteilles en stock par type de vin (blanc, rosé, rouge) par année, pour les années entre 1950 et 1959 :

```
cave=# SELECT t.libelle, s.annee, sum(s.nombre) FROM stock s
JOIN vin v ON v.id=s.vin_id
JOIN type_vin t ON t.id=v.type_vin_id
WHERE s.annee BETWEEN 1950 AND 1957
GROUP BY t.libelle, s.annee
ORDER BY s.annee;

 libelle | annee |  sum
---------+-------+-------
 blanc   |  1950 | 69166
 rose    |  1950 | 70311
```

```
 rouge   |  1950 | 69836
 blanc   |  1951 | 67325
 rose    |  1951 | 67616
 rouge   |  1951 | 66708
 blanc   |  1952 | 67501
 rose    |  1952 | 67481
 rouge   |  1952 | 66116
 blanc   |  1953 | 67890
 rose    |  1953 | 67902
 rouge   |  1953 | 67045
 blanc   |  1954 | 67471
 rose    |  1954 | 67759
 rouge   |  1954 | 67299
 blanc   |  1955 | 67306
 rose    |  1955 | 68015
 rouge   |  1955 | 66854
 blanc   |  1956 | 67281
 rose    |  1956 | 67458
 rouge   |  1956 | 66990
 blanc   |  1957 | 67323
 rose    |  1957 | 67374
 rouge   |  1957 | 66409
(26 rows)
```

Et maintenant nous souhaitons afficher ces informations avec en abscisse le libellé du type de vin et en ordonnée les années :

```
cave=# \crosstabview
 libelle | 1950  | 1951  | 1952  | 1953  | 1954  | 1955  | 1956  | 1957
---------+-------+-------+-------+-------+-------+-------+-------+------
 blanc   | 69166 | 67325 | 67501 | 67890 | 67471 | 67306 | 67281 | 67323
 rose    | 70311 | 67616 | 67481 | 67902 | 67759 | 68015 | 67458 | 67374
 rouge   | 69836 | 66708 | 66116 | 67045 | 67299 | 66854 | 66990 | 66409
(3 rows)
```

Et si nous souhaitons l' inverse (les années en abscisse et les types de vin en ordonnée), c'est tout aussi simple :

```
cave=# \crosstabview annee libelle
 annee | blanc | rose  | rouge
-------+-------+-------+-------
```

```
  1950 | 69166 | 70311 | 69836
  1951 | 67325 | 67616 | 66708
  1952 | 67501 | 67481 | 66116
  1953 | 67890 | 67902 | 67045
  1954 | 67471 | 67759 | 67299
  1955 | 67306 | 68015 | 66854
  1956 | 67281 | 67458 | 66990
  1957 | 67323 | 67374 | 66409
(8 rows)
```

4.4.9 FORMATAGE DANS LES SCRIPTS SQL

- La commande pset
  - `\pset option [ valeur ]`
- Activer le mode étendu
  - `\pset expanded on`
- Donner un titre au résultat de la requête
  - `\pset title 'Résultat de la requête'`
- Formater le résultat en HTML
  - `\pset format html`

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.

`option` décrit l'option à initialiser. Pour obtenir la liste de ces options et leur valeur, depuis la version 9.3 il est possible d'utiliser la commande `pset` seule :

```
cave=# \pset
border                  1
columns                 0
expanded                auto
fieldsep                '|'
fieldsep_zero           off
footer                  on
format                  aligned
linestyle               ascii
null                    ''
numericlocale           off
pager                   1
```

```
pager_min_lines          0
recordsep                '\n'
recordsep_zero           off
tableattr
title
tuples_only              off
unicode_border_linestyle single
unicode_column_linestyle single
unicode_header_linestyle single
```

Pour certaines options, omettre `valeur` active ou désactive l'option. Voici une illustration de ces possibilités sur l'utilisation de la pagination en mode interactif :

```
cave=# \pset pager off
Pager usage is off.
cave=# \pset pager
Pager is used for long output.
cave=# \pset pager
Pager usage is off.
cave=# \pset pager on
Pager is used for long output.
```

L'utilisation de la complétion peut permettre d'obtenir la liste possible des valeurs pour une option :

```
cave=# \pset format
aligned          html             latex-longtable  unaligned
asciidoc         latex            troff-ms         wrapped
```

La liste complète de ces options de formatage et leur description est disponible dans la documentation de la commande "psql"[63]

4.4.10 CRONTAB

- Attention aux variables d'environnement
- Configuration
  - `crontab -e`

Lorsqu'un script est exécuté par `cron`, l'environnement de l'utilisateur n'est pas initialisé, ou plus simplement, les fichiers de personnalisation (par ex. `.bashrc`) de l'environnement

[63] http://docs.postgresql.fr/9.4/app-psql.html

ne sont pas lus. Seule la valeur `$HOME` est initialisée. Il faut donc prévoir ce cas et initialiser les variables d' environnement requises de façon adéquate.

Par exemple, pour charger l'environnement de l'utilisateur :

```
#!/bin/bash

. ${HOME}/.bashrc

...
```

Enfin, 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`.

Chaque entrée dans la crontab doit respecter un format particulier.

Par exemple, pour lancer un script de maintenance à 03h10 chaque dimanche matin :

```
10 3 * * Sun /usr/local/pg/maintenance.sh >/dev/null &2>1
```

4.4.11 EXEMPLE

Sauvegarder une base et classer l'archive

```
#!/bin/bash

t=`mktemp`
pg_dump $1 | gzip > $t
d=`eval date +%d%m%y-%H%M%S`
mv $t /backup/${1}_${d}.dump.gz
exit 0
```

Par convention le script doit renvoyer 0 s'il s'est déroulé correctement

---

4.5 OUTILS GRAPHIQUES

- Outils graphiques d'administration
  - temBoard
  - pgAdminIII et pgAdmin 4
  - phpPgAdmin

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 et phpPgAdmin.

4.5.1 TEMBOARD

- Adresse: https://github.com/dalibo/temboard
- Version: 1.0
- Licence: PostgreSQL
- Notes: Serveur sur Linux, client web

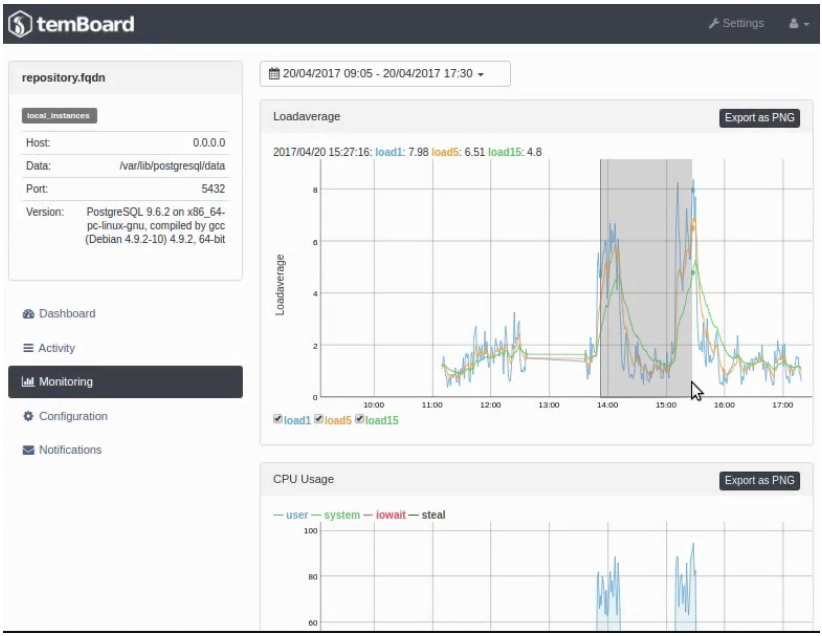
4.5.2 TEMBOARD - POSTGRESQL REMOTE CONTROL

- Multi-instances
- Surveillance OS / PostgreSQL
- Suivi de l'activité
- Configuration de chaque instance

" temBoard est un outil permettant à un DBA de mener à bien la plupart de ses tâches courantes.

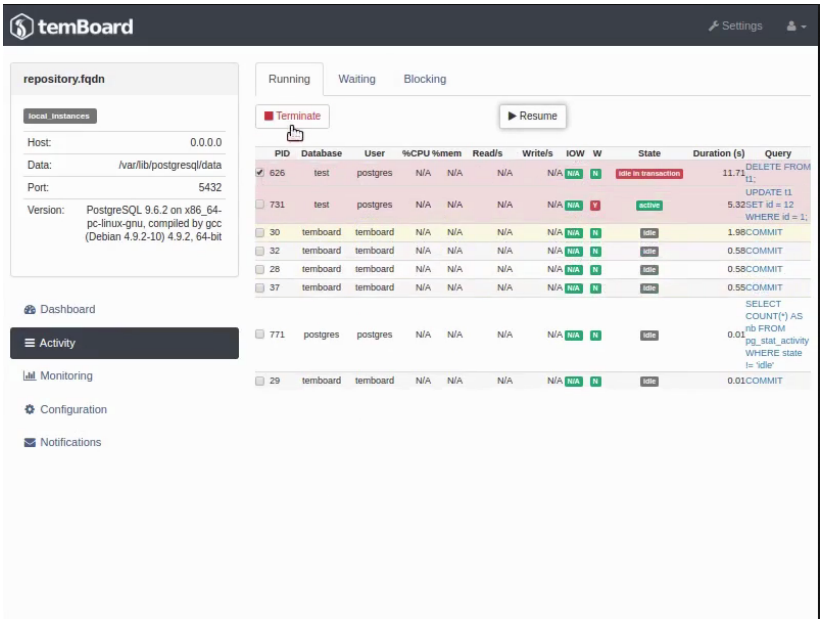
Le serveur web est installé de façon centralisée et un agent est déployé pour chaque instance.

4.5.3 TEMBOARD - MONITORING



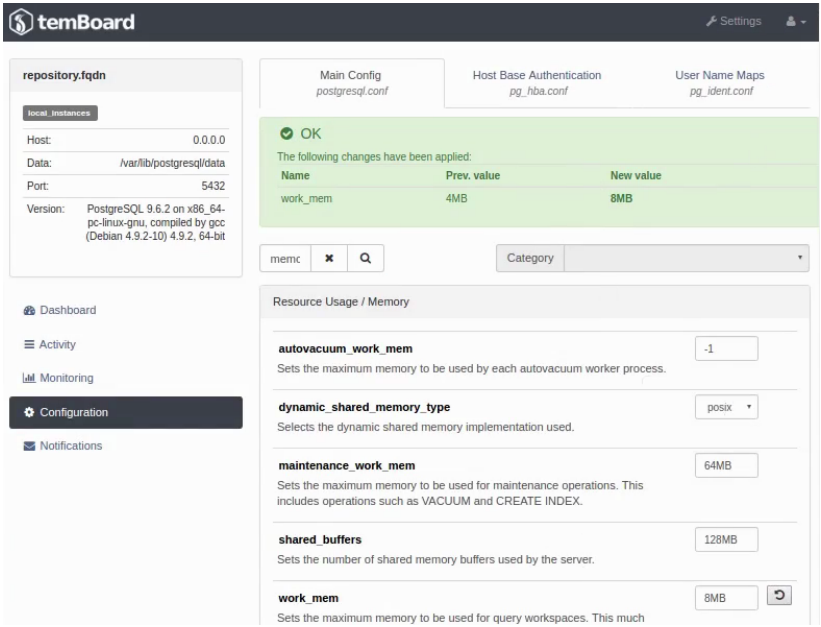
La section **Monitoring** permet de visualiser les graphiques historisés au niveau du système d'exploitation (CPU, mémoire, ...) ainsi qu'au niveau de l'instance PostgreSQL.

4.5.4 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 d'annuler une requête.

4.5.5 TEMBOARD - CONFIGURATION



La section **Configuration** permet de lister le paramètres des fichiers "postgresql.conf", "pg_hba.conf" et "pg_ident.conf".

Elle permet également de modifier ces paramètres. Suivant les cas, il sera proposer de recharger la configuration ou de redémarrer l'instance pour appliquer ces changements.

4.5.6 PGADMINIII

- Adresse: http://www.pgadmin.org
- Version: 1.20.0
- Licence: PostgreSQL
- Notes: Multiplateforme, multilangue

pgAdmin est un client lourd. Il dispose d'un installeur pour Windows et Mac OS X, et de packages pour Linux. Il est disponible sous licence PostgreSQL.

4.5.7 INSTALLATION

- Installeurs Windows et Mac
- Paquets RPM et Debian/Ubuntu

Installer le paquet debian est assez simple :

```
$ aptitude install pgadmin3
```

Néanmoins, il est difficile de trouver une version à jour, que ce soit pour Debian comme pour Red Hat.

L'installeur Windows et celui pour Mac OS X sont des installeurs standards, très simples à utiliser.

Il existe une extension, appelée `Admin Pack`, au sein des modules contrib de PostgreSQL qui ajoute quelques fonctionnalités intéressantes (comme la lecture des traces à distance, et la possibilité de modifier le fichier de configuration à distance) :

Pour l'installer en version 9.1 et ultérieures, il suffit d'exécuter la commande suivante (une fois les modules contrib installés) :

```
$ psql -c "CREATE EXTENSION adminpack" postgres
```

4.5.8 FONCTIONNALITÉS (1/2)

- Connexion possible sur plusieurs serveurs
- Édition des fichiers de configuration locaux
- Permet de gérer toutes les bases d'un même serveur
- Maintenance des bases de données (`vacuum`, `analyze`, `reindex`)
- Visualisation des verrous
- Visualisation des journaux applicatifs
- Débogueur PL/pgsql

4.5.9 FONCTIONNALITÉS (2/2)

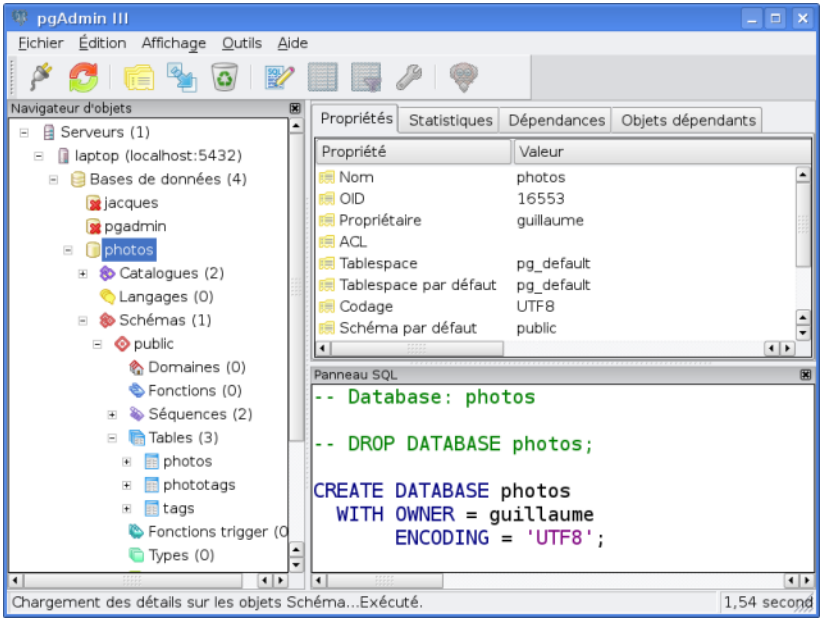
- Sauvegarde et restauration d'une base
- Gestion des rôles
- Création/modification/suppression de tous les objets PostgreSQL
- Possibilité de voir/cacher les objets systèmes
- Éditeur de requête

17.12

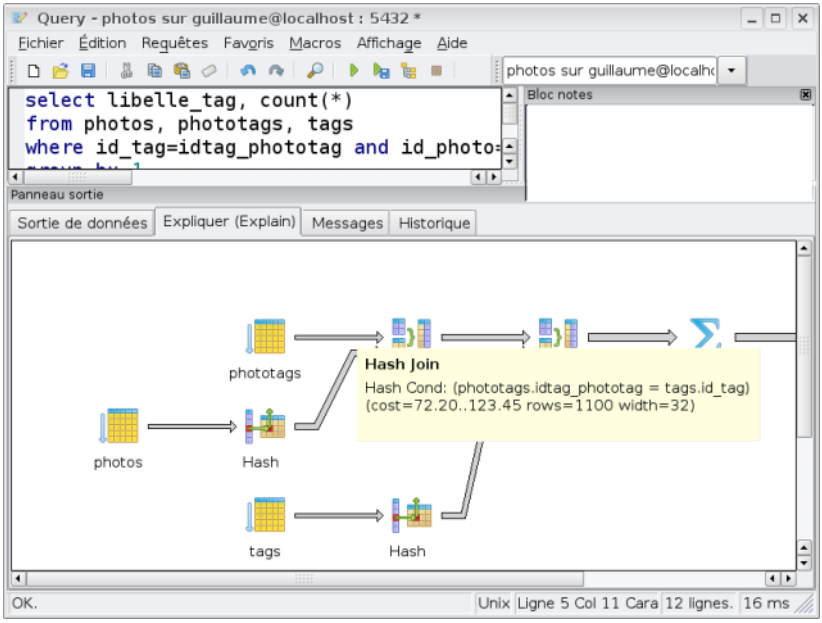
Les objets gérables par `pgAdmin` sont :

- la base
- les tables, les vues et les séquences
- les rôles, les types et les fonctions
- les tablespaces
- les agrégats
- les conversions
- les domaines
- les triggers et les procédures stockées
- les opérateurs, les classes et les familles d'opérateur
- les schémas

4.5.10 LE NAVIGATEUR D'OBJETS



4.5.11 FENÊTRE DE L'ÉDITEUR DE REQUÊTE

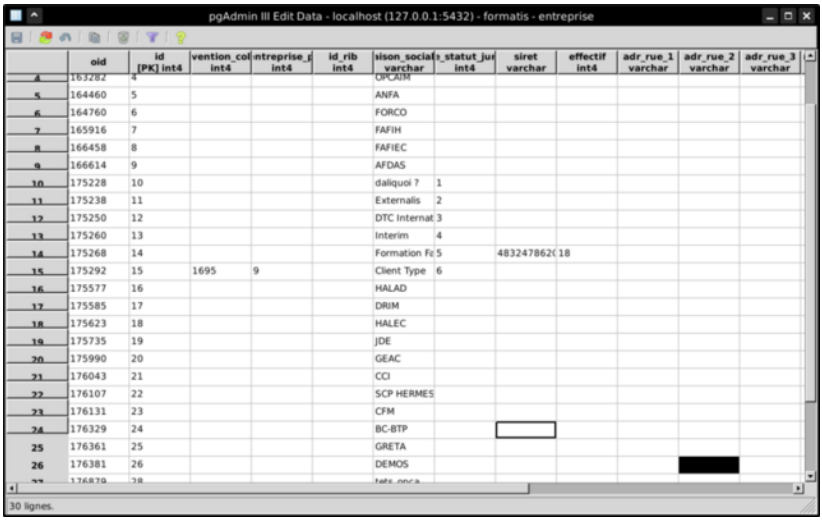


L'éditeur de requête permet de :

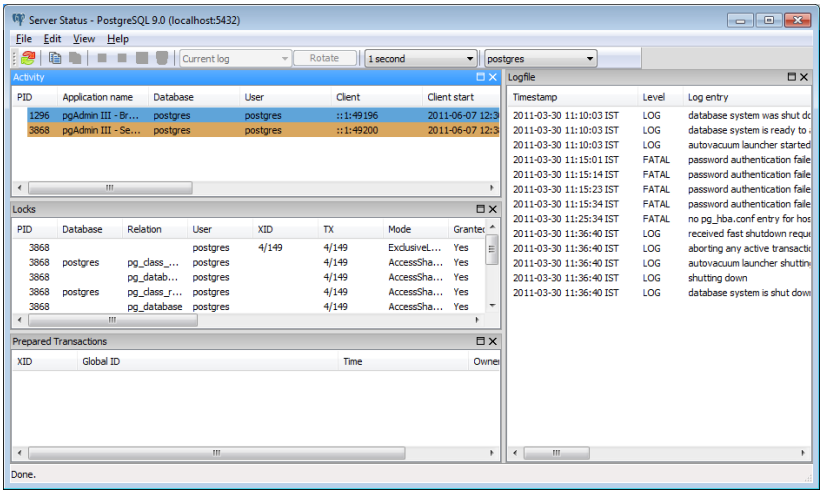
- lire/écrire un fichier de requêtes ;
- exécuter une requête ;
- sauvegarder le résultat d'une requête dans un fichier ;
- exécuter un `EXPLAIN` (avec les options `verbose` et `analyze`) sur une requête.

L'éditeur de requête est accessible depuis le menu *Outil/Éditeur de requêtes* ou depuis un icône de la barre d'outils.

4.5.12 AFFICHAGE DES DONNÉES



4.5.13 FENÊTRE D'ÉTAT DU SERVEUR

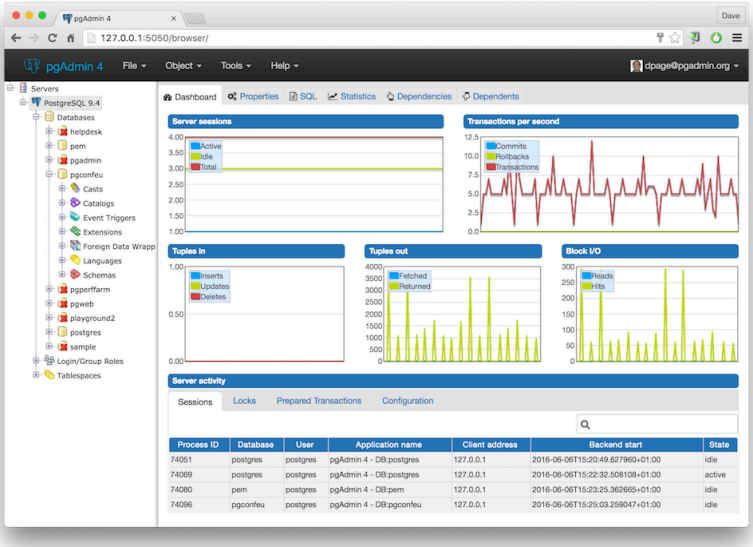


4.5.14 PGADMIN IV

- Adresse: http://www.pgadmin.org
- Version: 1.1.0
- Licence: PostgreSQL
- Notes: Multiplateforme, multilangue

pgAdmin 4 est une application web. Il peut être déployé sur Linux, Windows et Mac OS X. Il est disponible sous licence PostgreSQL.

4.5.15 PGADMIN IV - COPIE D'ÉCRAN



Une des nouvelles fonctionnalités de pgAdmin IV est l'apparition d'un dashboard remontant quelques métriques intéressantes.

4.5.16 PGADMIN III OU PGADMIN IV

- pgAdmin III
  - existe depuis longtemps
  - stabilisé
  - mais en fin de vie
- pgAdmin IV
  - très jeune
  - complexe à installer
  - le seul à être maintenu

pgAdmin III est un projet qui ne verra plus de développement et même de corrections de bugs. L'équipe de développement de pgAdmin n'est pas assez nombreuse pour maintenir pgAdmin III tout en développant pgAdmin IV.

pgAdmin IV est encore un peu jeune et rugueux. Cependant, son développement va bon train et il faudra compter sur lui dans les années à venir.

4.5.17 PHPPGADMIN

- Adresse: http://phppgadmin.sourceforge.net/
- Version: 5.1.0
- Licence: GNU Public License
- Notes: Multiplateforme
- Mais ne semble plus maintenu

PhpPgAdmin est une application web simple d'emploi.

Cependant il n'est plus maintenu (version 5.1 en 2013) et risque de ne plus être compatible avec les futures versions de PostgreSQL.

4.5.18 FONCTIONNALITÉS 1/2

- Application web déportée
- Création, maintenance de bases de données
- Import et export de données
- Exécution de commandes SQL et de scripts (upload)

- Gestion des tablespaces
- Gestion des utilisateurs et des droits
- Gestion des connexions
- Support multi-lingue (31 langues supportés)
- Support des différentes opérations de maintenance

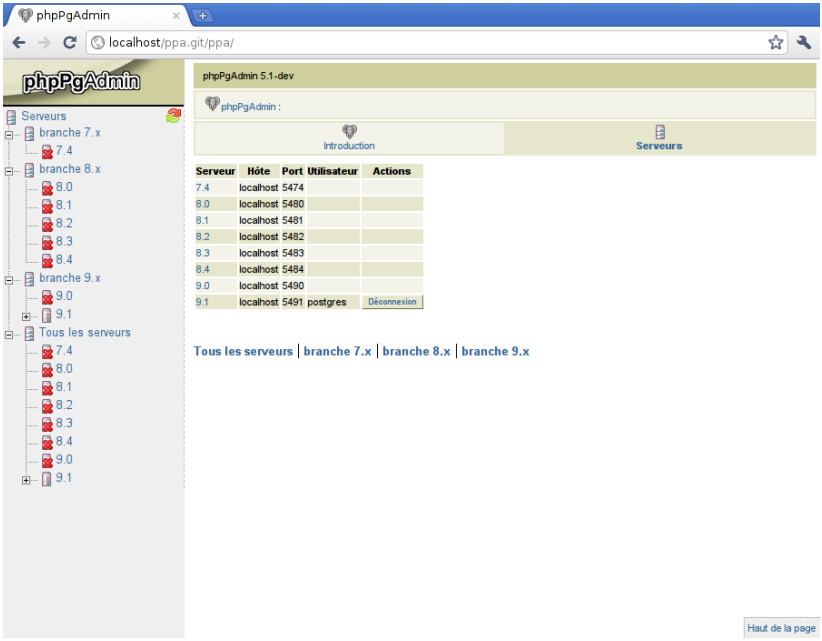
phpPgAdmin a certains avantages:

- capable de gérer un nombre illimité de serveurs PostgreSQL
- permet sur les serveurs PostgreSQL de n'ouvrir l'accès qu'à une seule machine dans le fichier `pg_hba.conf`, plutôt qu'à chacun des postes DBA
- centralise l'installation et la configuration des accès vers les serveurs en un seul point plutôt que de multiplier ceux-ci sur tous les postes DBA

4.5.19 FONCTIONNALITÉS 2/2

- Gestion des tables, vues, index, séquences
- Gestion des contraintes, fonctions, triggers, règles,
- Gestions des types, agrégats, domaines
- Visualisation des verrous
- Visualisation des opérateurs, classes d'opérateurs et conversions entre encodages
- Visualisation des languages et conversions de type
- Configuration de l'autovacuum
- Configuration de la recherche plein texte

4.5.20 PHPPGADMIN : PRÉSENTATION GÉNÉRALE



phpPgAdmin se compose de deux volets:

- une arborescence sur la gauche permettant de naviguer parmi les objets des serveurs et de leur bases de données
- la page principale à droite affichant les détails de l'objet voulu ou de l'action réalisée

4.5.21 ARBORESCENCE - APERÇUS

Détails

Le volet de gauche de phpPgAdmin présente les différents serveurs accessibles depuis l'instance configurée.

Pour chaque serveur, l'arborescence inclut chaque base ainsi que leurs objets.

Groupes de serveurs

phpPgAdmin permet de regrouper les différents serveurs PostgreSQL qui y sont configurés en groupe logique.

Sur la capture d'écran présentée ici, l'administrateur a décidé de regrouper ses serveurs en fonction de la branche à laquelle ils appartiennent, soit trois groupes distincts:

- "branche 7.x"
- "branche 8.x"
- "branche 9.x"

Voici l'extrait du fichier "conf/config.inc.php" qui permet cette configuration:

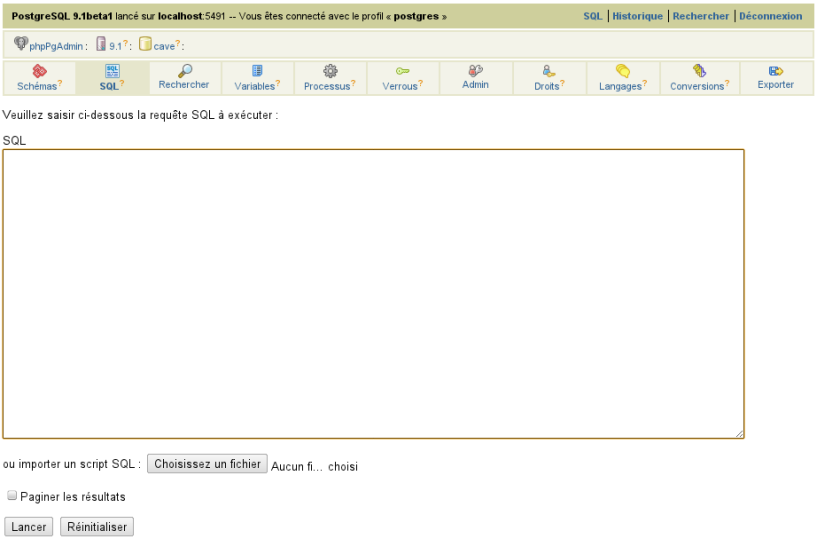
```
// Groups allow administrators to logicaly group servers together under group
// nodes in the left browser tree
$conf['srv_groups'][0]['desc'] = "branche 7.x";
$conf['srv_groups'][0]['servers'] = '0';
$conf['srv_groups'][1]['desc'] = 'branche 8.x';
$conf['srv_groups'][1]['servers'] = '1,2,3,4,5';
$conf['srv_groups'][2]['desc'] = 'branche 9.x';
$conf['srv_groups'][2]['servers'] = '6,7';
```

Ce qui peut-être aussi écrit de cette façon:

```
$conf['srv_groups'] = array(
  array(
    'desc' => 'branche 7.x',
    'servers' => '0'
  ),
  array(
    'desc' => 'branche 8.x',
    'servers' => '1,2,3,4,5'
  ),
  array(
    'desc' => 'branche 9.x',
    'servers' => '6,7'
  )
);
```

Le nœud nommé ”Tous les serveurs” regroupe l’ensemble des serveurs présents dans la configuration de phpPgAdmin.

4.5.22 EXÉCUTER DES REQUÊTES



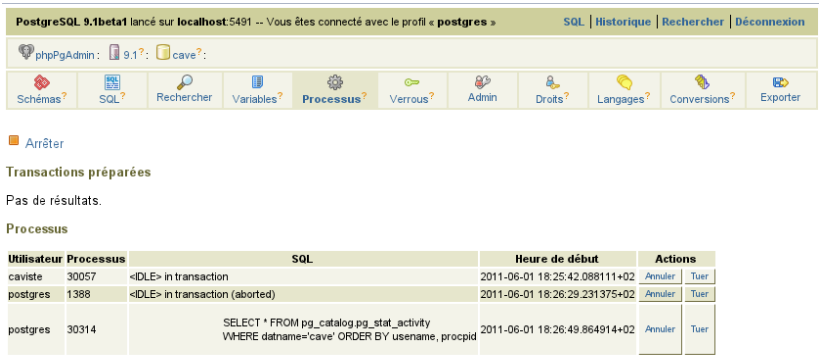
L’exécution de requêtes SQL depuis phpPgAdmin se fait depuis l’onglet ”SQL” accessible au niveau de chaque base de données ou d’un de leurs schémas.

Il existe deux méthodes pour exécuter ces requêtes SQL :

- l’exécution directe en ligne en remplissant le formulaire
- l’exécution d’un script SQL en choisissant un fichier SQL depuis votre disque dur local

---

4.5.23 PROCESSUS EN COURS



L'onglet "processus" accessible au niveau de chacune des bases de données permet de visualiser les processus connectés à la base de données courante. Chaque processus correspond à une connexion à la base de donnée.

Nous y trouvons l'utilisateur utilisé pour la connexion, son activité et à quel moment a été exécutée la requête courante. De plus, si votre utilisateur de connexion en a le droit, cette page vous permet d'interrompre une requête, voire même de forcer la déconnexion d'une session, respectivement à l'aide des actions "Annuler" et "Tuer".

Si votre navigateur supporte les requêtes asynchrones en javascript (aussi nommé *ajax*), les données sur les sessions sont alors rafraîchies automatiquement toutes les trois secondes. Il est possible d'arrêter ce comportement en cliquant sur "Arrêter".

4.6 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.

4.6.1 QUESTIONS

N'hésitez pas, c'est le moment !

4.7 TRAVAUX PRATIQUES

4.7.1 ÉNONCÉS

psql

Le but de ce TP est d'acquérir certains automatismes dans l'utilisation de `psql`.

L'ensemble des informations permettant de résoudre ces exercices a été abordé au cours de la partie théorique. Il est également possible de trouver les réponses dans le manuel de psql (`man psql`) ou dans l'aide en ligne de l'outil.

1. Lancez la console en se connectant à la base *cave* avec l'utilisateur `postgres`.
2. Déconnectez-vous.
3. Connectez-vous à la machine de votre voisin. Affichez l'ensemble des bases accessibles.
4. Déconnectez-vous.
5. Reconnectez-vous à votre machine.
6. Affichez l'ensemble des tables systèmes.
7. Sans vous déconnecter, prenez l'identité de l'utilisateur *caviste*.
8. Affichez l'ensemble des objets (tables, index, séquences...) de l'utilisateur *caviste*.
9. Affichez la liste des tables.
10. Effectuez une copie de la liste des appellations dans le fichier `/tmp/liste_app.txt`.
11. Affichez l'aide de la commande permettant de créer une table à partir des informations contenue dans une autre.
12. Créez une table `vin_text` composée de l'id du vin, le nom du récoltant, l'appellation et le type de vin.
13. Affichez le répertoire courant. Changez le pour `/tmp`. Vérifiez que vous êtes bien dans ce répertoire.
14. Sauvegardez le contenu de la table `vin_text` dans un fichier `CSV` (`liste_vin.txt`).
15. Écrivez un fichier contenant deux requêtes. Exécutez ce fichier.

16. Exécutez une requête (peu importe laquelle) à partir de psql. Affichez la dernière requête exécutée en utilisant une méta-commande. Exécutez de nouveau cette requête à l'aide d'une autre méta-commande.

4.7.2 SOLUTIONS

psql

1. `psql cave postgres`
2. `\q`
3.

```
$ psql cave postgres -h une_ip
cave=> \l
```

4. `<CRTL>+D`
5. `psql cave postgres`
6. `\dS`
7. `\c - caviste`
8. `\d`
9. `\dt`
10. `\copy appellation to '/tmp/liste_app.txt'`
11. `\h CREATE TABLE AS`
12.

```
CREATE TABLE vin_text AS
SELECT
vin.id, recoltant.nom AS recoltant, appellation.libelle AS appellation,
type_vin.libelle AS type
FROM vin
JOIN appellation ON (appellation.id=vin.appellation_id)
JOIN type_vin ON (type_vin.id=vin.type_vin_id)
JOIN recoltant ON (recoltant.id=vin.recoltant_id);
```

13.

```
\! pwd
\cd /tmp
\! pwd
```

14. `\copy vin_text to '/tmp/liste_vin.txt' (format CSV, delimiter ';')`

15.

```
postgres@wanted:/tmp$ vi requete.sql
postgres@wanted:/tmp$ psql -f requete.sql cave caviste
```

16.

```
\p
\g
```

5 TÂCHES COURANTES

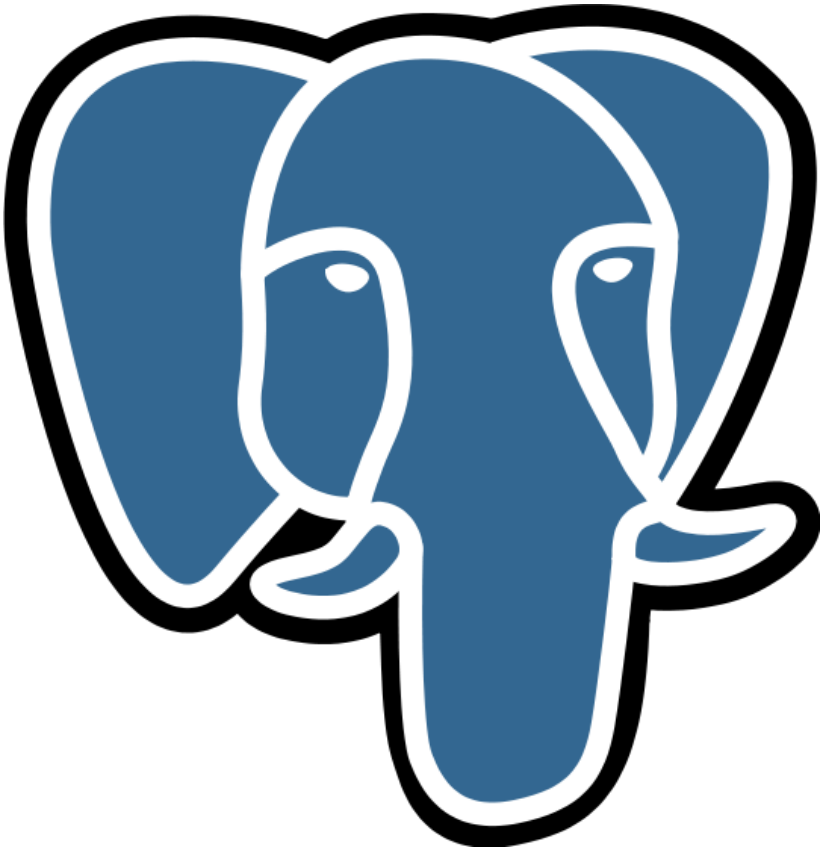


FIGURE 1: POSTGRESQL

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 `\l` dans `psql`

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;
-[ RECORD 1 ]-+----------------------------------
datname       | postgres
datdba        | 10
encoding      | 6
datcollate    | C
datctype      | C
datistemplate | f
datallowconn  | t
datconnlimit  | -1
datlastsysoid | 13355
datfrozenxid  | 1797
datminmxid    | 1
dattablespace | 1663
datacl        |
```

```
-[ RECORD 2 ]-+-----------------------------------
datname       | template1
datdba        | 10
encoding      | 6
datcollate    | C
datctype      | C
datistemplate | t
datallowconn  | t
datconnlimit  | -1
datlastsysoid | 13355
datfrozenxid  | 1797
datminmxid    | 1
dattablespace | 1663
datacl        | {=c/postgres,postgres=CTc/postgres}
-[ RECORD 3 ]-+-----------------------------------
datname       | template0
datdba        | 10
encoding      | 6
datcollate    | C
datctype      | C
datistemplate | t
datallowconn  | f
datconnlimit  | -1
datlastsysoid | 13355
datfrozenxid  | 1797
datminmxid    | 1
dattablespace | 1663
datacl        | {=c/postgres,postgres=CTc/postgres}
```

Voici la signification des différentes colonnes :

- `datname`, le nom de la base ;
- `datdba`, l'identifiant de l'utilisateur propriétaire de cette base (pour avoir des informations sur cet utilisateur, il suffit de chercher l'utilisateur dont l 'OID correspond à cet identifiant dans le catalogue système `pg_roles`) ;
- `encoding`, l'identifiant de l'encodage de cette base ;
- `datcollate`, la locale gérant le tri des données de type texte pour cette base (à noter que cette colonne apparait à partir de la version 8.4) ;
- `datctype`, la locale gérant le jeu de caractères pour les données de type texte pour cette base (à noter que cette colonne apparait à partir de la version 8.4) ;

- datistemplate, pour préciser si cette base est une base de données utilisable comme modèle ;
- datallowconn, pour préciser s'il est autorisé de se connecter à cette base ;
- datconnlimit, limite du nombre de connexions pour les utilisateurs standards, en simultanée sur cette base (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) ;
- datlastsysoid, information système indiquant le dernier OID utilisé sur cette base ;
- datfrozenxid, plus ancien identifiant de transaction géré par cette base ;
- dattablespace, l'identifiant du tablespace par défaut de cette base (pour avoir des informations sur ce tablespace, il suffit de chercher le tablespace dont l'OID correspond à cet identifiant dans le catalogue système pg_tablespaces) ;
- datacl, droits pour cette base (un champ vide indique qu'il n'y a pas de droits spécifiques pour cette base).

Pour avoir une vue plus simple, il est préférable d'utiliser la méta-commande \l dans psql :

```
postgres=# \l
                                 List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 cave      | caviste  | UTF8     | C       | C     |
 module_C1 | postgres | UTF8     | C       | C     |
 postgres  | postgres | UTF8     | C       | C     |
 prod      | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(6 rows)
```

Avec le suffixe +, il est possible d'avoir plus d'informations (comme la taille, le commentaire, etc). Néanmoins, la méta-commande \l ne fait qu'accéder aux tables systèmes. Par exemple :

```
$ psql -E postgres
psql (10beta3)
Type "help" for help.
```

```
postgres=# \x
Expanded display is on.
postgres=# \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",
    d.datcollate as "Collate",
    d.datctype as "Ctype",
    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;

**************************

List of databases
-[ RECORD 1 ]-----+-------------------------------------------
Name              | cave
Owner             | caviste
Encoding          | UTF8
Collate           | C
Ctype             | C
Access privileges |
Size              | 40 MB
Tablespace        | pg_default
Description       |
-[ RECORD 2 ]-----+-------------------------------------------
Name              | module_C1
Owner             | postgres
Encoding          | UTF8
Collate           | C
Ctype             | C
```

```
Access privileges |
Size              | 7223 kB
Tablespace        | pg_default
Description       |
-[ RECORD 3 ]-----+-------------------------------------------
Name              | postgres
Owner             | postgres
Encoding          | UTF8
Collate           | C
Ctype             | C
Access privileges |
Size              | 7175 kB
Tablespace        | pg_default
Description       | default administrative connection database
-[ RECORD 4 ]-----+-------------------------------------------
(...)
-[ RECORD 5 ]-----+-------------------------------------------
Name              | template0
Owner             | postgres
Encoding          | UTF8
Collate           | fr_FR.UTF-8
Ctype             | fr_FR.UTF-8
Access privileges | =c/postgres                              +
                  | postgres=CTc/postgres
Size              | 6953 kB
Tablespace        | pg_default
Description       | unmodifiable empty database
-[ RECORD 6 ]-----+-------------------------------------------
Name              | template1
Owner             | postgres
Encoding          | UTF8
Collate           | fr_FR.UTF-8
Ctype             | fr_FR.UTF-8
Access privileges | =c/postgres                              +
                  | postgres=CTc/postgres
Size              | 6953 kB
Tablespace        | pg_default
Description       | default template for new databases
```

La requête affichée montre bien que `psql` accède au catalogue `pg_database`, ainsi qu'à

des fonctions systèmes 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
- Par défaut, `template1` est utilisée
- 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`, disponible à partir de la version 8.4, 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) ;
- `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. Avant la 9.0, il était coutume d'ajouter le langage PL/pgsql dans la base de données template1 pour que toutes les bases créées après 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 sur 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
```

```
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;
UPDATE 1
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
                                  List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 b1        | postgres | UTF8     | C       | C     |
 b2        | u1       | UTF8     | C       | C     |
 b3        | u2       | UTF8     | C       | C     |
 b4        | u2       | UTF8     | C       | C     |
 b5        | u2       | UTF8     | C       | C     |
 cave      | caviste  | UTF8     | C       | C     |
 module_C1 | postgres | UTF8     | C       | C     |
 postgres  | postgres | UTF8     | C       | C     |
 prod      | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(11 rows)
```

Avant la version 9.5, la clause `IS_TEMPLATE` n'existait pas. Il fallait modifier manuellement le catalogue système ainsi :

```
postgres=# UPDATE pg_database SET datistemplate=true WHERE datname='b2';
```

Il est possible de créer une base sans avoir à se rappeler de la commande SQL. Le plus simple est certainement l'outil `createdb`, livré avec PostgreSQL, mais c'est aussi possible avec n'importe quel autre outil d'administration de bases de données PostgreSQL.

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 `-e` de cette commande permet de voir exactement ce que `createdb` exécute :

```
$ createdb -e --owner u1 b6
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: 248.477 ms  statement: CREATE DATABASE b6 OWNER u1;
createdb - LOG:  disconnection: session time: 0:00:00.252 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, base existante
- **Outil système** : `dropdb`

Supprimer une base de données supprime tous les objets et toutes les données contenus 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 même les déconnecter autoritairement :

```
postgres=# SELECT pg_terminate_backend(pid)
postgres-# FROM pg_stat_activity
postgres-# WHERE datname='b6';
 pg_terminate_backend
----------------------
 t
(1 row)

postgres=# DROP DATABASE b6;
DROP DATABASE
```

Attention, l'utilisation de pg_terminate_backend() n'est disponible que pour les utilisateurs appartenant au même rôle que l'utilisateur à déconnecter ainsi que pour les utilisateurs membre du rôle `pg_signal_backend`.

```
postgres=> SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE datname='b5';
ERROR:  must be a member of the role whose process is being terminated or member
        of pg_signal_backend
```

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 :

```
[guillaume@laptop formation]$ dropdb -e b5
DROP DATABASE b5;
```

Contrairement à `createdb`, sans nom de base, `dropdb` ne fait rien.

5.2.5 MODIFICATION / CONFIGURATION

- `ALTER DATABASE`
  - pour modifier quelques méta-données ;
  - pour ajouter, modifier ou supprimer une configuration.

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
----------
 16MB
(1 row)

b1=# ALTER DATABASE b1 SET work_mem TO '2MB';
ALTER DATABASE
b1=# SHOW work_mem;
 work_mem
----------
 16MB
(1 row)

b1=# \c b1
You are now connected to database "b1" as user "postgres".
b1=# SHOW work_mem;
 work_mem
----------
 2MB
(1 row)
```

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 * FROM pg_db_role_setting ;
 setdatabase | setrole |           setconfig
-------------+---------+-------------------------------
       57553 |       0 | {work_mem=32MB}
           0 |   57552 | {maintenance_work_mem=256MB}
       67396 |       0 | {work_mem=2MB}
(1 row)

b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
b1-# setconfig AS "Configuration"
b1-# FROM pg_db_role_setting
b1-# LEFT JOIN pg_database d ON d.oid=setdatabase
b1-# LEFT JOIN pg_roles r ON r.oid=setrole
b1-# ORDER BY 1, 2;
 Base | Utilisateur |        Configuration
------+-------------+-------------------------------
 b1   |             | {work_mem=2MB}
 cave |             | {work_mem=32MB}
      | caviste     | {maintenance_work_mem=256MB}
(3 rows)

b1=# ALTER DATABASE b2 SET work_mem to '10MB';
ALTER DATABASE
b1=# ALTER DATABASE b2 SET maintenance_work_mem to '128MB';
ALTER DATABASE
b1=# ALTER DATABASE b2 SET random_page_cost to 3;
ALTER DATABASE
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
b1-# setconfig AS "Configuration"
b1-# FROM pg_db_role_setting
b1-# LEFT JOIN pg_database d ON d.oid=setdatabase
b1-# LEFT JOIN pg_roles r ON r.oid=setrole
b1-# ORDER BY 1, 2;
 Base | Utilisateur |                          Configuration
------+-------------+---------------------------------------------------------------
 b1   |             | {work_mem=2MB}
```

```
 b2   |             | {work_mem=10MB,maintenance_work_mem=128MB,                +
      |             |  random_page_cost=3}
 cave |             | {work_mem=32MB}
      | caviste     | {maintenance_work_mem=256MB}
(4 rows)
```

Pour annuler la configuration d'un paramètre, utilisez :

```
ALTER DATABASE base RESET paramètre;
```

Par exemple :

```
b1=# ALTER DATABASE b2 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=10MB,maintenance_work_mem=128MB}
 cave |             | {work_mem=32MB}
      | caviste     | {maintenance_work_mem=256MB}
(4 rows)
```

Si vous copiez une base de données dont certains paramètres ont été configurés spécifiquement pour cette base de données, ces paramètres ne sont pas appliqués à la nouvelle base de données. De même, une sauvegarde effectuée avec l'outil `pg_dump` ne contient pas cette configuration spécifique.

5.3 RÔLES

- Liste des rôles
- Création
- Suppression
- Modification

- Utilisateur/groupe
- Gestion des mots de passe

Avec les versions de PostgreSQL antérieures à la 8.1, les utilisateurs et groupes sont gérés comme dans Unix. On peut créer les deux, les groupes étant une manière logique de grouper les utilisateurs pour faciliter la gestion des droits. Ces derniers peuvent être accordés ou révoqués à un groupe entier.

À partir de la version 8.1 apparaît la notion de rôle en lieu et place des utilisateurs et des groupes. 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.

Les concepts des « utilisateurs » et des « groupes » restent disponibles dans les versions 8.1 et ultérieures. Il n'y a donc pas de problème de compatibilité ascendante.

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

- Rôles à partir de la 8.1
- Utilisateurs et de groupes avant... mais aussi après
- Ordres SQL
  - `CREATE/DROP/ALTER USER`
  - `CREATE/DROP/ALTER GROUP`

Les rôles n'apparaissent qu'à partir de 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`

- Commande `\du` dans `psql`

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_signal_backend
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      |
oid            | 4200
-[ RECORD 3 ]--+------------------------------
rolname        | caviste
rolsuper       | f
rolinherit     | t
```

```
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      | {maintenance_work_mem=256MB}
oid            | 57552
```

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 n'est-il pas contraint aux 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 méta-commande `\du` dans `psql` :

```
postgres=# \du
List of roles
-[ RECORD 1 ]----------------------------------------------------------
Role name  | admin
Attributes |
Member of  | {}
-[ RECORD 2 ]----------------------------------------------------------
Role name  | caviste
```

```
Attributes |
Member of  | {}
-[ RECORD 3 ]-----------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 4 ]-----------------------------------------------------------
Role name  | stagiaire2
Attributes |
Member of  | {}
-[ RECORD 5 ]-----------------------------------------------------------
Role name  | u1
Attributes |
Member of  | {pg_signal_backend}
-[ RECORD 6 ]-----------------------------------------------------------
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 méta-commande `\du` ne fait qu'accéder aux tables systèmes. Par exemple :

```
[guillaume@laptop formation]$ psql -E postgres
psql (9.1.3)
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;
**************************

-[ RECORD 1 ]----------------------------------------------------------
Role name  | admin
Attributes |
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
- `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  | admin
Attributes |
Member of  | {}
-[ RECORD 2 ]-------------------------------------------------------
Role name  | caviste
Attributes |
Member of  | {}
-[ RECORD 3 ]-------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 4 ]-------------------------------------------------------
Role name  | stagiaire2
Attributes |
Member of  | {}
-[ RECORD 5 ]-------------------------------------------------------
Role name  | u1
```

```
Attributes |
Member of  | {pg_signal_backend}
-[ RECORD 6 ]-------------------------------------------------------
Role name  | u2
Attributes | Create DB
Member of  | {}
-[ RECORD 7 ]-------------------------------------------------------
Role name  | u3
Attributes | Cannot login
Member of  | {}
-[ RECORD 8 ]-------------------------------------------------------
Role name  | u4
Attributes | Create role, Cannot login
Member of  | {}
-[ RECORD 9 ]-------------------------------------------------------
Role name  | u5
Attributes |
Member of  | {u2,u6}
-[ RECORD 10 ]------------------------------------------------------
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 | {}
 u1        |                                              | {}
 u2        | Create DB                                    | {}
 u3        | Cannot login                                 | {}
 u4        | Create role, Cannot login                    | {}
 u6        | Cannot login                                 | {}
 u7        | Create role                                  | {}
 u8        |                                              | {}
 u9        | Create DB                                    | {}
```

Pour compatibilité avec les applications développées pour les versions de PostgreSQL antérieures à la 8.1, 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 `-e` de cette commande nous permet de voir exactement ce que `createuser` exécute :

```
$ createuser -e u10 --superuser
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 créé 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 superutilisateurs et les utilisateurs disposant de l'attribut 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 -e u10
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.

Avec la commande `ALTER ROLE`, il est possible de modifier quelques méta-données :

- le nom du rôle ;
- son mot de passe ;
- sa limite de validité ;
- ses attributs
  - SUPERUSER
  - CREATEDB
  - CREATEROLE
  - CREATEUSER
  - LOGIN
  - REPLICATION
  - BYPASSRLS
  - CONNECTION LIMIT

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 (10beta3)
Type "help" for help.

postgres=> SHOW work_mem;
 work_mem
----------
 16MB
```

```
(1 row)

postgres=> ALTER ROLE u2 SET work_mem TO '20MB';
ALTER ROLE
postgres=> SHOW work_mem;
 work_mem
----------
 16MB
(1 row)

postgres=> \q
$ psql -U u2 postgres
psql (10beta3)
Type "help" for help.

postgres=> SHOW work_mem;
 work_mem
----------
 20MB
(1 row)
```

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",
b1=# setconfig AS "Configuration"
b1-# FROM pg_db_role_setting
b1-# LEFT JOIN pg_database d ON d.oid=setdatabase
b1-# LEFT JOIN pg_roles r ON r.oid=setrole
b1-# ORDER BY 1, 2;
 Base | Utilisateur |               Configuration
------+-------------+--------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=10MB,maintenance_work_mem=128MB}
 cave |             | {work_mem=32MB}
      | caviste     | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
(5 rows)
```

À partir de la version 9.0, 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
(1 row)

postgres=> \c b1 u2
You are now connected to database "b1" as user "u2".
b1=> SHOW work_mem;
 work_mem
----------
 10MB
(1 row)

b1=> \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> SHOW work_mem;
 work_mem
----------
 2MB
(1 row)

b1=> \c postgres u1
You are now connected to database "postgres" as user "u1".
postgres=> SHOW work_mem;
 work_mem
----------
 16MB
(1 row)

b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
b1=# setconfig AS "Configuration"
b1-# FROM pg_db_role_setting
b1-# LEFT JOIN pg_database d ON d.oid=setdatabase
```

```
b1-# LEFT JOIN pg_roles r ON r.oid=setrole
b1-# ORDER BY 1, 2;
 Base | Utilisateur |                Configuration
------+-------------+-------------------------------------------
 b1   | u2          | {work_mem=10MB}
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=10MB,maintenance_work_mem=128MB}
 cave |             | {work_mem=32MB}
      | caviste     | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
(6 rows)
```

Pour annuler la configuration d'un paramètre pour un rôle, utilisez :

```
ALTER ROLE rôle RESET paramètre;
```

Une sauvegarde effectuée avec l'outil `pg_dump` ne contient pas cette configuration spécifique. Par contre, cette configuration est sauvegardée avec l'outil `pg_dumpall` (avec ou sans l'option `-r`).

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

- Toujours mettre un mot de passe
- Attention aux traces
- Mot de passe chiffré à privilégier
- Mais
  - Pas de vérification de la faiblesse du mot de passe
  - Pas de date limite de validité du rôle (le mot de passe peut avoir une date limite de validité)

Par défaut, les utilisateurs n'ont pas de mot passe. Si la méthode d'authentification demande la saisie d'un mot de passe, les utilisateurs sans mot de passe ne pourront pas se connecter. Comme il est très fortement conseillé d'utiliser une méthode

d'authentification avec saisie du mot de passe, il faut ajouter un mot de passe aux utilisateurs.

Cela se fait simplement :

```
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.

Si les traces enregistrent les requêtes exécutées, le mot de passe paraîtra en clair dans les traces :

```
$ grep PASSWORD $PGDATA/log/traces.log
psql - LOG: duration: 1.865 ms  statement: ALTER ROLE u1 PASSWORD 'supersecret';
```

Il est donc essentiel de s'arranger pour que seules des personnes de confiance aient accès aux traces. S'il n'est pas possible de faire cela, le mieux est d'envoyer le mot de passe chiffré. Pour cela, il faut coder le mot de passe au format MD5 après lui avoir concaténé le nom du rôle. Cela nous donne ceci :

```
$ echo -n "supersecretu1" | md5sum
fb75f17111cea61e62b54ab950dd1268  -
$ psql postgres
psql (10beta3)
Type "help" for help.

postgres=# ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
ALTER ROLE
postgres=# \q
$ grep PASSWORD $PGDATA/log/traces.log
psql - LOG:  duration: 2.100 ms  statement: ALTER ROLE u1
                                 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Le mot de passe à saisir est toujours supersecret, mais seule sa forme chiffrée apparaît dans les traces.

Une autre solution est de demander temporairement la désactivation des traces pendant le changement de mot de passe. Il faut cependant être superutilisateur pour pouvoir le faire :

```
$ psql postgres
psql (10beta3)
Type "help" for help.
```

```
postgres=# SET log_min_duration_statement TO 0;
SET
postgres=# SET log_statement TO none;
SET
postgres=# ALTER ROLE u1 PASSWORD 'supersecret';
ALTER ROLE
postgres=# \q
$ grep PASSWORD $PGDATA/log/postgresql-2012-01-10_093849.log
[rien]
```

Il est à noter que PostgreSQL ne vérifie pas la faiblesse d'un mot de passe. Il ne permet pas non plus d'avoir une date limite de validité sur les mots de passe. Pour le premier, il est possible d'installer une extension appelée `check_password`. Pour l'activer, le paramètre `shared_preload_libraries` doit être modifié ainsi :

```
shared_preload_libraries = 'passwordcheck'
```

Après avoir redémarré le serveur, toute modification de mot de passe sera vérifié :

```
postgres=# ALTER ROLE u1 PASSWORD 'supersecret';
ERROR:  password must contain both letters and nonletters
```

Il est très fortement conseillé de modifier le code source de cette extension pour y ajouter les règles convenant à votre cas. L'utilisation de la bibliothèque `Cracklib` permet d'aller encore plus loin.

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`
- Retirer un droit : `REVOKE`
- Droits spécifiques pour chaque type d'objets
- Avoir le droit de donner le droit : `WITH GRANT OPTION`

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. 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
b1=> INSERT INTO s1.t1 VALUES (1), (2);
INSERT 0 2
b1=> SELECT * FROM s1.t1;
 c1
----
  1
  2
(2 rows)

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
(2 rows)
```

```
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... long et fastidieux... En 9.0, 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. Tous les droits sont donnés sur ce schéma à un pseudo-rôle appelé public. Tous les rôles sont automatiquement membres de ce pseudo-rôle. Si vous voulez gérer complètement les droits sur ce schéma, il faut d'abord penser à enlever les droits pour ce pseudo-rôle. Pour cela, il vous faut utiliser la commande `REVOKE` ainsi :

```
REVOKE ALL ON SCHEMA public FROM public;
```

Enfin, toujours avec la 9.0, 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.

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
(1 row)

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
(1 row)

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
(1 row)
```

5.4.2 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
  - mais qui peut être un groupe

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
------------+----------------------------------------------+--------------------
 admin      |                                              | {}
 caviste    |                                              | {}
 postgres   | Superuser, Create role, Create DB,          +| {}
            | Replication, Bypass RLS                      |
 stagiaire2 |                                              | {}
 u1         |                                              | {pg_signal_backend}
 u11        |                                              | {}
 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 | clients   | table | r31
 public | factures  | table | r31
 public | t1        | table | r31
 public | t2        | table | r32
(4 rows)

b1=# REASSIGN OWNED BY r31 TO u1;
REASSIGN OWNED
b1=# \d
         List of relations
 Schema |   Name    | Type  | Owner
--------+-----------+-------+-------
 public | clients   | table | u1
 public | factures  | table | u1
 public | t1        | table | u1
 public | t2        | table | r32
(4 rows)

b1=# DROP OWNED BY u1;
DROP OWNED
b1=# \d
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
```

```
 public | t2   | table | r32
(1 row)
```

5.4.3 DROITS PLUS GLOBAUX

- Rôles systèmes d'administration
  - pg_signal_backend (9.6+)
- Rôles systèmes de supervision
  - pg_read_all_stats (10+)
  - pg_read_all_settings (10+)
  - pg_stat_scan_tables (10+)
  - pg_monitor (10+)

Certaines fonctionnalités nécessitent l'attribut `SUPERUSER` alors qu'il serait bon de pouvoir les effectuer sans avoir ce droit suprême. Cela concerne principalement 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
(2 rows)

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
```

17.12

```
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT pg_terminate_backend(23194);
 pg_terminate_backend
----------------------
 t
(1 row)

postgres=> SELECT usename, pid FROM pg_stat_activity WHERE usename IS NOT NULL;
 usename |  pid
---------+-------
 u1      | 23212
(1 row)
```

Par contre, les connexions des superutilisateurs ne sont pas concernées.

En version 10, quatre nouveaux rôles sont ajoutées. `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`.

5.4.4 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
```

5.4.5 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
(2 rows)

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
(2 rows)

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)
- Suivant les trois 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. HBA est l'acronyme de Host Based Authentication.

5.5.1 INFORMATIONS DE CONNEXION

- Quatre informations :
  - socket Unix ou adresse/alias IP
  - numéro de port
  - nom de la base
  - nom du rôle
- Fournies explicitement ou implicitement

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
- 5432 pour le numéro de port
- suivant l'outil, la base postgres ou le nom de l'utilisateur PostgreSQL, pour le nom de la base
- 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

- PostgreSQL utilise les informations de connexion pour sélectionner la méthode
- Fichier de configuration: `pg_hba.conf`

- Se présente sous la forme d'un tableau
  - 4 colonnes d'informations
  - 1 colonne indiquant la méthode à appliquer
  - 1 colonne optionnelle d'options

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. Il va donc parcourir les lignes du tableau enregistré dans le fichier `pg_hba.conf`. Il les parcourt dans l'ordre. La première qui correspond aux informations fournies lui précise la méthode d'authentification. Il ne lui 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.

Il est donc essentiel de bien configurer ce fichier pour avoir une protection maximale.

Le tableau se présente ainsi :

```
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnossl  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

5.5.3 COLONNE TYPE

- 4 valeurs possibles
  - `local`
  - `host`
  - `hostssl`
  - `hostnossl`
- `hostssl` nécessite d'avoir activé `ssl` dans `postgresql.conf`

La colonne type peut contenir quatre valeurs différentes. La valeur `local` concerne les connexions via la socket Unix. Toutes les autres valeurs concernent les connexions via la socket TCP. La différence réside dans l'utilisation forcée ou non du SSL :

- `host`, connexion via la socket TCP, avec ou sans SSL ;
- `hostssl`, connexion via la socket TCP, avec SSL ;
- `hostnossl`, connexion via la socket TCP, sans SSL.

17.12

Il est à noter que l'option `hostssl` n'est utilisable que si le paramètre `ssl` du fichier `postgresql.conf` est à `on`.

5.5.4 COLONNE DATABASE

- Nom de la base
- Plusieurs bases (séparées par des virgules)
- Nom d'un fichier contenant la liste des bases (précédé par une arobase)
- Mais aussi
  - `all` (pour toutes les bases)
  - `sameuser`, `samerole` (pour la base de même nom que le rôle)
  - `replication` (pour les connexions de réplication)

La colonne 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. La valeur replication est utilisée pour les connexions de réplication (il n'est pas nécessaire d'avoir une base nommée replication). Enfin, la valeur `sameuser` spécifie que l'enregistrement n'intercepte que si la base de données demandée a le même nom que le rôle demandé, 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.

5.5.5 COLONNE USER

- Nom du rôle
- Nom d'un groupe (précédé par un signe plus)
- Plusieurs rôles (séparés par des virgules)
- Nom d'un fichier contenant la liste des rôles (précédé par une arobase)
- Mais aussi
  - `all` (pour tous les rôles)

La colonne peut recueillir le nom d'un rôle, le nom d'un groupe en le précédant d'un signe plus, le nom de plusieurs rôles 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 tous les rôles.

5.5.6 COLONNE ADRESSE IP

- Uniquement dans le cas d'une connexion `host`, `hostssl` et `hostnossl`
- Soit l'adresse IP et le masque réseau
- Soit l'adresse au format CIDR
- Pas possible d'utiliser un nom DNS avant la 9.1

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 deux exemples d'adresses IP au format CIDR :

```
192.168.168.1/32
192.168.168.0/24
```

Il est possible d'utiliser un nom d'hôte ou un domaine DNS à partir de la version 9.1, au prix d'une recherche DNS pour chaque hostname présent, pour chaque nouvelle connexion.

5.5.7 COLONNE MÉTHODE

- Précise la méthode d'authentification à utiliser
- Deux types de méthodes
  - internes
  - externes
- Possibilité d'ajouter des options dans une dernière colonne

La colonne de la méthode est la dernière colonne, voire l'avant-dernière si vous voulez ajouter une option à la méthode d'authentification.

5.5.8 COLONNE OPTIONS

- Dépend de la méthode d'authentification
- Méthode externe : option `map`

Les options disponibles dépendent de la méthode d'authentification sélectionnée. Toutes les méthodes externes permettent l'utilisation de l'option map. Cette option a pour but d'indiquer la carte de correspondance à sélectionner dans le fichier `pg_ident.conf`.

5.5.9 MÉTHODES INTERNES

- `trust`
- `reject`
- `password`
- `md5`

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.

La méthode `md5` est certainement la méthode la plus intéressante. La saisie du mot de passe est forcée. De plus, le mot de passe transite chiffré en md5.

La méthode `reject` est intéressante dans certains cas de figure. 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  md5
local  all  u1  reject
```

5.5.10 MÉTHODES EXTERNES

- `ldap`, `radius`, `cert`
- `gss`, `sspi`
- `ident`, `peer`, `pam`
- `bsd`

Ces différentes méthodes permettent d'utiliser des annuaires d'entreprise comme RADIUS, LDAP ou un ActiveDirectory. Certaines méthodes sont spécifiques à Unix (comme `ident` et `peer`), voire à Linux (comme `pam`).

La méthode `LDAP` utilise un serveur LDAP pour authentifier l'utilisateur. Ceci n'est disponible qu'à partir de la version 8.2.

La méthode `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 ». Attention, ce n'est disponible qu'à partir de la version 8.3.

La méthode `Radius` permet d'utiliser un serveur RADIUS pour authentifier l' utilisateur, dès la version 9.0.

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.

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.

Quant à `pam`, il authentifie l'utilisateur en passant par les `Pluggable Authentication Modules` (PAM) fournis par le système d'exploitation.

Avec la version 9.6 apparaît la méthode `bsd`. Cette méthode est similaire à la méthode `password` mais utilise le système d'authentification BSD.

5.5.11 MÉTHODES OBSOLÈTES

- `crypt` depuis la version 8.4
- `krb5` depuis la version 9.3

La méthode `crypt` est jugée trop faible et ne fait plus partie des méthodes utilisables à partir de la version 8.4.

La méthode `krb5` a été définitivement supprimée du code en version 9.4. Elle n'était déjà plus utilisable depuis la version 9.3 et signalée obsolète depuis la version 8.3. Il convient d'employer la méthode `GSSAPI` pour utiliser une authentification à base de Kerberos.

5.5.12 UN EXEMPLE DE PG_HBA.CONF

Un exemple:

17.12

```
TYPE    DATABASE  USER        CIDR-ADDRESS    METHOD
local   all       postgres                    ident
local   web       web                         md5
local   sameuser  all                         ident
host    all       postgres    127.0.0.1/32    ident
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   md5
```

à ne pas suivre…

Ce fichier comporte plusieurs erreurs:

```
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                         ident
```

Le masque CIDR de

```
host    all       all         89.192.0.3/8    md5
```

est incorrect, ce qui fait qu'au lieu d' autoriser 89.192.0.3 à se connecter, on autorise tout le réseau 89.*

L'entrée

```
hostssl recherche recherche   89.192.0.4/32   md5
```

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.

5.6 TÂCHES

- Maintenance
- Sauvegarde
- Supervision

Après installation d'une instance, l'administrateur de bases de données a trois tâches principales. Il doit d'abord s'assurer qu'une maintenance de l' instance est faite régulièrement. Il ne s'agit pas de s'assurer de la fiabilité du système, mais de ses performances. Ensuite, il doit mettre en place une sauvegarde des bases : soit individuellement, soit de l'instance

complète. Quelle que soit la solution de sauvegarde choisie il doit pouvoir réinstaller le serveur si besoin est. Enfin, la mise en place d'une solution de supervision est un point essentiel pour garder un œil sur le bon état du système.

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.

5.6.1 INTRODUCTION À LA MAINTENANCE

- Trois opérations essentielles
  - `VACUUM`
  - `ANALYZE`
  - `REINDEX`
- Automatisable par cron
- ... et par autovacuum (pour les deux premiers)

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 des tables ;
- le `REINDEX`, pour favoriser l'utilisation des 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.

Pour éviter d'avoir à exécuter manuellement ces opérations, il est possible de les automatiser, soit avec cron (sous Unix), soit avec autovacuum. Cependant, l'autovacuum ne s'occupe pas du `REINDEX`. Il sera donc nécessaire de passer par cron pour une opération `REINDEX`.

5.6.2 MAINTENANCE : VACUUM

- Lutter contre la fragmentation
- VACUUM
  - cartographie les espaces libres pour une prochaine réutilisation
  - utilisable en parallèle avec les autres opérations
  - vue pg_stat_progress_vacuum en 9.6
- VACUUM FULL

- défragmente la table
- verrou exclusif

PostgreSQL ne supprime pas les versions périmées des lignes après un `UPDATE` ou un `DELETE`. 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 » marque les données expirées dans les tables et les index pour une utilisation future. Il ne tente pas de récupérer 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 puisse 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.

La version 8.4 améliore les performances du `VACUUM` en lui permettant de ne parcourir que la partie de la table qui a été modifiée. Cela a un gros impact pour les tables les moins modifiées. La version 9.6 apporte une vue appelée `pg_stat_progress_vacuum` qui permet de suivre l'exécution d'un `VACUUM`.

Cependant, un `VACUUM` fait rarement gagner de l'espace disque. Il faut utiliser l'option `FULL` pour ça. La commande `VACUUM FULL` libère l'espace consommé par les lignes périmées et le rend au système d'exploitation.

Cette variante de la commande `VACUUM` acquiert un verrou exclusif sur chaque table. Elle peut donc avoir un effet extrêmement négatif sur les performances de la base de données.

Notez cependant que la version 9.0 améliore grandement les performances de cette commande.

Quand faut-il utiliser `VACUUM` ?

- pour des nettoyages réguliers
- il s'agit d'une maintenance de base

Quand faut-il utiliser `VACUUM FULL` ?

- après des suppressions massives de données
- lorsque la base n'est pas en production
- il s'agit 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.

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, de façon à ce que la taille de la table soit

réduite de façon conséquente. Avant la 9.0, un REINDEX est essentiel après un VACUUM FULL. Ce n'est plus le cas à partir de la 9.0.

La commande vacuumdb permet d'exécuter facilement un VACUUM sur une ou toutes les bases, elle permet également la parallélisation de VACUUM sur plusieurs tables (à partir de la version 9.5).

5.6.3 MAINTENANCE : ANALYZE

- Met à jour les statistiques sur les données
- Utilisées par l'optimiseur de requêtes
- Échantillonnage global : default_statistics_target
- Échantillonnage par colonne possible

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.

En général, une bonne stratégie est de programmer ANALYZE au moins une fois par jour. Ceci peut être couplé à un VACUUM (la nuit par exemple) pour gagner en performances.

Le paramètre default_statistics_target initialise l'échantillonnage par défaut des statistiques pour les colonnes de chacune des tables. Des valeurs plus importantes accroissent le temps nécessaire à exécuter ANALYZE et augmentent la place prise sur disque, mais pourraient améliorer les estimations du planificateur. La valeur par défaut est de 10 jusqu'en version 8.3 et 100 à partir de la version 8.4.

Ainsi, pour chaque colonne avant la version 8.4, les 10 valeurs les plus fréquentes et 10 histogrammes sont stockés dans pg_stats en guise d' échantillon représentatif des données (3000 lignes sont considérées). 10 fournissant des statistiques plutôt pauvres et parfois éloignées de la réalité, avec l'optimisation du processus ANALYZE, la valeur par défaut a donc été relevée à 100 en 8.4 (pour un échantillon de 30000 lignes).

Si vous voulez des statistiques plus fines, vous pouvez passer ce paramètre à 300. Des valeurs supérieures sont possibles, sans dépasser la limite de 1000, mais provoquent un ralentissement important d ' ANALYZE, un accroissement de la table pg_stats, et un temps de calcul des plans d'exécution plus long (non mesurable jusqu'à 100).

Voici la commande à utiliser si l'on veut modifier cette valeur colonne par colonne, la valeur ainsi spécifiée prévaut sur la valeur de default_statistics_target:

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 200;
```

Généralement, il faut passer par l'outil `psql`. Cependant, depuis la version 9.0, il est aussi possible de passer par la commande `vacuumdb`.

5.6.4 MAINTENANCE : REINDEX

- Lancer `REINDEX` régulièrement permet
  - de gagner de l'espace disque
  - d'améliorer les performances
  - de réparer un index corrompu
- `VACUUM` ne provoque pas de réindexation
- `VACUUM FULL` réindexe

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index.

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 serait toujours 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 parce que les pages adjacentes logiquement sont habituellement aussi physiquement adjacentes dans un index nouvellement créé. Cette considération ne s'applique qu'aux index B-tree. Il pourrait être intéressant de ré-indexer périodiquement, simplement pour améliorer la vitesse d'accès.

La réindexation est aussi utile dans le cas d'un index corrompu ne contenant plus de données valides. Deux raisons peuvent amener un index à être invalide. Avant la versio 10, les index de type hash n'étaient pas journalisés. Donc un système reconstruit à partir des journaux de transactions aura tous ses index hash invalides. Il faudra les reconstruire pour qu'ils soient utilisables. L'autre raison vient de la clause `CONCURRENTLY` de l'ordre `CREATE INDEX`. Cette clause permet de créer un index sans bloquer les écritures dans la table. Si, au bout de deux passes, l'index n'est toujours pas complet, il est considéré comme invalide et doit être soit détruit puis construit, soit reconstruit avec la commande `REINDEX`.

Il est à savoir que l'opération `VACUUM` sans le mode `FULL` ne provoque pas de réindexation. Une réindexation est effectuée dans le cas de l'utilisation du mode `FULL`, depuis la version 9.0.

La commande système `reindexdb` peut être utilisée pour réindexer une table, une base ou un cluster de bases de données.

5.6.5 MAINTENANCE : CLUSTER

- `CLUSTER`, alternative à `VACUUM FULL`
- Plus rapide que `VACUUM FULL` suivi de `REINDEX` avant la 9.0
- Attention, `CLUSTER` nécessite près du double de l'espace disque utilisé pour stocker la table et ses index

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. Par contre, la grosse différence se situe au niveau des performances. Du fait de son implémentation, `CLUSTER` est beaucoup plus rapide qu'un `VACUUM FULL` pour les versions antérieures à la 9.0. À partir de la 9.0, l'implémentation du `VACUUM FULL` a été revue pour se calquer sur celle de `CLUSTER` (sans la partie tri des données) et ainsi gagner en performance.

Au final, `CLUSTER` est un équivalent plus rapide d'un `VACUUM FULL` suivi d'un `REINDEX`, à l'organisation physique des blocs de données près.

L'outil standard s'appelle `clusterdb` et permet de lancer la réorganisation de tables ayant déjà fait l'objet d'une "clusterisation".

5.6.6 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`.

5.6.7 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 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 `VACUUM`. Il serait plus intéressant d'avoir 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.

Cet outil apparaît dès la version 7.4 et est intégré au moteur à partir de la 8.1. Il est fortement amélioré au fil des versions, en terme de performance, de fonctionnalités, et de paramétrage. Il est maintenant activé par défaut. Il est conseillé de le laisser ainsi. Son paramétrage permet d' aller plus loin si nécessaire.

5.6.8 INTRODUCTION À LA SAUVEGARDE

- Deux outils historiques
  - `pg_dump`
  - `pg_dumpall`
- Sauvegarde automatisable par cron

La sauvegarde d'un serveur PostgreSQL peut s'effectuer de plusieurs façons. Nous y reviendrons en détail dans le module sur la sauvegarde. Ici, nous allons simplement donner quelques pistes pour mettre en place une sauvegarde à chaud, cohérente et complète. PostgreSQL propose deux outils pour cela : pg_dump, qui permet de sauvegarder une base de données de façon complète ou partielle, et pg_dumpall, qui sauvegarde une instance complète (autrement dit toutes les bases de données, mais aussi la définition des bases, des utilisateurs et des tablespaces).

5.6.9 SAUVEGARDE AVEC PG_DUMP

- Sauvegarde une base
  - À chaud
  - Cohérente
  - Complète ou partielle
- Exemple:

```
pg_dump -f b1.dump b1
```

Le principe de `pg_dump` est de générer un fichier texte (par défaut) ou binaire de commandes SQL, qui, s'il est renvoyé au serveur, recrée une base de données identique à celle sauvegardée.

L'usage basique est :

```
pg_dump base_de_donnees > fichier_de_sortie
```

Le fichier dump peut être restauré avec la commande :

```
psql base_de_donnees < fichier_d_entree
```

Cette sauvegarde se fait sans bloquer les utilisateurs. Ils peuvent toujours lire des données, insérer/modifier/supprimer des données, créer des objets. Par contre, ils ne peuvent pas modifier la structure d'un objet qui est en train d'être sauvegardé (donc pas de `ALTER TABLE` par exemple). Ils ne peuvent pas non plus supprimer les objets qui sont en train d'être sauvegardés (pas de `DROP TABLE`).

La restauration doit se faire sur une base vide ou, tout du moins, ne contenant pas les objets qui vont être restaurés.

5.6.10 SAUVEGARDE AVEC PG_DUMPALL

- Sauvegarde l'instance complète
  - définition des utilisateurs et tablespaces
  - contenu de chaque base
- Comme pg_dump, c'est :
  - À chaud
  - Cohérent

17.12

`pg_dumpall` simplifie la tâche de l'administrateur en sauvegardant toutes les bases de données d'une instance et préserve les données communes à l' instance (les rôles et les tablespaces par exemple) :

```
pg_dumpall > fichier_de_sortie
```

Le fichier de sauvegarde résultant peut être restauré avec `psql` :

```
psql -f fichier_d_entree postgres
```

5.6.11 INTRODUCTION À LA SUPERVISION

- PostgreSQL fournit deux canaux d'information
  - les statistiques d'activité
  - les traces
- Mais ne fait pas d'historisation sur les statistiques
- Donc, il est nécessaire de mettre en place un outil externe

Dernière responsabilité de l'administrateur de bases de données : la mise en place d'une solution de supervision.

PostgreSQL fournit deux canaux d'informations pour l'administrateur :

- les statistiques d'activité (par exemple, le nombre de lignes lues par parcours séquentiel et par parcours d'index, ou encore le nombre de blocs lus dans le cache et celui lus en dehors du cache) ;
- les traces (contenant les messages d'erreurs, des informations sur les performances des requêtes, des checkpoint ou de l'autovacuum).

Pour les statistiques, PostgreSQL ne fait aucune historisation. Il est donc important de mettre en place un outil capable de récupérer ces informations.

Quant aux traces, là-aussi, disposer d'un outil capable d'analyser les traces automatiquement est une bonne chose.

5.6.12 SUPERVISION : STATISTIQUES

- Activer le collecteur de statistiques
- Vues statistiques
- Outils disponibles

Le récupérateur de statistiques de PostgreSQL rapporte des informations sur l' activité du serveur. Il peut compter l'accès aux tables et index en terme de blocs disques et de lignes individuelles. Il permet également de déterminer la commande en cours d'exécution par les processus serveur. Ces informations sont ensuite disponibles dans des vues systèmes. Elles commencent toutes par le préfixe `pg_stat`. Plusieurs outils permettent de récupérer les informations des tables pour générer des alertes (comme `check_pgactivity`[64]) et pour générer des graphes (comme munin).

5.6.13 SUPERVISION : TRACES

- Bien configurer les traces
- Outils :
  - tail_n_mail
  - pgBadger

PostgreSQL dispose de nombreuses options pour les traces. Il est essentiel de bien configurer le système de journalisation applicative pour avoir suffisamment d'informations : ni trop (auquel cas il serait difficile de trouver l'information importante) ni trop peu (auquel cas l'information ne serait même pas disponible). Comme pour les statistiques, il est possible d'utiliser des outils comme `tail_n_mail` qui permet de recevoir des mails lorsque certains messages apparaissent et comme `pgBadger` qui génère des rapports HTML sur les performances des requêtes.

5.7 SÉCURITÉ

- Ce qu'un utilisateur standard peut faire
  - et ne peut pas faire
- Restreindre les droits
- Chiffrement
- Corruption silencieuse

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.

[64] https://github.com/OPMDG/check_pgactivity/

5.7.1 PAR DÉFAUT

- Un utilisateur standard peut
  - accéder à toutes les bases de données
  - créer des objets dans le schéma PUBLIC de toute base de données
  - créer des objets temporaires
  - modifier les paramètres de la session
  - créer des fonctions
  - exécuter des fonctions définies par d'autres dans le schéma PUBLIC
  - récupérer des informations sur l'instance
  - visualiser le source des vues et des fonctions

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;
```

Il peut créer des objets dans le schéma disponible par défaut (nommé `public`) sur chacune des bases de données où il peut se connecter. Il est possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE CREATE ON SCHEMA public FROM nom_utilisateur;
```

Il 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;
```

Il peut modifier les paramètres de session (par exemple le paramètre `work_mem` pour s'allouer beaucoup plus de mémoire). Il est impossible d'empêcher cela.

Il peut créer des fonctions, uniquement avec les langages de confiance, uniquement dans les schémas où il a le droit de créer des objets. 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;
```

Il peut exécuter toute fonction, y compris définie par d'autres, à condition qu'elles soient créées dans des schémas où il a accès. Il est possible d'empêcher cela en supprimant le droit d'exécution d'une fonction:

```
REVOKE EXECUTE ON FUNCTION nom_fonction FROM nom_utilisateur;
```

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.

Enfin, il peut visualiser le source 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.

5.7.2 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. 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.

5.7.3 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.

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 abordé dans le chapitre Droits de connexion de ce module de formation.

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. Il est possible d'aller plus loin avec l'instruction `SECURITY LABEL` disponible depuis la version 9.1. 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 appelé `se-pgsql`.

Certains objets disposent de droits particuliers. Par exemple, les fonctions disposent des clauses `SECURITY DEFINER` et `LEAKPROOF`. La première permet d' indiquer 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. La seconde 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.

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.

```
demo_9_2=# CREATE TABLE elements (id serial, contenu text, prive boolean);
NOTICE:  CREATE TABLE will create implicit sequence "elements_id_seq" for
         serial column "elements.id"
CREATE TABLE
demo_9_2=# INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true), ('d', false),
       ('e prive aussi', true);
INSERT 0 5
```

```
demo_9_2=# 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
(5 rows)
```

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.

```
demo_9_2=# CREATE OR REPLACE VIEW elements_public AS SELECT * FROM elements
WHERE CASE WHEN current_user='guillaume' THEN TRUE ELSE NOT prive END;
CREATE VIEW
demo_9_2=# SELECT * FROM elements_public;
 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
(5 rows)

demo_9_2=# CREATE USER u1;
CREATE ROLE
demo_9_2=# GRANT SELECT ON elements_public TO u1;
GRANT
demo_9_2=# \c - u1
You are now connected to database "postgres" as user "u1".
demo_9_2=> SELECT * FROM elements;
ERROR:  permission denied for relation elements
demo_9_2=> SELECT * FROM elements_public ;
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

```
(3 rows)
```

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, cela peut changer :

```
demo_9_2=> CREATE OR REPLACE FUNCTION abracadabra(integer, text, boolean)
RETURNS bool AS $$
BEGIN
RAISE NOTICE '% - % - %', $ 1, $ 2, $ 3;
RETURN true;
END$$
LANGUAGE plpgsql
COST 0.00000000000000000001;
CREATE FUNCTION
demo_9_2=> SELECT * FROM elements_public WHERE 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
(3 rows)
```

Que s'est-il passé ? Pour comprendre, il suffit de regarder l'EXPLAIN de cette requête :

```
demo_9_2=> EXPLAIN SELECT * FROM elements_public
WHERE abracadabra(id, contenu, prive);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.15 rows=202 width=37)
   Filter: (abracadabra(id, contenu, prive) AND
           CASE WHEN ("current_user"() = 'u1'::name)
                THEN (NOT prive) ELSE true END)
(2 rows)
```

La fonction `abracadrabra` a un coût si faible que PostgreSQL l'exécute avant le filtre de

la vue. Du coup, la fonction voit toutes les lignes de la table.

Seul moyen d'échapper à cette optimisation du planificateur, utiliser l'option `security_barrier` en 9.2 :

```
demo_9_2=> \c - guillaume
You are now connected to database "postgres" as user "guillaume".
demo_9_2=# 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;
CREATE VIEW
demo_9_2=# \c - u1
You are now connected to database "postgres" as user "u1".
demo_9_2=> SELECT * FROM elements_public WHERE 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
(3 rows)

demo_9_2=> EXPLAIN SELECT * FROM elements_public
WHERE abracadabra(id, contenu, prive);
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..34.20 rows=202 width=37)
   Filter: abracadabra(elements_public.id, elements_public.contenu,
                       elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.15 rows=605 width=37)
         Filter: CASE WHEN ("current_user"() = 'u1'::name)
                      THEN (NOT prive) ELSE true END
(4 rows)
```

Voir aussi cet article, par Robert Haas[65] .

Avec PostgreSQL 9.3, l'utilisation de l'option `security_barrier` empêche également l'utilisateur de modifier les données de la table sous-jacente, même s'il a les droits en

[65] http://rhaas.blogspot.com/2012/03/security-barrier-views.html

écriture sur la vue :

```
postgres=# GRANT UPDATE ON elements_public TO u1;
GRANT

postgres=# \c - u1
You are now connected to database "postgres" as user "u1".

postgres=> update elements_public set prive = true where id = 2 ;
ERROR:  cannot update view "elements_public"
DETAIL:  Security-barrier views are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an
       unconditional ON UPDATE DO INSTEAD rule.
```

À partir de la version 9.4 de PostgreSQL, c'est désormais possible :

```
postgres=# GRANT UPDATE ON elements_public TO u1;
GRANT

postgres=# \c - u1
You are now connected to database "postgres" as user "u1".

postgres=# update elements_public set contenu = 'e' where id = 1 ;
UPDATE 1

postgres=# select * from elements_public ;
 id | contenu | prive
----+---------+-------
  2 | b       | f
  4 | d       | f
  1 | e       | f
(3 rows)
```

À noter que cela lui permet également par défaut de modifier les lignes même si la nouvelle valeur les lui rend inaccessibles :

```
postgres=> update elements_public set prive = true where id = 1 ;
UPDATE 1

postgres=> select * from elements_public ;
 id | contenu | prive
----+---------+-------
```

```
  2 | b       | f
  4 | d       | f
(2 rows)
```

La version 9.4 de PostgreSQL apporte donc également la possibilité d'empêcher ce comportement, grâce à l'option `WITH CHECK OPTION` de la syntaxe de création de la vue :

```
postgres=# 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 ;

postgres=# \c - u1
You are now connected to database "postgres" as user "u1".

postgres=> update elements_public set prive = true where id = 2 ;
ERREUR:  new row violates WITH CHECK OPTION for view "elements_public"
DETAIL:  La ligne en échec contient (2, b, t)
```

5.7.4 CHIFFREMENTS

- Connexions
  - SSL
  - avec ou sans certificats serveur et/ou client
- Données disques
  - pas en natif
  - 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.

5.7.5 CORRUPTION SILENCIEUSE

- Pas de sommes de contrôle dans les fichiers de données
  - sauf à partir de la 9.3
- S'active avec initdb
- Plus de sécurité
  - mais une diminution des performances

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Il est donc possible de modifier un fichier sans qu'il s'en rende compte, ce qui aboutit à une corruption silencieuse. La raison en est que PostgreSQL ne dispose pas de sommes de contrôle sur les fichiers de données par défaut. À partir de la version 9.3, il est possible de créer une instance avec somme de contrôle. En voici un exemple :

Tout d'abord, créons un cluster sans utiliser les checksums, et un autre qui les utilisera. Attention, on ne peut pas modifier un cluster existant pour changer ce paramètre.

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

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

```
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/12036/16317
(1 row)
```

Ensuite, on va s'attacher à corrompre ce fichier, en remplaçant la valeur `toto` avec un éditeur hexadécimal.

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

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

Sans checksums:

```
postgres=# TABLE test;
 name
------
 qoto
```

Avec checksums:

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

Côté performances, on peut aussi réaliser un benchmark rapide. Celui-ci a été réalisé en utilisant `pgbench`, avec une échelle de 1000 (base de 16Go), en utilisant les tests par défaut. La configuration de PostgreSQL utilisée est celle par défaut. En utilisant la moyenne de trois exécutions, on obtient en moyenne 226 transactions par seconde lorsque l'on utilise les checksums contre 281 lorsque l'on ne les utilise pas.

5.8 CONCLUSION

PostgreSQL demande peu de travail au quotidien.

À l'installation, certaines tâches doivent être automatisées, par exemple la sauvegarde, les `VACUUM`.

Pour le reste, il s'agit surtout de surveiller la bonne exécution des scripts automatisés et le contenu des journaux applicatifs.

5.8.1 POUR ALLER PLUS LOIN

- Documentation officielle, chapitre Planifier les tâches de maintenance[66]
- Opérations de maintenance sous PostgreSQL[67]
- Total security in a PostgreSQL database[68]

[66] http://docs.postgresql.fr/9.6/maintenance.html
[67] http://dalibo.org/glmf109_operations_de_maintenance_sous_postgresql
[68] http://www.ibm.com/developerworks/opensource/library/os-postgresecurity/index.html

17.12

- Managing rights in PostgreSQL, from the basics to SE PostgreSQL[69]

5.8.2 QUESTIONS

N'hésitez pas, c'est le moment !

5.9 TRAVAUX PRATIQUES

5.9.1 ÉNONCÉS

Afin de visualiser avec quel rôle vous êtes connecté et sur quelle base, il est possible de modifier le prompt de l'outil `psql`. Pour cela, créer le fichier `$HOME/.psqlrc` avec le contenu suivant: `\set PROMPT1 '%n@%/%R%# '` Pour plus d'information à ce propos, consulter la page de manuel de `psql`, section *Advanced Features* > *Prompting*.

Sur Windows, ce fichier est `%APPDATA%/postgresql/psqlrc.conf`.

MCD Base `cave`

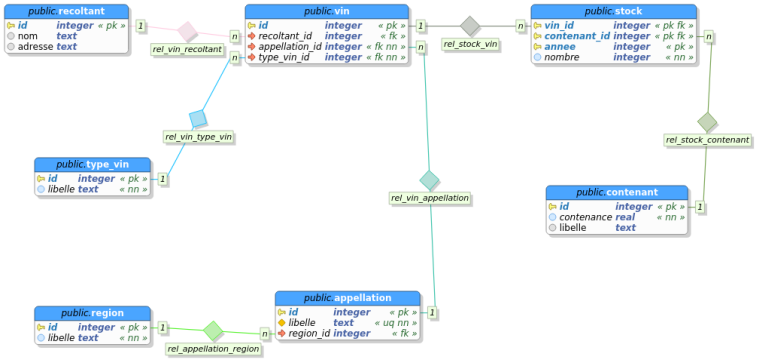


FIGURE 2: SCHÉMA DE LA BASE CAVE

Gestion des rôles

Ajouter la table `facture (id int, objet text, creationts timestamp)` à la base de données `cave`.

[69] http://www.dalibo.org/_media/managing_rights_in_postgresql.pdf

Création d'un utilisateur et d'un groupe

À l'aide du super-utilisateur `postgres` :

- créer un rôle `secretariat` sans droit de connexion, mais qui peut visualiser, ajouter, mettre à jour et supprimer des éléments de la table `facture` ;
- créer un rôle `bob` qui peut se connecter et appartient au rôle `secretariat` ;
- vérifier la création des deux rôles.

À l'aide du rôle `bob` :

- insérer les factures ayant pour objet « Vin de Bordeaux » et « Vin de Bourgogne » avec la date et l'heure courante ;
- sélectionner les factures de la table `factures` ;
- mettre à jour la deuxième facture avec la date et l'heure courante ;
- supprimer la première facture.

Modification des permissions

Retirer les droits `DELETE` sur la table `facture` pour le rôle `secretariat`. Vérifier qu'il n'est pas possible de supprimer la deuxième facture avec le rôle `bob`.

Retirer tous les droits pour le rôle `secretariat` sur la table `facture` et vérifier que le rôle `bob` ne peut pas sélectionner les appellations contenues dans la table `appellation`.

Autoriser `bob` à accéder à la table `appellation` en lecture. Vérifier que `bob` peut désormais accéder à la table `appellation`.

Héritage des droits au login

Créer un rôle `tina` appartenant au rôle `secretariat`, avec l'attribut `LOGIN`, mais n'héritant pas des droits à la connexion.

Vérifier que `tina` ne peut pas accéder à la table `facture`.

Dans la session, activer le rôle `secretariat` (`SET ROLE`) et sélectionner les données de la table `facture`. Vérifier que `tina` possède maintenant les droits du rôle `secretariat`.

pg_hba.conf

Établissez la configuration suivante pour accéder à la base de données:

- autorisez votre voisin de gauche à se connecter sans mot de passe avec l'utilisateur `tina` ;
- autorisez votre voisin de droite à se connecter avec un mot de passe (autorisation en IP sans ssl) avec l'utilisateur `bob`.

VACUUM

17.12

Pré-requis

Désactiver le démon autovacuum de l'instance.

Nettoyage avec VACUUM

Afficher la taille de la table `stock`.

Copier le contenu de la table dans une nouvelle table (`stock_bis`).

Supprimer le contenu de la table stock avec un ordre `DELETE`. Quel est l'espace disque utilisé par cette table ?

Insérer le contenu de la table `stock_bis` dans la table `stock`. Quel est l'espace disque utilisé par la table ?

Répéter plusieurs fois les deux dernières commandes (en terminant par insert). Quel est l'espace disque utilisé par la table ?

Effectuer un VACUUM simple. Vérifier la taille de la base.

Vider la table `stock`, insérer à nouveau le contenu de la table `stock_bis`. L'espace mis à disposition a-t-il été utilisé ?

Nettoyage avec VACUUM FULL

Exécuter à nouveau une série de suppression et d'insertion dans la table **stock**.

Effectuer un VACUUM FULL. Quel est l'impact sur la taille de la base ?

Truncate ou Delete ?

De la même façon que `stock_bis`, créer la table `vin_bis`.

Observer la différence entre les commandes `DELETE` et `TRUNCATE` en tronquant la table `vin_bis`.

Statistiques

Vérifier les paramètres de récupération de statistiques au sein du fichier `postgresql.conf`.

Quelle est la table système stockant les statistiques collectées par la commande `ANALYZE` ?

Il existe une méthode plus « confortable » d'accéder aux données de cette table.

Afficher les statistiques collectées pour la table `region`.

Lancer la collecte pour cette table uniquement.

Observer les modifications dans la table système de statistiques.

Réindexation

Recréer les index de la table `appellation`.

Comment recréer tous les index de la base `cave` ?

Comment recréer uniquement les index des tables systèmes ?

Quelle est la différence entre la commande `REINDEX` et la séquence `DROP INDEX` + `CREATE INDEX` ?

Traces

Quelle est la méthode de gestion des logs utilisée par défaut ?

Modifier le fichier `postgresql.conf` pour utiliser le programme interne de rotation des journaux. Les logs doivent désormais être sauvegardés dans le répertoire `/tmp/TP_MAINTENANCE/` et la rotation des journaux être automatisée pour générer un nouveau fichier de logs toutes les 5 minutes, quelle que soit la quantité de logs archivés.

Modifier la configuration pour collecter un maximum d'information dans les journaux.

Ecrire un script shell qui compresse les fichiers de logs vieux de 10 minutes et qui « nettoie » le répertoire en supprimant les journaux vieux de 30 minutes.

5.9.2 SOLUTIONS

Gestion des rôles

Connexion avec l'utilisateur `caviste` (administrateur de la base `cave`) :

```
$ psql -U caviste cave
```

Création de la table `facture` :

```
caviste@cave=> CREATE TABLE facture (id int, objet text, creationts timestamp);
```

Création d'un utilisateur et d'un groupe

Création du rôle `secretariat` avec l'utilisateur `postgres` :

```
postgres@cave=> CREATE ROLE secretariat;
postgres@cave=> GRANT SELECT, INSERT, UPDATE, DELETE ON facture TO secretariat;
```

Création de l'utilisateur bob appartenant au rôle `secretariat` :

```
postgres@cave=> CREATE ROLE bob LOGIN IN ROLE SECRETARIAT;
```

Vérification de la création des deux rôles:

```
postgres@cave=> SELECT * FROM pg_roles;
```

(Vous pouvez aussi utiliser `\du` pour obtenir la liste des utilisateurs, et `\dp` pour connaître les droits des objets).

Connexion avec l'utilisateur bob :

```
$ psql -U bob cave
```

Insertion des factures:

```
bob@cave=> insert into facture values (1, 'Vin de Bordeaux', current_timestamp),
    (2, 'Vin de Bourgogne', current_timestamp);
```

Sélectionner les factures :

```
bob@cave=> select * from facture;
```

Modifier la deuxième facture avec la date et l'heure courante :

```
bob@cave=> update facture set creationts = now() where id = 2;
```

Supprimer la première facture :

```
bob@cave=> delete from facture where id = 1;
```

Modification des permissions

Retirer les droits `DELETE` sur la table facture pour le rôle `secretariat` avec l'utilisateur `postgres` :

```
postgres@cave=> REVOKE DELETE ON facture FROM secretariat;
```

Vérifier qu'il n'est pas possible de supprimer la deuxième facture avec l'utilisateur bob :

```
bob@cave=> delete from facture where id = 2;
```

Retirer tous les droits pour le groupe `secretariat` sur la table `appellation` :

```
postgres@cave=> REVOKE ALL ON appellation from secretariat;
```

Vérifier que l'utilisateur bob appartenant au groupe `secretariat` ne peut pas sélectionner les appellations contenues dans la table `appellation` :

```
bob@cave=> select * from appellation;
```

Autoriser l'utilisateur bob à accéder à la table `appellation` en lecture :

```
postgres@cave=> GRANT SELECT ON appellation to bob;
```

Vérifier que l'utilisateur bob peut désormais accéder à la table `appellation` :

```
bob@cave=> select * from appellation;
```

Héritage des droits au login

Créer un utilisateur `tina` appartenant au rôle `secrétariat` :

```
postgres@cave=> CREATE ROLE tina LOGIN NOINHERIT;
```

Donner les droits du rôle `secretariat` à l'utilisateur `tina` :

```
postgres@cave=> GRANT secretariat to tina;
```

Se connecter avec l'utilisateur `tina` à la base de données cave et vérifier qu'il n'est pas possible d'accéder à la table `facture` :

```
$ psql -U tina cave
```

```
tina@cave=> select * from facture;
```

Positionner le rôle `secretariat` et sélectionner les données de la table `facture` :

```
tina@cave=> set role secretariat;
secretariat@cave=>  select * from facture;
```

L'utilisateur `tina` possède maintenant les droits du rôle `secretariat`.

pg_hba.conf

Pour autoriser vos voisins à accéder à la base, ajouter les lignes suivantes dans le fichier `pg_hba.conf` :

```
# pour le voisin à votre gauche avec l'adresse IP 10.42.X.Y/32
host       cave    tina    10.42.X.Y/32  trust
# pour le voisin à votre droite avec l'adresse IP 10.42.X.Z/32
hostnossl  cave    bob     10.42.X.Z/32  md5
```

VACUUM

Pré-requis

Dans le fichier `postgresql.conf`, désactiver le démon autovacuum en modifiant le paramètre suivant :

```
autovacuum = off
```

Puis, relancer PostgreSQL, par exemple (à adapter en fonction du nom du service utilisé à l'installation) :

```
# service postgresql restart
```

Nettoyage avec VACUUM

Pour visualiser la taille de la table, il suffit d'utiliser la fonction `pg_relation_size` :

```
caviste@cave=> SELECT pg_size_pretty(pg_relation_size('stock'));
```

Retrouver facilement une fonction Il est facile de retrouver facilement une fonction en effectuant une recherche par mot clé dans psql. Exemple :

```
postgres=# \df *pretty*
Liste des fonctions
-[ RECORD 1 ]------------------+---------------
Schéma                         | pg_catalog
Nom                            | pg_size_pretty
```

```
Type de données du résultat    | text
Type de données des paramètres | bigint
Type                           | normal
```

Création d'une copie de la table `stock` :

```
caviste@cave=> CREATE table stock_bis AS SELECT * FROM stock;
```

Suppression des données dans la table :

```
caviste@cave=> DELETE FROM stock;
```

L'espace disque est toujours utilisé :

```
caviste@cave=>  SELECT pg_size_pretty(pg_relation_size('stock'));
```

Réinsertion des données à partir de la copie :

```
caviste@cave=>  INSERT into stock SELECT * FROM stock_bis;
```

L'espace disque utilisé a doublé :

```
caviste@cave=>  SELECT pg_size_pretty(pg_relation_size('stock'));
```

Répéter ces deux dernières commandes plusieurs fois (en terminant par `INSERT`) et observer la taille de la table :

```
caviste@cave=>  SELECT pg_size_pretty(pg_relation_size('stock'));
```

La commande vacuum « nettoie » mais ne « libère » pas d'espace disque :

```
caviste@cave=> VACUUM stock;
...
caviste@cave=> SELECT pg_size_pretty(pg_relation_size('stock'));
```

Vider la table `stock`, insérer à nouveau le contenu de la table `stock_bis` :

```
caviste@cave=> DELETE FROM stock;
-- ...
caviste@cave=> INSERT into stock SELECT * FROM stock_bis;
```

Vérifier la taille de la table, PG a réutilisé des espaces disponibles :

```
caviste@cave=> SELECT pg_size_pretty(pg_relation_size('stock'));
```

Nettoyage avec VACUUM FULL

À nouveau, supprimer et insérer massivement des tuples dans la table puis lancer un `VACUUM FULL` :

```
caviste@cave=> DELETE FROM stock;
-- ...
caviste@cave=> INSERT into stock SELECT * FROM stock_bis;
-- ...
caviste@cave=> DELETE FROM stock;
```

```
-- ...
caviste@cave=> INSERT into stock SELECT * FROM stock_bis;
-- ...
caviste@cave=> DELETE FROM stock;
-- ...
caviste@cave=> INSERT into stock SELECT * FROM stock_bis;
-- ...
caviste@cave=> DELETE FROM stock;
-- ...
caviste@cave=> INSERT into stock SELECT * FROM stock_bis;
-- ...
caviste@cave=> SELECT pg_size_pretty(pg_relation_size('stock'));
-- ...
caviste@cave=> vacuum full stock;
```

Truncate ou Delete ?

On tronque la table `vin_bis`. L'espace disque est immédiatement libéré :

```
cave=> TRUNCATE vin_bis;
TRUNCATE TABLE
cave=> SELECT pg_size_pretty(pg_relation_size('vin_bis'));
pg_size_pretty
----------------
 0 bytes
(1 ligne)
```

Statistiques

Les paramètres de contrôle de la fonction de récupération de statistiques sont :

```
track_activities = on
track_counts = on
```

Les données collectées sont stockées dans la table `pg_statistics`. L'utilisateur `caviste` ne dispose pas des droits nécessaire pour accéder à ces données :

```
$ su - postgres
$ psql cave
...
postgres@cave=> SELECT * FROM pg_statistic;
```

La vue `pg_stats` facilite l'accès à ces données :

```
cave=> SELECT * FROM pg_stats;
```

Pour afficher les informations concernant la table `region`, on exécute la requête suivante :

```
cave=> SELECT * FROM pg_stats WHERE tablename='region';
```

Pour le moment, aucune donnée n'a été collectée pour cette table. On déclenche donc la collecte avec la commande `ANALYZE` :

```
cave=> analyze region;
```

Cette fois, la table `pg_statistics` contient des informations à propos des statistiques de la table `region` :

```
cave=> SELECT * FROM pg_stats WHERE tablename='region';
```

Réindexation

La réindexation d'une table se fait de la manière suivante :

```
REINDEX TABLE appellation;
```

Pour recréer tous les indexes :

```
REINDEX DATABASE cave;
```

Pour réindexer uniquement les tables systèmes :

```
REINDEX SYSTEM cave;
```

`REINDEX` est similaire à une suppression et à une nouvelle création de l'index. Cependant les conditions de verrouillage sont différentes :

- `REINDEX` verrouille les écritures mais pas les lectures de la table mère de l'index. Il prend aussi un verrou exclusif sur l'index en cours de traitement, ce qui bloque les lectures qui tentent d'utiliser l'index.
- Au contraire, `DROP INDEX` crée temporairement un verrou exclusif sur la table parent, bloquant ainsi écritures et lectures. Le `CREATE INDEX` qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets.

Traces

Par défaut le mode de journalisation est **stderr**.

Pour activer le récupérateur interne de log il faut modifier le paramètre suivant dans le fichier `postgresql.conf` :

```
logging_collector = on
```

Puis paramétrer le comportement du récupérateur :

```
log_directory = '/tmp/TP_MAINTENANCE'
# Attention !
# ce repertoire doit être accessible pour l'utilisateur postgres
```

```
log_rotation_age = 5
log_rotation_size = 0
log_min_messages = debug5
```

Un script shell pour gérer la rotation des logs et conserver un volume de logs constant :

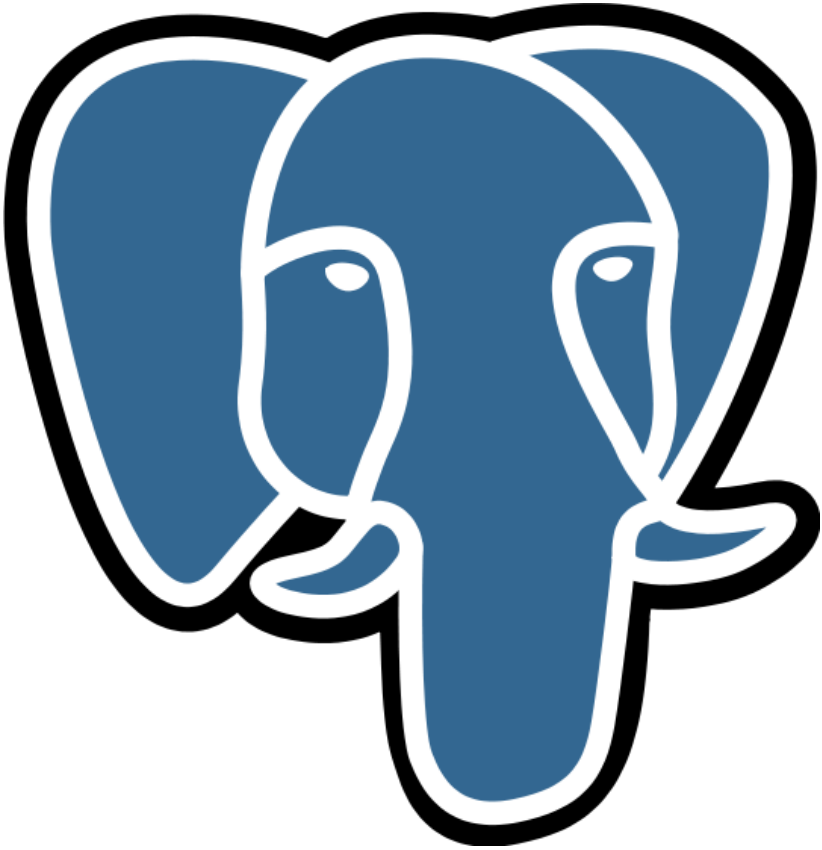
```
#/bin/bash

# On compresse les fichiers vieux de 3 minutes
find /tmp/TP_MAINTENANCE \
    -type f -name "*.log" -cmin +3 -exec gzip -9 {} \; \
    -print

# On supprime les fichiers vieux de 30 minutes
find /tmp/TP_MAINTENANCE \
     -type f -name "*.log.gz" -cmin +30 -exec rm -f {} \; \
     -print
```

Pour fonctionner correctement, ce script doit être exécuté par l'utilisateur ‘`postgres` ou par `root`.

6 POSTGRESQL : SAUVEGARDE / RESTAURATION



6.1 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 bug 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 cohérente. 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.

6.1.1 MENU

- Politique de sauvegarde
- Sauvegarde logique
- Sauvegarde physique

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 ;
- 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 Netbios 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.

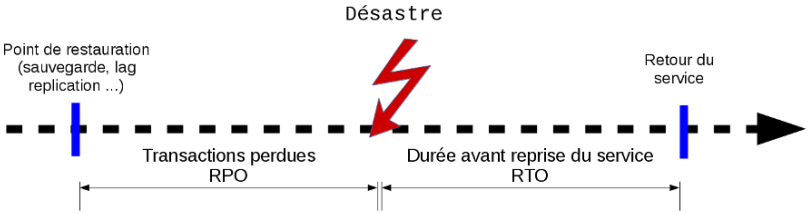
La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données.

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

- RPO (*Recovery Point Objective*) : Perte de Données Maximale Admissible
- RTO (*Recovery Time Objective*) : Durée Maximale d'Interruption Admissible
- => Permettent de définir la politique de sauvegarde/restauration



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

- RPO faible : Le perte de données admissible est très faible voire nulle, il faudra s'orienter vers des solutions de type :
  - Sauvegarde à chaud
  - PITR
  - Réplication (asynchrone/synchrone).
- RPO important : On s'autorise une perte de données importante, on peut utiliser des solutions de type :
  - Sauvegarde logique (dump)
  - Sauvegarde fichier à froid
- RTO court : Durée d'interruption courte, le service doit vite remonter. Nécessite des procédures avec le moins de manipulations possible et réduisant le nombre d'acteurs :
  - Réplication
  - Solutions Haut Disponibilité
- RTO long : La durée de reprise du service n'est pas critique on peut utiliser des solutions simple comme :
  - Restauration fichier
  - Restauration sauvegarde logique (dump).

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

6.3 SAUVEGARDES LOGIQUES

- À chaud
- Cohérente
- Locale ou à distance
- 2 outils
  - `pg_dump`
  - `pg_dumpall`

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 permet de n'avoir qu'un impact faible pour les utilisateurs. En terme de modifications de données, ils peuvent tout faire. En revanche, ce verrou est en conflit avec la suppression des tables présentes au début de la transaction ou la modification de leurs structures. Tout le reste est possible. 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* (à partir de la 9.1) ou *SERIALIZABLE* (pour les versions antérieures) et non de type *READ COMMITED* utilisé par défaut.

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`.

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.

6.3.1 PG_DUMP

- Sauvegarde une base de données
- 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`.

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.

6.3.2 PG_DUMP - FORMAT DE SORTIE

- `-F`
  - `p` : plain, SQL
  - `t` : tar
  - `c` : custom (spécifique PostgreSQL)
  - `d` : directory

`pg_dump` accepte d'enregistrer la sauvegarde suivant quatre formats :

- un fichier SQL, donc un fichier texte dont l'encodage dépend de la base ;
- un répertoire disposant du script SQL et des fichiers, compressés avec gzip, contenant les données de chaque table ;
- un fichier tar intégrant tous les fichiers décrits ci-dessus mais non compressés ;
- un fichier personnalisé, sorte de fichier tar compressé avec gzip.

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 le fichier SQL ;
- `tar` ou `t` pour le fichier tar ;
- `custom` ou `c` pour le fichier personnalisé ;
- `directory` ou `d` pour le répertoire (à partir de la version 9.1).

Le fichier SQL est naturellement lisible par n'importe quel éditeur texte. Le fichier texte est divisé en plusieurs parties :

- configuration de certaines variables ;

- ajout des objets à l'exception des index, des contraintes et triggers (donc schémas, tables, vues, procédures stockées) ;
- ajout des données aux tables ;
- ajout des index, contraintes et triggers ;
- définition des droits d'accès aux objets.

Les index sont la dernière étape 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 le sont parce qu'il faut que les données soient restaurées dans leur ensemble avant de pouvoir mettre en place les contraintes. Les triggers ne devant pas être déclenchés pendant la restauration, ils sont aussi restaurés à 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 car elle est compressée. La sauvegarde au format répertoire se trouve entre la sauvegarde au format personnalisée et la sauvegarde au format tar car 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 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.

Le format `tar` est un vrai format tar, comme le montre la commande `file` :

```
$ file b1.F?.dump
b1.Fc.dump: PostgreSQL custom database dump - v1.12-0
b1.Fd.dump: directory
b1.Fp.dump: UTF-8 Unicode text, with very long lines
b1.Ft.dump: tar archive
```

Il s'agit bien du format POSIX tar[70] , et il en partage les limites. Les fichiers intégrés ne peuvent pas dépasser une taille totale de 8 Go, autrement dit il n'est pas possible, avec une sauvegarde au format tar, de stocker une table dont la représentation physique au niveau de l'archive tar fait plus de 8 Go. Voici l'erreur obtenue :

```
$ pg_dump -Ft tar > tar.Ft.dump
pg_dump: [tar archiver] archive member too large for tar format
```

Cependant, depuis la version 9.5, il est possible de le faire car la plupart des implémentations modernes de tar supportent une extension le permettant. De ce fait, PostgreSQL utilise le format étendu si nécessaire, plutôt que d'échouer.

6.3.3 PG_DUMP - COMPRESSION

- `-Z` : de 0 à 9

Pour éviter de passer beaucoup de temps à compresser ou d'utiliser trop de ressources processeur, il est possible d'agir sur le niveau de compression. Seuls des tests permettent de déterminer le niveau acceptable pour un cas d'utilisation particulier.

Par défaut, `pg_dump` utilise le niveau de compression par défaut de la libz (`Z_DEFAULT_COMPRESSION`) qui correspond au meilleur compromis entre compression et vitesse (actuellement équivalent au niveau 6).

[70] https://fr.wikipedia.org/wiki/Tar_(informatique)

Il est aussi à noter que la compression proposée est zlib. D'autres algorithmes peuvent être plus intéressants (meilleur taux de compression, meilleures performances, utilisation de plusieurs processeurs, etc.).

6.3.4 PG_DUMP - FICHIER OU SORTIE STANDARD

- `-f` : fichier où est stockée la sauvegarde
- sans `-f`, sur la 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.

6.3.5 PG_DUMP - STRUCTURE OU DONNÉES ?

- `--schema-only` ou `-s` : uniquement la structure
- `--data-only` ou `-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`).

6.3.6 PG_DUMP - SÉLECTION DE SECTIONS

- `--section`
  - `pre-data`, la définition des objets (hors contraintes et index)
  - `data`, les données
  - `post-data`, la définition des contraintes et index

17.12

Ces options apparaissent avec la version 9.2.

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 plus intéressant de sauvegarder par section 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.

6.3.7 PG_DUMP - SÉLECTION D'OBJETS

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf celui-là
- `-t <table>` : uniquement cette table
- `-T <table>` : toutes les tables sauf celle-là
- En option
  - possibilité d'en mettre plusieurs
  - exclure les données avec `--exclude-table-data=<table>`
  - avoir une erreur si l'objet est inconnu avec`--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`.

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 $?
```

6.3.8 PG_DUMP - OPTION DE PARALLÉLISATION

- `-j <nombre_de_threads>`

Historiquement, `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. À partir de la version 9.3, l'option `-j` permet de préciser le nombre de connexions réalisées vers la base de données. Chaque connexion est gérée dans `pg_dump` par un processus sous Unix et par un thread sous Windows. Par contre, cette option est compatible uniquement avec le format de sortie *directory* (option `-F d`).

Cela permet d'améliorer considérablement la vitesse de sauvegarde.

Dans certains cas, il est possible qu'il n'y ait pas de gain. 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 change fondamentalement la durée de sauvegarde.

Il est à noter que, même si la version 9.6 de PostgreSQL apporte la parallélisation de certains types de requêtes, cela ne concerne pas la commande `COPY` et, de ce fait, `pg_dump` ne peut pas en bénéficier.

6.3.9 PG_DUMP - OPTIONS DIVERSES

- `-O` : ignorer le propriétaire
- `-x` : ignorer les droits
- `--no-tablespaces` : ignorer les tablespaces
- `--inserts` : remplacer COPY par INSERT
- `-v` : pour voir la progression

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques. Les trois premières (`--no-owner`, `--no-privileges` et `--no-tablespaces`) permettent de ne pas indiquer respectivement le propriétaire, les droits et le tablespace de l'objet dans la sauvegarde.

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`.

Enfin, l'option `-v` (ou `--verbose`) permet de voir la progression de la commande.

6.3.10 PG_DUMPALL

- Sauvegarde d'une instance complète
  - Objets globaux
  - Bases de données

`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.

6.3.11 PG_DUMPALL - FORMAT DE SORTIE

- `-F`
  - `p` : plain, SQL

Contrairement à `pg_dump`, `pg_dumpall` ne dispose que d'un format en sortie : le fichier SQL. L'option est donc inutile.

6.3.12 PG_DUMPALL - FICHIER OU SORTIE STANDARD

- `-f` : fichier où est stockée la sauvegarde
- sans `-f`, sur la 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.

6.3.13 PG_DUMPALL - SÉLECTION DES OBJETS

- `-g` : tous les objets globaux
- `-r` : uniquement les rôles
- `-t` : uniquement les tablespaces
- `--no-role-passwords` : pour ne pas sauvegarder 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;
```

17.12

```
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;



--
-- 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
--
```

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. Pour permettre d'utiliser `pg_dumpall` sans avoir un rôle superutilisateur, il faut ajouter 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. Cependant, cette option n'est disponible qu'à partir de la version 10.

6.3.14 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`.

6.3.15 PG_DUMP/PG_DUMPALL - OPTIONS DE CONNEXIONS

- `-h` / `$PGHOST` / socket Unix
- `-p` / `$PGPORT` / 5432
- `-U` / `$PGUSER` / utilisateur du système
- `$PGPASSWORD`
- `.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. Il force seulement psql à demander un mot de passe (en interactif donc).

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 esclave,maitre -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. L'outil vérifiera cela avant d'accepter d'utiliser les informations qui s'y trouvent.

6.3.16 TRAITER AUTOMATIQUEMENT LA SORTIE

- Pour compresser : `pg_dump | bzip2`
- Il existe des outils multi-thread de compression, bien plus rapides:
  - pbzip2
  - pigz

Dans le cas de la sauvegarde au format texte, il est possible de traiter la sortie standard pour aller plus loin.

Par exemple, il est possible de compresser la sauvegarde texte immédiatement. Cela vous permet d'utiliser des outils de compression comme `bzip2` ou `lzma` qui ont une compression plus forte (au prix d'une exécution plus longue) ou comme `pigz`, `pbzip2` (http://compression.ca/pbzip2/) qui compressent plus rapidement grâce à l'utilisation de plusieurs threads, au prix d'un très léger surcoût en taille.

Par exemple, la version 9.1 met 134 minutes à faire une sauvegarde au format custom d'une base de 110 Go, alors qu'alliée avec `pigz`, elle ne met plus que 45 minutes (soit trois fois moins). Voici la commande intégrant `pigz` :

```
$ pg_dump -Fc -Z0 -v foobar | pigz > sauvegarde.dump.gzip
```

L'option `-Z0` est obligatoire pour s'assurer de ne pas compresser en amont de pigz.

On peut aussi utiliser n'importe quel autre outil Unix. Par exemple, pour répartir sur plusieurs fichiers : `pg_dump | split`

6.3.17 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.

À partir de la version 9.0, 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`. La valeur par défaut (`hex`) ne correspond pas à la valeur en dur des versions précédentes (`escape`). D'où des comportements différents lors du passage d'une version antérieure à la 9.0 vers la version 9.0 ou une version ultérieure. Le format `hex` utilise des 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).

6.4 RESTAURATION D'UNE SAUVEGARDE LOGIQUE

- Sauvegarde texte (option p) : psql
- Sauvegarde binaire (options t, c ou d) : pg_restore

`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.

6.4.1 PSQL

- client standard PostgreSQL
- capable d'exécuter des requêtes
- donc capable de restaurer une sauvegarde au format texte
- 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 exécutable via `psql`.

6.4.2 PSQL - OPTIONS

- `-f` pour indiquer le fichier contenant la sauvegarde
  - sans option `-f`, lit l'entrée standard
- `-1` pour tout restaurer en une seule transaction
- `-e` pour afficher les ordres SQL exécutés
- `ON_ERROR_ROLLBACK`/`ON_ERROR_STOP`

Pour cela, il existe plusieurs moyens :

- 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 i:

```
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, après une erreur, les requêtes précédentes sont toujours 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 préférable parfois 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 possibles, mais 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 (valeur off), dans **une transaction**, toute erreur entraîne le ROLLBACK de toute la transaction. Activer ce paramètre permet que seule la commande en erreur soit annulée. psql effectue des *savepoints* avant chaque ordre, et y retourne en cas d'erreur, avant de continuer le script. Ceci est utile surtout si vous avez utilisé l'option -1. Il 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.

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'
```

- de façon interactive dans psql:

```
psql>\set ON_ERROR_ROLLBACK on
```

6.4.3 PG_RESTORE

- restaure uniquement les sauvegardes au format binaire
  - donc tar, custom ou directory
  - format autodétecté (-F inutile, même si présent)
- 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 :

17.12

```
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.

6.4.4 PG_RESTORE - FICHIER OU ENTRÉE STANDARD

- Fichier à restaurer en dernier argument de la ligne de commande
- Attention à `-f` (fichier en sortie)

Le fichier à restaurer s'indique en dernier argument sur la ligne de commande.

L'option `-f` permet d'indiquer un fichier qui contiendra un script SQL correspondant aux ordres à générer à la restauration. Il sera donc écrit. S 'il est utilisé pour indiquer le nom du fichier de sauvegarde à restaurer, ce dernier se verra vidé pour contenir les erreurs éventuels de la restauration. Autrement dit, il faut faire extrêmement attention lors de l'utilisation de l'option `-f` avec l'outil `pg_restore`.

Utilisation de l'option `-f`

Il est déconseillé d'utiliser l'option `-f` avec `pg_restore` ! Cette dernière est contre intuitive : elle indique le journal d'activité de `pg_restore` et **NON** le fichier à restaurer. Il est ainsi toujours recommandé d'utiliser simplement la redirection standard qui ne portent jamais à confusion pour récupérer les messages de `pg_restore`.

6.4.5 PG_RESTORE - STRUCTURE OU DONNÉES ?

- `-s` : uniquement la structure
- `-a` : uniquement les données

Comme pour `pg_dump`, les options `-s` et `-a` permettront de sélectionner une restauration de la structure seule ou des données seules.

6.4.6 PG_RESTORE - SÉLECTION DE SECTIONS

- `--section`
  - `pre-data`, la définition des objets (hors contraintes et index)
  - `data`, les données

  - `post-data`, la définition des contraintes et index

Ces options apparaissent avec la version 9.2.

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, ce qui accélère la restauration.

6.4.7 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 est à noter que l'option `-t` ne concernait que les tables avant la version 9.6. À partir de cette version, elle concerne toutes les relations : tables, vues, vues matérialisées et séquences.

Il existe en plus les options `-I` et `-P` (respectivement `--index` et `--function`) pour restaurer respectivement un index et une procédure 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 :

17.12

```
$ 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
```

6.4.8 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 Tue Dec 13 09:35:17 2011
;     dbname: b1
;     TOC Entries: 16
;     Compression: -1
;     Dump Version: 1.12-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 9.1.3
;     Dumped by pg_dump version: 9.1.3
;
;
; Selected TOC Entries:
;
2752; 1262 37147 DATABASE - b1 guillaume
5; 2615 2200 SCHEMA - public guillaume
```

```
2753; 0 0 COMMENT - SCHEMA public guillaume
2754; 0 0 ACL - public guillaume
164; 3079 12528 EXTENSION - plpgsql
2755; 0 0 COMMENT - EXTENSION plpgsql
165; 1255 37161 FUNCTION public f1() guillaume
161; 1259 37148 TABLE public t1 guillaume
162; 1259 37151 TABLE public t2 guillaume
163; 1259 37157 VIEW public v1 guillaume
2748; 0 37148 TABLE DATA public t1 guillaume
2749; 0 37151 TABLE DATA public t2 guillaume
2747; 2606 37163 CONSTRAINT public t2_pkey guillaume
2745; 1259 37164 INDEX public t1_c1_idx guillaume
```

Toutes les lignes qui commencent avec un point-virgule sont des commentaires. Le reste indique les objets à créer : un schéma public, le langage plpgsql, la procédure stockée f1, 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.

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 SCHEMA public
pg_restore: creating COMMENT SCHEMA public
pg_restore: creating EXTENSION plpgsql
pg_restore: creating COMMENT EXTENSION plpgsql
pg_restore: creating TABLE t1
```

```
pg_restore: creating TABLE t2
pg_restore: creating VIEW v1
pg_restore: restoring data for table "t1"
pg_restore: creating CONSTRAINT t2_pkey
pg_restore: setting owner and privileges for SCHEMA public
pg_restore: setting owner and privileges for COMMENT SCHEMA public
pg_restore: setting owner and privileges for ACL public
pg_restore: setting owner and privileges for EXTENSION plpgsql
pg_restore: setting owner and privileges for COMMENT EXTENSION plpgsql
pg_restore: setting owner and privileges for TABLE t1
pg_restore: setting owner and privileges for TABLE t2
pg_restore: setting owner and privileges for VIEW v1
pg_restore: setting owner and privileges for TABLE DATA t1
pg_restore: setting owner and privileges for CONSTRAINT t2_pkey
```

L'option `-v` de `pg_restore` permet de visualiser sa progression dans la restauration. On remarque bien que la procédure stockée f1 ne fait pas partie des objets restaurés. Tout comme l'index sur t1 et les données de la table t2.

6.4.9 PG_RESTORE - OPTION DE PARALLÉLISATION

- `-j <nombre_de_threads>`

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. À partir de la version 8.4, 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. Un test effectué a montré qu'une restauration d'une base de 150 Go prenait 5 h avec une seule connexion, mais seulement 3h avec plusieurs connexions.

Dans certains cas, 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, il y a peu de chance que la parallélisation change fondamentalement la durée de restauration.

Il est à noter que, même si la version 9.6 de PostgreSQL apporte la parallélisation de certains types de requêtes, cela ne concerne pas la commande `COPY` et, de ce fait,

`pg_restore` ne peut pas en bénéficier.

6.4.10 PG_RESTORE - OPTIONS DIVERSES

- `-O` : ignorer le propriétaire
- `-x` : ignorer les droits
- `--no-tablespaces` : ignorer le tablespace
- `-1` pour tout restaurer en une seule transaction
- `-c` : pour détruire un objet avant de le restaurer

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques. Les trois premières (`-O`, `-x` et `--no-tablespaces`) permettent de ne pas restaurer respectivement le propriétaire, les droits 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.

Enfin, l'option `-v` permet de voir la progression de la commande.

6.5 AUTRES CONSIDÉRATIONS SUR LA SAUVEGARDE LOGIQUE

- Script de sauvegarde
- Sauvegarder sans passer par un fichier
- Gestion des statistiques sur les données
- 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.

6.5.1 SCRIPT DE SAUVEGARDE IDÉAL

- pg_dumpall -g
- suivi d'un appel à pg_dump -Fc pour chaque base

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. Voici un exemple de script qui fait cela :

```
#!/bin/sh
# Script de sauvegarde pour PostgreSQL

REQ="SELECT datname FROM pg_database WHERE datallowconn ORDER BY datname"

pg_dumpall -g > globals.dump
psql -Atc "$REQ" postgres | while read base
do
  pg_dump -Fc $base > ${base}.dump
done
```

Évidemment, il ne conviendra pas à tous les cas mais il donne une idée de ce qu'il est possible de faire.

Exemple de script de sauvegarde adapté pour un serveur Windows :

```
@echo off

SET PGPASSWORD=super_password
SET PATH=%PATH%;C:\Progra~1\PostgreSQL\9.1\bin\

pg_dumpall -g -U postgres postgres > c:\pg-globals.sql

for /F %%v in ('psql -At -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
```

Attention: pg_dumpall -g suivi de pg_dump ne sauvegardera pas tout ce qui se trouve

dans un cluster. Il y a quelques rares exceptions:

- 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, et `pg_dump` n'exporte pas ce paramètre ;
- 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 ils ne seront pas restaurés, car les bases n'existeront pas au moment où l'`ALTER ROLE` sera exécuté.

Autre exemple de script réalisant une sauvegarde totale de toutes les bases avec une période de rétention des sauvegardes :

```
#!/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 prefixe the name of all backup file
PREFIX=bkup
# Set it to save a remote server, default to unix socket
HOSTNAME=""
# Set it to set the remote user to login
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 database
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 addition 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;

# Dumping 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
# Dumping 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 database 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
```

6.6 PG_BACK - PRÉSENTATION

- Type de sauvegardes: **logiques** (pg_dump)
- Langage: **bash**
- Licence: **BSD** (libre)
- Type de stockage: **local**
- Planification: **crontab**
- OS: **Unix/Linux**
- Compression: **gzip**
- Versions compatibles: **toutes**
- Rétention: **durée** en jour

pg_back[71] est un outil écrit en bash par Nicolas Thauvin de Dalibo, également auteur de pitrery.

Il s'agit d'un script assez complet permettant de gérer simplement des sauvegardes logiques (`pg_dump`, `pg_dumpall`), y compris au niveau de la rétention.

L'outil se veut très simple et facile à adapter, il peut donc être facilement modifié pour mettre en place une stratégie de sauvegarde logique plus complexe.

L'outil ne propose pas d'options pour restaurer les données. Il faut pour cela utiliser les outils interne de PostgreSQL (`pg_restore`, `psql`).

[71] https://github.com/orgrim/pg_back

6.6.1 SAUVEGARDE ET RESTAURATION, SANS FICHIER INTERMÉDIAIRE

- pg_dump | pg_restore
- Utilisation des options -h et -p

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` à `psql` ou à `pg_restore`. Par exemple :

```
$ createdb nouvelleb1
$ pg_dump b1 | psql nouvelleb1
```

Ces deux commandes permettent de dupliquer b1 dans nouvelleb1.

Notez aussi que l'utilisation des options `-h` et `-p` permet de sauvegarder et restaurer une base en un seul temps sur des instances différentes, qu'elles soient locales ou à distance.

6.6.2 STATISTIQUES SUR LES DONNÉES

- Ne fait pas partie de la sauvegarde
- Donc, `ANALYZE` après une restauration

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.

Par ailleurs, après un chargement, quelques bits vont être mis à jour pour chaque enregistrement à la relecture suivante (ils sont appelés «hint bits», c'est à dire bits d'indice), et donc générer de nombreuses écritures. Il est donc préférable d'anticiper ces écritures et de les réaliser avant la mise en production de la base.

L'autovacuum diminue ce risque car il va exécuter automatiquement un `ANALYZE` sur les grosses tables. Malgré cela, il est préférable de lancer un `VACUUM ANALYZE` manuel à la fin de la restauration, afin de procéder immédiatement à la réécriture des tables et au passage des statistiques.

L'`ANALYZE` est impératif, le `VACUUM` optionnel (le positionnement des hint bits peut avoir lieu durant la production, le coût sera élevé mais pas prohibitif), sachant que l'`ANALYZE` seul étant probablement entre 10 et 50 fois plus rapide que le `VACUUM ANALYZE`.

Il est à noter que si vous ne souhaitez que l'`ANALYZE`, cette opération est plus simple à réaliser à partir de la version 9.1, grâce à l'option `-Z` (ou `--analyze-only`) de l'outil `vacuumdb`.

6.6.3 DURÉE D'EXÉCUTION

- Difficile à chiffrer
- Dépend de l'activité sur le serveur

6.6.4 TAILLE D'UNE SAUVEGARDE LOGIQUE

- Difficile à évaluer
- Le contenu des index n'est pas sauvegardé
  - Donc sauvegarde plus petite
- Les objets binaires prennent plus place
  - 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 quatre octets), soit de l'hexadécimal (trois octets, à partir de la version 9.0). Donc un objet binaire prend 3 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.

6.7 SAUVEGARDE AU NIVEAU SYSTÈME DE FICHIERS

- À froid

- Donc cohérente
- Beaucoup d'outils
  - aucun spécifique à PostgreSQL
- Attention à ne pas oublier les tablespaces
- Possibilité de la faire à chaud (*PITR*)
  - nécessite d'avoir activé l'archivage des WAL
  - technique avancée, complexe à mettre en place et à maintenir
  - pas de coupure de service

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 arrêter PostgreSQL pour disposer d'une sauvegarde cohérente si la sauvegarde se fait au niveau du système de fichiers.

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.

Il est cependant essentiel d'être attentif aux données qui ne se trouvent pas directement dans le répertoire des données. Notamment le répertoire des journaux de transactions, qui est souvent placé dans un autre système de fichiers pour gagner en performances. Si c'est le cas et que ce répertoire n'est pas sauvegardé, la sauvegarde ne sera pas utilisable. De même, si des tablespaces sont créés, il est essentiel d'intégrer ces autres répertoires dans la sauvegarde de fichiers. Dans le cas contraire, une partie des données manquera.

Voici un exemple de sauvegarde :

```
$ /etc/init.d/postgresql stop
$ tar cvfj data.tar.bz2 /var/lib/postgresql
$ /etc/init.d/postgresql start
```

Il est possible de réaliser les sauvegardes de fichiers sans arrêter l'instance (*à chaud*), mais il s'agit 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 fichiers WAL.

6.7.1 AVANTAGES

- 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.

6.7.2 INCONVÉNIENTS

- 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

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 LittleEndian/BigEndian doit être respectée.

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.

6.7.3 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. Voici un exemple de ce cas d'utilisation :

```
$ rsync /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql stop
$ rsync -av /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql start
```

6.7.4 SNAPSHOT DE PARTITION

- Avec certains systèmes de fichiers
- Avec LVM
- Avec la majorité des SAN

Certains systèmes de fichiers (principalement ZFS et le système de fichiers en cours de développement 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.

6.8 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

6.9 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.

6.10 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.

17.12

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.

6.10.1 QUESTIONS

N'hésitez pas, c'est le moment !

6.11 TRAVAUX PRATIQUES

6.11.1 ÉNONCÉS

Sauvegardes logiques

Créer un répertoire `backups` dans le répertoire home de `postgres` pour y déposer les fichiers d'export.

Sauvegarde logique de toutes les bases

Sauvegarder toutes les bases de données de l'instance PostgreSQL à l'aide de `pg_dumpall` dans le fichier `~postgres/backups/base_all.sql.gz`.

La sauvegarde doit être compressée.

Sauvegarde logique d'une base

Sauvegarder la base de données `cave` au format *custom* à l'aide de `pg_dump` dans le fichier `~postgres/backups/base_cave.dump`.

Export des objets globaux

Exporter uniquement les objets globaux de l'instance (rôles et définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier `~postgres/backups/base_globals.sql`.

Sauvegarde logique de tables

Sauvegarder la table `stock` dans le fichier `~postgres/backups/table_stock.sql` au format *plain text*.

Créer un script nommé `sauvegarde_tables.sh` qui sauvegarde toutes les tables du schéma `public` au format *plain text* d'une base dans des fichiers séparés nommés `table_<nom_table>.sql`.

Restaurations logiques

Restauration d'une base de données

Restaurer la base de données `cave` dans une nouvelle base de données nommée `cave2` en utilisant le fichier de sauvegarde `base_cave.dump`. Puis renommer la base de données `cave` en `cave_old` et la base de données `cave2` en `cave`.

Restauration d'une table

À partir de la sauvegarde de la partie 1.2 (sauvegarde de la base `cave` au format *custom*), restaurer uniquement la table `stock`. Attention, vous avez peut-être, suite à un TP antérieur, une vue s'appuyant sur `stock`, ce qui vous empêchera de faire un `DROP` sur cette table. Si nécessaire, supprimer cette vue (`DROP VIEW`) auparavant.

Migration de données

Copier les données de la base `cave` dans une nouvelle base `cave_test` sans passer par un fichier de sauvegarde.

Restauration partielle

Restaurer dans une base `cave3` tout le contenu de la sauvegarde de la base `cave`, sauf les données de la table `stock`.

La définition de la table `stock` et toutes les contraintes s'y rapportant doivent être restaurées.

6.11.2 SOLUTIONS

Sauvegardes logiques

Créer un répertoire `backups` dans le répertoire `home` de `postgres` pour y déposer les fichiers d'export.

Se logguer avec l'utilisateur `postgres`, puis exécuter les commandes suivantes :

```
cd ~postgres
mkdir backups
```

Sauvegarde logique de toutes les bases

Sauvegarder toutes les bases de données du cluster PostgreSQL à l'aide de `pg_dumpall` dans le fichier : `~postgres/backups/base_all.sql.gz`

Se logguer avec l'utilisateur `postgres`, puis exécuter la commande suivante :

```
pg_dumpall | gzip > ~postgres/backups/base_all.sql.gz
```

Sauvegarde logique d'une base

Sauvegarder la base de données `cave` au format *custom* à l'aide de `pg_dump` dans le fichier `~postgres/backups/base_cave.dump`

Se connecter avec l'utilisateur `postgres`, puis exécuter la commande suivante :

```
pg_dump -Fc  -f ~postgres/backups/base_cave.dump cave
```

Export des objets globaux

Exporter uniquement les objets globaux de l'instance (rôles et définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier `~postgres/backups/base_globals.sql`.

```
pg_dumpall -g > ~postgres/backups/base_globals.sql
```

Sauvegarde logique de tables

Sauvegarder la table `stock` dans le fichier `~postgres/backups/table_stock.sql` :

```
pg_dump -t stock cave > ~postgres/backups/table_stock.sql
```

Créer un script nommé `sauvegarde_tables.sh` qui sauvegarde toutes les tables du schéma `public` au format *plain text* d'une base dans des fichiers séparés nommés `table_<nom_table>.sql`.

En tant qu'utilisateur système `postgres`, créer le script avec un éditeur de texte comme `vi` :

```
vi ~postgres/backups/sauvegarde_tables.sh
```

Un exemple de contenu pour ce script pourrait être le suivant :

```
#!/bin/bash
LIST_TABLES=$(psql -U caviste -At -c '\dt public.*' cave | awk -F' \
              |' '{print $2}')
for t in $LIST_TABLES; do
  echo "sauvegarde $t..."
  pg_dump -t $t cave > ~postgres/backups/table_${t}.sql
done
```

Autre solution possible, incluant le schéma dans le nom des fichiers:

```
#!/bin/bash
for t in $(psql -At -F'.' -c "SELECT schemaname, tablename FROM pg_tables
                              WHERE schemaname = 'public'" cave)
do
  pg_dump -t "$t" cave > ~postgres/backups/table_$t.sql
done
```

Enfin, il faut donner les droits d'exécution au script, et l'exécuter :

```
chmod +x ~postgres/backups/sauvegarde_tables.sh
~postgres/backups/sauvegarde_tables.sh
```

Restaurations logiques

Restauration d'une base de données

Restaurer la base de données `cave` dans une nouvelle base de données nommée `cave2` en utilisant le fichier de sauvegarde `base_cave.dump` :

```
createdb -O caviste cave2
pg_restore -d cave2 ~postgres/backups/base_cave.dump
```

Se connecter à l'instance PostgreSQL, et renommer la base de données `cave` en `cave_old` et la base de données `cave2` en `cave` :

```
ALTER DATABASE cave RENAME TO cave_old;

ALTER DATABASE cave2 RENAME TO cave;
```

Restauration d'une table

À partir de la sauvegarde de la partie 1.2 (sauvegarde de la base `cave` au format *custom*), restaurer uniquement la table `stock` :

```
pg_restore -d cave -c -t stock ~postgres/backups/base_cave.dump
```

Attention, il se pourrait que suite à un TP d'un module précédent, vous ayez une vue définie sur `stock`. Il faudra la supprimer (DROP VIEW).

Migration de données

Copier les données de la base `cave` dans une nouvelle base `cave_test` sans passer par un fichier de sauvegarde :

```
createdb -O caviste cave_test
pg_dump -Fc cave | pg_restore -d cave_test
```

Restauration partielle

Lister le contenu de l'archive dans un fichier:

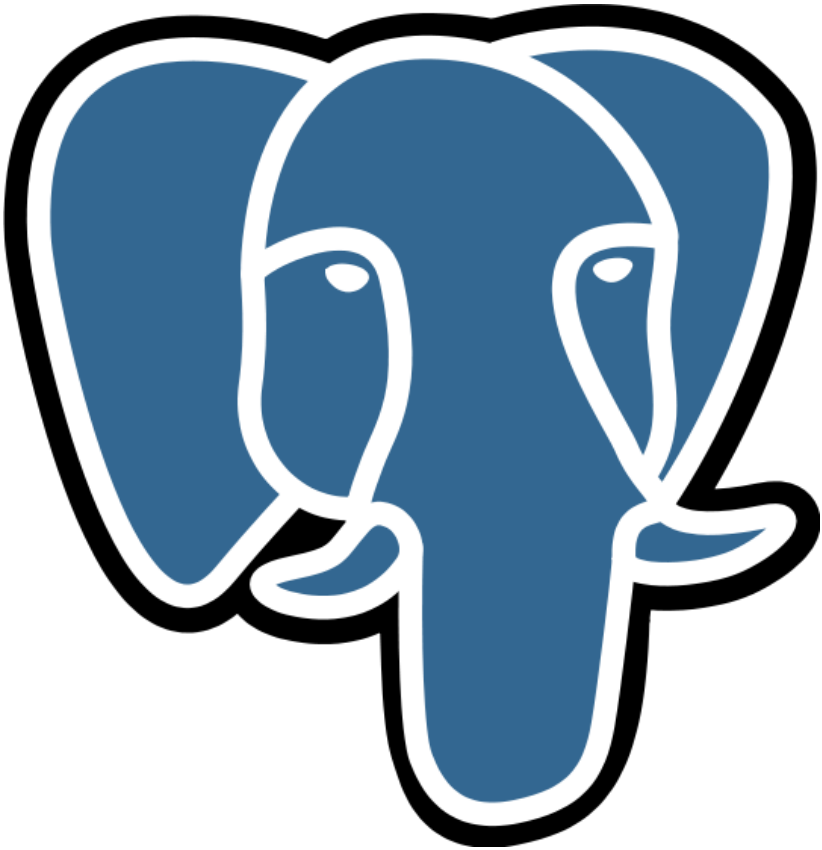
```
pg_restore -l ~postgres/backups/base_cave.dump > /tmp/contenu_archive
```

Éditer /tmp/contenu_archive, et supprimer ou commenter la ligne TABLE DATA de la table `stock`.

Créer la base de données `cave3`, puis restaurer en utilisant ce fichier :

```
createdb -O caviste cave3
pg_restore -L /tmp/contenu_archive -d cave3 ~postgres/backups/base_cave.dump
```

7 SUPERVISION



7.1 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 (permettant l'envoi de messages à une personne en astreinte).

7.1.1 MENU

- Politique de supervision
- Supervision occasionnelle
- Supervision automatique

7.2 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, son objectif et du coup 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.

7.2.1 OBJECTIFS DE LA SUPERVISION

- Améliorer 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 le problème de performances. 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.

7.2.2 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).

7.2.3 EXEMPLES D'INDICATEURS - SYSTÈME D'EXPLOITATION

- Charge CPU
- Entrées/sorties disque
- Espace disque
- Sur-activité et non-activité du serveur
- Temps de réponse

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 soucis au niveau du système disque : soit PostgreSQL écrit trop à cause d'une mauvaise configuration des journaux de transactions, soit les requêtes exécutées utilisent des fichiers temporaires pour trier les données, ou pour une toute autre raison.

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.

Il est possible aussi d'utiliser une requête étalon dont la durée d'exécution sera testée de temps à autre pour détecter les moments problématiques sur le serveur.

7.2.4 EXEMPLES D'INDICATEURS - BASE DE DONNÉES

- Nombre de connexions
- Requêtes lentes et/ou fréquentes
- Ratio d'utilisation du cache

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), et quelques ratios (utilisation du cache par exemple).

7.3 LA SUPERVISION AVEC POSTGRESQL

- Supervision occasionnelle : pour les cas où il est possible d'intervenir immédiatement
- Supervision automatique
  - permet de remonter des informations rapidement
  - permet de conserver les informations

La supervision occasionnelle est intéressante lorsqu'un utilisateur se plaint d 'un problème survenant maintenant. Cependant, cela reste assez limité.

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 envoyer des alertes quand des seuils sont dépassés.

7.3.1 INFORMATIONS INTERNES

- PostgreSQL propose deux canaux d'informations :
  - les statistiques
  - les traces
- Mais rien pour les conserver, les historiser

PostgreSQL propose deux canaux d'informations : les statistiques d'activité et les traces applicatives.

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.

7.3.2 OUTILS EXTERNES

- Nécessaire pour conserver les informations
- ... et exécuter automatiquement des actions dessus
  - Génération de graphiques (Munin, Zabbix, OPM)
  - 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.

7.3.3 CHECK_POSTGRES

- Script de monitoring PostgreSQL pour Nagios ou MRTG
  - nombreuses sondes spécifiques
  - nombreuses données de performance remontées
- http://bucardo.org/wiki/Check_postgres

Le script de monitoring `check_postgres` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios.

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. Voici les sondes `check_postgres` à mettre en place sur ces différents aspects.

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.

Vacuum :

- `bloat` : vérifie le volume de données « mortes » et la fragmentation des tables et des 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` : vérifie le nombre de verrous
- `wal_files` : Compte le nombre de segments du journal de transaction présents dans le répertoire `pg_wal`.
- `query_time` : identifie les requêtes en cours d'exécution depuis trop longtemps.

Configuration :

- `settings_checksum` : indique si la configuration a été modifiée depuis la dernière vérification.

Réplication :

- `archive_ready` : compte le nombre de segments du journal de transaction en attente d'archivage.
- `hot_standby_delay` : calcule le délai de réplication entre un serveur maître et un esclave.

7.3.4 CHECK_PGACTIVITY

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - reprend la plupart des sondes de `check_postgres`, en corrigeant certaines
  - davantage de données de performance remontées
- Peut être utilisé en remplacement de `check_postgres`
- Développé par Dalibo pour les besoins de l'outil OPM
  - mais peut être utilisé indépendamment

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

Tout comme le script `check_postgres`, le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios.

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 l'outil OPM (*Open PostgreSQL Monitoring*), apportant notamment un nombre plus important de métriques de performances. Il utilise notamment un fichier local qui permet de retourner directement des ratios par rapport à la valeur précédente plutôt que des valeurs fixes. Le script peut être utilisé de façon autonome, en remplacement de `check_postgres`, sans nécessité d'installer toute l'infrastructure OPM.

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. Voici les sondes `check_pgactivity` à mettre en place sur ces différents aspects.

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`.
- `backend_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*).

Vacuum :

- `bloat` : vérifie le volume de données « mortes » et la fragmentation des tables - requête entièrement réécrite par rapport à celle utilisée par `check_postgres`.
- `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.
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL.

Configuration :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut.

Réplication :

- `ready_archives` : compte le nombre de segments du journal de transaction en attente d'archivage.
- `hot_standby_delta` : calcule le délai de réplication entre un serveur maître et un esclave.
- `is_master` : vérifie que l'instance est bien démarrée en lecture/écriture.
- `is_hot_standby` : vérifie que l'instance est en *recovery* et accepte les requêtes en lecture.

7.4 TRACES

- Configuration
- Récupération
  - des problèmes importants
  - des requêtes lentes/fréquentes
- Outils externes de classement

La première information que fournit PostgreSQL quant à l'activité sur le serveur est 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 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 quelques informations intéressantes à récupérer. Enfin,

nous verrons quelques outils permettant de traiter automatiquement les fichiers de trace.

7.4.1 CONFIGURATION

- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
- Tracer certains comportements

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas en même temps trop nombreuses pour ne pas être submergé par les informations et trop peu pour ne pas savoir ce qu'il se passe. Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

7.4.2 CONFIGURATION : OÙ TRACER

- `log_destination`
- `logging_collector`
  - `log_directory`, `log_filename`, `log_file_mode`
  - `log_rotation_age`, `log_rotation_size`, `log_truncate_on_rotation`
- `syslog` (Unix)
- `eventlog` (Windows)

PostgreSQL peut envoyer les traces sur plusieurs destinations :

- `syslog`, fonctionne uniquement sur un serveur Unix, et est intéressant pour rassembler la configuration des traces ;
- `eventlog`, disponible uniquement sur Windows ;
- `stderr` et `csvlog`, disponible sur toutes les plateformes, correspond à la sortie des erreurs. La différence entre les deux réside dans le format des traces (texte simple ou CSV).

Pour `eventlog`, un dernier paramètre est disponible depuis la version 9.2. Ce paramètre, appelé `event_source`, correspond au nom du programme indiqué pour identifier les messages de PostgreSQL dans les traces. Par défaut, la valeur est PostgreSQL.

Si les traces sont envoyées à `syslog`, il reste à configurer 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. Deux nouveaux paramètres apparaissent avec la version 9.6. `syslog_sequence_numbers` préfixe chaque message d'un numéro de séquence incrémenté automatiquement. Ceci permet d'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`, il détermine la façon dont les messages sont envoyées à syslog. S'il est activé, les messages sont divisés par lignes et les lignes longues sont divisées pour tenir sur 1024 octets. Les autres paramètres de cette page n'ont pas d'intérêt avec la destination `syslog`.

Les autres paramètres ne concernent que la destination `stderr` et `csvlog`. Il est possible d'indiquer l'emplacement des journaux applicatifs (paramètre `log_directory`), de spécifier le motif de leur nom (avec `log_filename`), ainsi que les droits des fichiers (avec `log_file_mode`). Une rotation est configurable suivant la taille (`log_rotation_size`) et la durée de vie (`log_rotation_age`).

Voici un graphe récapitulant les différentes possibilités :

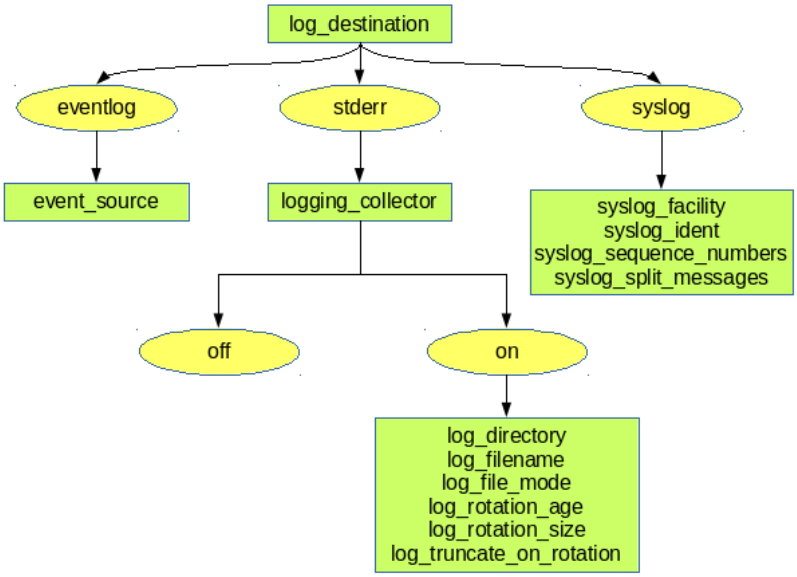


FIGURE 3: GESTION DES TRACES

7.4.3 CONFIGURATION : NIVEAU DES TRACES

- `log_min_messages`
- `log_min_error_statement`
- `log_error_verbosity`

`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 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.

7.4.4 CONFIGURATION : TRACER LES REQUÊTES

- `log_min_duration_statement`
- `log_statement`
- `log_duration`

Pour les problèmes de performances, il est intéressant de pouvoir tracer les requêtes et leur durée d'exécution. PostgreSQL propose deux solutions à cela.

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 est préférable dans ce cas de configurer `log_statement` à la

valeur `all`. Dans le cas contraire, il est impossible de dire la requête qui a pris ce temps d'exécution.

Donc pour tracer toutes les requêtes et leur durée d'exécution, une solution serait de réaliser la configuration suivante :

```
log_statements = 'all'
log_duration = on
```

Cela génère deux entrées dans les traces, de cette façon :

```
2016-09-01 13:34:11 CEST LOG:  statement: SELECT * FROM pg_stat_activity;
2016-09-01 13:34:11 CEST LOG:  duration: 0.923 ms
```

Parfois, on ne souhaite garder une trace que des requêtes très lentes, par exemple celles qui prennent plus de deux secondes à s'exécuter. Le paramètre `log_min_duration_statement` est là pour ça. Sa valeur correspond au nombre de millisecondes maximum avant qu'une requête ne soit tracée. Par exemple, avec cette configuration :

```
log_min_duration_statement = 2000
```

on pourrait obtenir cette trace :

```
2016-09-01 13:37:01 CEST LOG:  duration: 2906.270 ms  statement:
                               insert into t1 select i, i
                               from generate_series(1, 200000) as i;
```

Remarquez que, cette fois-ci, la requête et la durée d'exécution sont indiquées dans le même message.

7.4.5 CONFIGURATION : TRACER CERTAINS COMPORTEMENTS

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits`
- `log_temp_files`

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements.

Quand on souhaite avoir une trace de qui se connecte, il est intéressant de pouvoir tracer les connexions et, parfois aussi, les déconnexions. En activant le paramètre

log_connections, nous obtenons les traces suivantes pour une connexion réussie :

```
2016-09-01 13:34:32 CEST LOG:  connection received: host=[local]
2016-09-01 13:34:32 CEST LOG:  connection authorized: user=u1 database=b1
```

Le nom de l'utilisateur et la base de données sont tracés.

log_disconnections fait l'inverse :

```
2016-09-01 13:34:35 CEST LOG:  disconnection: session time: 0:01:04.634
                               user=u1 database=b1 host=[local]
```

Il ajoute une information importante : la durée de la session. Il est possible de récupérer cette information pour connaître la durée moyenne des sessions. Cette information est importante pour savoir si un outil de pooling de connexions a un intérêt.

log_autovacuum_min_duration correspond à log_min_duration_statement, mais pour l'autovacuum. Son but est de tracer l'activité de l'autovacuum si son exécution demande plus d'un certain temps.

log_checkpoints permet de tracer l'activité des checkpoints. Cela ajoute un message dans les traces pour indiquer qu'un checkpoint commence et une autre quand il termine. Cette deuxième trace est l'occasion d'ajouter des statistiques sur le travail du checkpoint :

```
2016-09-01 13:34:17 CEST LOG: checkpoint starting: xlog
2016-09-01 13:34:20 CEST LOG: checkpoint complete: wrote 13115 buffers (80.0%);
                              0 transaction log 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 paramètre log_lock_waits permet de tracer une attente trop importante de verrous. En fait, quand un verrou est en attente, un chronomètre est déclenché. Lorsque l'attente dépasse la durée indiquée par le paramètre deadlock_timeout, un message est enregistré, comme dans cet exemple :

```
2016-09-01 13:38:40 CEST LOG:  process 15976 still waiting for
```

```
                                AccessExclusiveLock on relation 26160 of
                                database 16384 after 1000.123 ms
2016-09-01 13:38:40 CEST STATEMENT:  DROP TABLE t1;
```

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur le serveur, ce qui peut diminuer fortement les performances.

Le paramètre `log_temp_files` permet de tracer toute création de fichiers temporaires, comme ici :

```
2016-09-01 13:41:11 CEST LOG:  temporary file: path
                               "base/pgsql_tmp/pgsql_tmp15617.1",
                               size 59645952
```

Tout fichier temporaire demande des écritures disques. Ces écritures peuvent poser problème pour les performances globales du système. Il est donc important de savoir si des fichiers temporaires sont créés ainsi que leur taille.

7.4.6 CONFIGURATION : DIVERS

- `log_line_prefix`
- `lc_messages`
- `log_timezone`

Lorsque la destination des traces est `syslog` ou `eventlog`, elles se voient automatiquement ajouter quelques informations dont un horodatage essentiel. Lorsque la destination est `stderr`, ce n'est pas le cas. Par défaut, l'utilisateur se retrouve avec des traces sans horodatage, autrement dit des traces inutilisables. PostgreSQL propose donc le paramètre `log_line_prefix` qui permet d'ajouter un préfixe à une trace. Ce préfixe peut contenir un grand nombre d'informations, comme un horodatage, le PID du processus serveur, le nom de l'application cliente, le nom de l'utilisateur, le nom de la base, etc.

Par défaut, les traces sont enregistrées dans la locale par défaut du serveur. Avoir des traces en français peut présenter certains intérêts pour les débutants mais cela présente plusieurs gros inconvénients. Chercher sur un moteur de recherche avec des traces en français donnera beaucoup moins de résultats qu'avec des traces en anglais. De même, les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il est vraiment préférable d'avoir les traces en anglais. Cela peut se faire ainsi :

```
lc_messages = 'C'
```

Quant à log_timezone, il permet de choisir le fuseau horaire pour l'horodatage des traces.

7.4.7 INFORMATIONS INTÉRESSANTES À RÉCUPÉRER

- Durée d'exécution
- Messages PANIC
- Rechargement de la configuration
- Fichiers temporaires

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 de niveau PANIC. Ces messages indiquent un état anormal du serveur qui s'est soldé par un arrêt brutal. Ce genre de problème doit être surveillé fréquemment.

7.4.8 DURÉE D'EXÉCUTION DES REQUÊTES

- log_statement et log_duration
- log_min_duration_statement

Exemple:

```
LOG:  duration: 112.615 ms  statement: select  * from t1 where c1=4;
```

- Outils : pgBadger, pgfouine, pgsi, etc.

PostgreSQL peut tracer les requêtes exécutées ainsi que leur durée d'exécution. Il existe deux moyens pour cela :

- le couple log_statement et log_duration ;
- ou log_min_duration_statement.

Le paramètre log_statement permet de tracer les requêtes, suivant leur type. Par exemple, il est possible de tracer uniquement les requêtes de modification de schéma (requêtes DDL).

Le paramètre log_duration permet de tracer la durée d'exécution des requêtes. Il s'agit d'un booléen : soit la trace est activée, soit elle ne l'est pas.

Les deux paramètres ne sont pas liés. Par contre, il est possible de tracer toutes les requêtes avec leur durée avec cette configuration :

17.12

```
log_statement = 'all'
log_duration = 'on'
```

Une telle configuration donnera deux lignes dans les traces :

```
2016-09-01 17:32:39 CEST LOG: statement:
                              insert into t1
                              values (2000000,'test');
2016-09-01 17:32:39 CEST LOG: duration: 167.138 ms
```

Pour la recherche d'optimisation de requêtes, il est préférable de passer par un autre paramètre. Ce dernier, appelé `log_min_duration_statement`, trace toute requête dont la durée d'exécution dépasse la valeur du paramètre (l'unité est la milliseconde). Il trace aussi la durée d'exécution des requêtes tracées. Par exemple, avec une valeur de 500, toute requête dont la durée d'exécution dépasse 500 ms sera tracée. À 0, toutes les requêtes se voient tracées. Pour désactiver la trace, il suffit de mettre la valeur -1 (qui est la valeur par défaut).

Suivant la charge que le système va subir à cause des traces, il est possible de configurer finement la durée à partir de laquelle une requête est tracée. Cependant, il faut bien comprendre que plus la durée est importante, plus la vision des performances est partielle. Il est parfois plus intéressant de mettre 0 ou une très petite valeur sur une petite durée, qu'une grosse valeur sur une grosse durée. 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 les plus longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

La trace fournie par `log_min_duration_statement` ressemble à ceci :

```
2016-09-01 17:34:03 CEST LOG:  duration: 136.811 ms  statement: insert into t1
                               values (2000000,'test');
```

7.4.9 MESSAGES PANIC

- Exemple:

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

- Envoi immédiat d'une alerte
- Outils : 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 indiqué ci-dessous :

```
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.

7.4.10 RECHARGEMENT DE LA CONFIGURATION

- Exemple :

```
LOG:  received SIGHUP, reloading configuration files
```

- Envoi d'une alerte pour s'assurer que cette configuration est voulue
- Outils : tail_n_mail

Il est intéressant de savoir quand la configuration du serveur change, 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. Une telle alerte vous permet de vérifier de la bonne configuration du serveur.

7.4.11 FICHIERS TEMPORAIRES

- Exemple :

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp9894.0",
      size 26927104
```

- Envoi d'une alerte sur un problème potentiel de performances

17.12

L'exécution d'une requête peut nécessiter la création de fichiers temporaires. 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. Toute écriture de ce type de fichiers impacte défavorablement l'exécution des requêtes. Être averti lors de la création de ce type de fichiers peut être intéressant. Cela étant dit, il est préférable de faire analyser après coup un fichier de traces pour savoir si un grand nombre de fichiers temporaires a été créé. Il est possible ainsi, avec un peu d'expérience, de voir que le nombre de fichiers augmente dans le mois, ce qui va demander une action de la part de l'administrateur de bases de données : vérifier les requêtes exécutées, vérifier la configuration, etc.

7.4.12 OUTILS

- Beaucoup d'outils existent
- Deux types :
  - en temps réel
  - rétro-analyse
- Nous verrons les plus connus/intéressants :
- 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).

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, et des outils spécifiques pour PostgreSQL comme tail_n_mail et logsaw.

L'analyse après coup permet une analyse plus fine, se terminant généralement par un rapport, fréquemment en HTML, parfois avec des graphes. Cette analyse plus fine nécessite des outils spécialisés. Là-aussi, il en existe plusieurs :

- pgBagder ;
- pgFouine ;
- pgsix ;
- pqa ;

- epqa ;
- etc.

Dans cette partie, nous allons voir les trois outils les plus intéressants.

7.4.13 UTILISER PGBADGER

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations
  - sur les requêtes 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 est un script Perl écrit par Gilles Darold. Il 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, il peut contenir des graphes zoomables.

C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL.

Le site web de pgBadger se trouve sur http://dalibo.github.com/pgbadger/

7.4.14 CONFIGURER POSTGRESQL POUR PGBADGER

- Configuration minimale
  - `log_destination`, `log_line_prefix` et `lc_messages`
- Configuration de base
  - `log_connections`, `log_disconnections`
  - `log_checkpoints`, `log_lock_waits`, `log_temp_files`
  - `log_autovacuum_min_duration`
- Configuration temporaire
  - `log_min_duration_statement`

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. Par contre, il faudra le préciser à pgBadger si la configuration est différente de celle qui suit :

```
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d '
```

La langue des traces doit être l'anglais. De toute façon, il s'agit de la meilleure configuration des traces. En effet, il est difficile de trouver de l'information sur des traces en français, alors que ce n'est pas le cas avec des traces en anglais. Pour cela, il suffit de configurer `lc_messages` à la valeur `C`.

Enfin, il faut demander à PostgreSQL de tracer les requêtes. Il est préférable de passer par `log_min_duration_statement` plutôt que `log_statement` et `log_duration` pour que pgBadger puisse faire l'association entre les requêtes et leur durée :

```
log_min_duration_statement = 0
log_statement = none
log_duration = off
```

Il est aussi possible 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.

Il est à noter que la version 9.0 essaie de récupérer les informations sur les adresses IP, utilisateurs connectés, bases de données à partir des traces sur les connexions (voir le paramètre `log_connections`) si jamais ces informations ne sont pas indiquées dans le préfixe (paramètre `log_line_prefix`).

La version 9.1 ajoute aussi un tableau sur les erreurs en utilisant le code SQLState si ce dernier est tracé (joker "%e" avec le paramètre `log_line_prefix`).

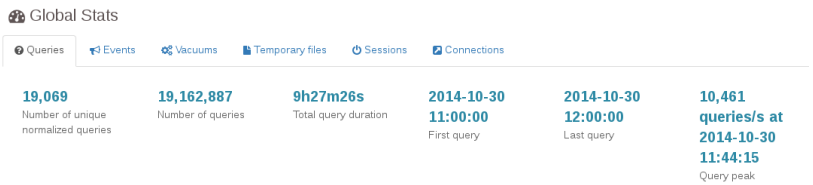
7.4.15 OPTIONS DE PGBADGER

- `--outfile`
- `--begin`, `--end`

- `--dbname`, `--dbuser`, `--dbclient`, `--appname`

Il existe énormément d'options. L'aide fournie sur le site web officiel les cite intégralement. Il serait difficile de les citer ici, des options étant ajoutées très fréquemment.

7.4.16 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, les requêtes :

```
SELECT * FROM utilisateurs WHERE id = 1;
```

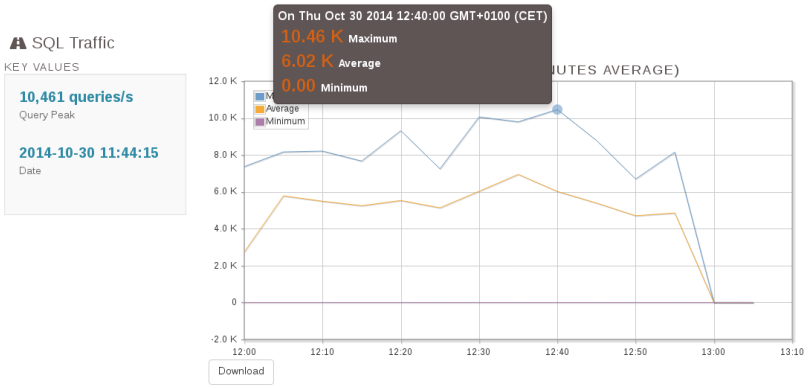
et

```
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 fiche 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 montré ci-dessus, pgBadger a comptabilisé environ 5,5 millions de requêtes, mais seulement 3941 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 4000.

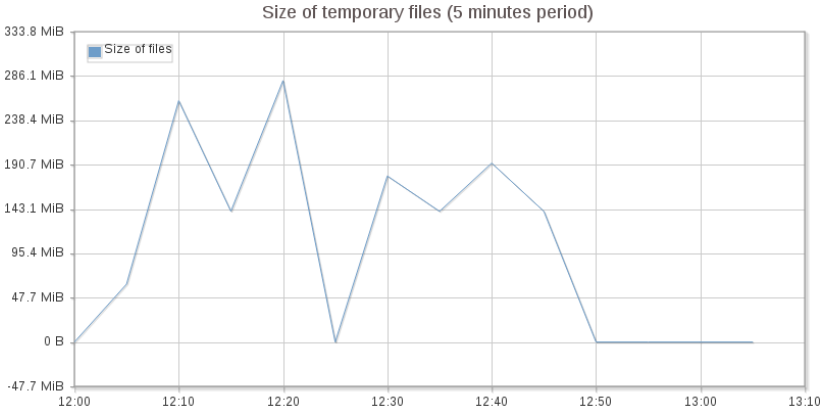
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 1 jour et 7 heures d'exécution de requêtes. Cependant, les traces vont du 20 au 23 mars, soit plus de trois jours. Cela indique que le serveur est assez peu sollicité. Il est plus fréquent que la durée d'exécution sérielle des requêtes soit 3 à 4 fois plus importants que la durée des traces.

7.4.17 PGBADGER : EXEMPLE 2



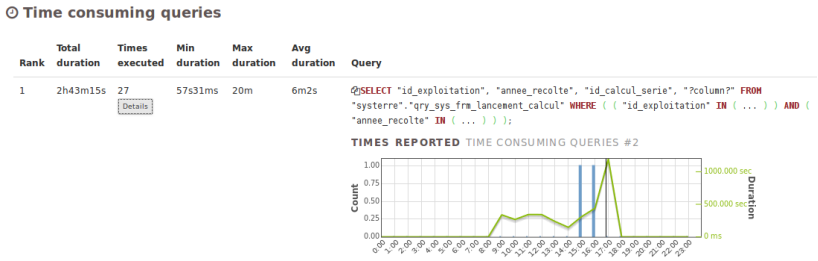
Ce graphe indique le nombre de requêtes par seconde. Le système ici est assez peu utilisé quand on considère le nombre moyen (en orange sur le graphe).

7.4.18 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. On remarque ainsi la création de fichiers pour un total de plus de 400 Mo par moment.

7.4.19 PGBADGER : EXEMPLE 4



Le plus important est certainement ce tableau. Il affiche 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. On remarque d'ailleurs dans cet exemple qu'avec les trois premières requêtes, on arrive à un total de 27 heures. Le premier exemple nous indique que l'exécution sérielle des requêtes aurait pris 23 heures. En ne travaillant que sur ces trois requêtes, nous travaillons en fait sur 87% du temps total d'exécution des requêtes comprises dans les traces. Pas besoin donc de travailler sur les 4000 requêtes normalisées.

7.4.20 UTILISER LOGWATCH

- Outil externe écrit en Perl
  - http://sourceforge.net/projects/logwatch/
  - Licence MIT
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs

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 impres-

sionnant et il est simple d'en ajouter de nouveaux. logwatch est le plus souvent intégré à votre distribution Linux.

Après son installation, vous pouvez voir qu'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 comme dans l'exemple qui suit.

7.4.21 CONFIGURER LOGWATCH

- Pas de configuration disponible par défaut pour PostgreSQL
- Dépôt git Dalibo
  - https://github.com/dalibo/pgsql_logwatch/
  - à installer manuellement

La configuration et le script relatifs au traitement des journaux applicatifs de PostgreSQL par logwatch ne sont pas disponibles en standard dans le paquet logqwatch. Vous devrez l'installer manuellement à partir du dépôt git disponible sur https://github.com/dalibo/pgsql_logwatch/ puis procéder aux actions suivantes :

```
sudo cp logfiles_postgresql.conf /etc/logwatch/conf/logfiles/postgresql.conf
sudo cp services_postgresql.conf /etc/logwatch/conf/services/postgresql.conf
sudo cp scripts_postgresql /etc/logwatch/scripts/services/postgresql
sudo chmod 755 /etc/logwatch/scripts/services/postgresql
```

Ces fichiers sont maintenant intégrés à la version 7.4.2 de Logwatch[72] .

7.4.22 OPTIONS DE LOGWATCH

- `--logfile`, les fichiers à traiter
- `--service`, filtre sur le service unique à traiter
- `--detail`, niveau de détails du rapport
- `--print`, pour afficher le rapport sur la sortie standard
- `--save`, pour sauvegarder le rapport dans un fichier
- `--mailto`, pour envoyer le rapport par mail

logwatch dispose d'autres options, la page man est certainement la meilleure documentation à l'heure actuelle.

[72] http://sourceforge.net/projects/logwatch/

7.4.23 LOGWATCH : EXEMPLE 1

```
/usr/sbin/logwatch --detail Med --service postgresql --range All
```

La valeur `All` pour le paramètre `--range` indique que l'on veut un rapport sur tous les fichiers de traces existants. Pour n'avoir de rapport que sur la journée, la valeur devrait être `Today`. La valeur par défaut est `Yesterday` .

On peut changer aussi le niveau de détail, avec `--detail` à `Low`. Seuls les messages de type `FATAL`, `PANIC` et `ERROR` seront remontés. Avec la valeur `Med`, les messages de type `WARNING` et `HINTS` sont ajoutés aux précédents.

7.4.24 LOGWATCH : EXEMPLE 2

```
/usr/sbin/logwatch --detail Med --service postgresql \
                   --range Yesterday --output mail \
                   --mailto admin@mydom.com --format html
```

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

 6 times:
 [2011-11-01 06:38:20 +/-23 hour(s)]  password authentication failed for
                                      user "test"

 4 times:
 [2011-10-24 17:26:24 +/-14 seconds]  Ident authentication failed for
                                      user "test"
 [2011-11-24 23:22:02 +/-1 day(s)]  database system is shut down

 3 times:
 [2011-12-10 02:49:02 +/-23 day(s)]  database "test" does not exist

 1 times:
 [2011-12-09 17:39:06]  role "test" is not permitted to log in

 Errors:
 -------

 7 times:
 [2011-11-14 12:20:44 +/-58 minute(s)]  relation "COUNTRIES" does not exist

 6 times:
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "dept_loc_fk" of relation
                                        "departments" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "loc_c_id_fk" of relation
                                        "locations" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "dept_mgr_fk" of relation
                                        "departments" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "afpr_aft_fk" of relation
                                        "flx_properties" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "jhist_job_fk" of relation
                                        "job_history" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_manager_fk" of relation
                                        "employees" does not exist
 [2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_dept_fk" of relation
                                        "employees" does not exist
 [2011-11-14 12:19:42 +/-56 minute(s)]  syntax error at or near "dbms_lob"
```

```
[2011-11-14 12:19:43 +/-56 minute(s)]  constraint "emp_job_fk" of relation
                                       "employees" does not exist
[2011-11-14 12:19:42 +/-56 minute(s)]  syntax error at or near "NAME"

5 times:
[2011-11-14 12:21:24 +/-59 minute(s)]  syntax error at or near
                                       "role_permission_view"

4 times:
[2011-12-08 20:46:37 +/-16 hour(s)]  conflicting or redundant options
[2011-11-01 19:10:33 +/-6 minute(s)]  relation "t_test" does not exist

2 times:
[2011-11-14 12:30:51 +/-1 minute(s)]  schema "hr" already exists
[2011-11-14 14:32:15 +/-4 hour(s)]  syntax error at or near "hr"

Warnings:
---------

55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  terminating connection because of
                                   crash of another server process

1 times:
[2011-11-29 09:05:47]  pgstat wait timeout
[2011-11-14 16:34:03]  nonstandard use of \' in a string literal

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.
```

```
 2 times:
 [2011-10-10 17:38:28 +/-3 minute(s)]  No operator matches the given name
                                       and argument type(s). You might need
                                       to add explicit type casts.
 [2011-10-25 09:18:52 +/-2 seconds]  Consider increasing the configuration
                                     parameter "checkpoint_segments".

 1 times:
 [2011-11-15 15:20:50]  Could not choose a best candidate function. You
                        might need to add explicit type casts.
 [2011-11-14 16:34:03]  Use '' to write quotes in strings, or use the
                        escape string syntax (E'...').


 ---------------------- PostgreSQL End ------------------------


 ####################### Logwatch End #########################
```

7.4.25 UTILISER 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.

7.4.26 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
```

La clé MAILSUBJECT précise le sujet du mail envoyé. La clé EMAIL permet de préciser à quelle adresse mail envoyer les rapports.

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.

Enfin, la clé FILE permet de préciser le fichier à analyser.

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.

7.4.27 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
```

7.5 STATISTIQUES

- 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.

7.5.1 STATISTIQUES - CONFIGURATION 1

- Tracer l'activité
  - `track_activities`
- S'assurer que les requêtes ne sont pas tronquées
  - `track_activity_query_size`

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`. Si ce dernier est désactivé, la liste des sessions dans la vue `pg_stat_activity` est toujours présente et actualisée, mais les colonnes `xact_start`, `query_start`, `state_change`, `wait_event_type`, `wait_event`, `state` et `query` ne sont plus mises à jour. Une fois le paramètre activé, les colonnes sont renseignées.

Les colonnes `wait_event_type` et `wait_event` remplacent la colonne `waiting` à partir de la version 9.6.

```
b1=# SHOW track_activities;
 track_activities
------------------
 off
(1 row)

b1=# \x
Expanded display is on.
postgres=# select * from pg_stat_activity;
-[ RECORD 1 ]----+------------------------------
datid            | 13356
datname          | postgres
pid              | 15256
usesysid         | 67397
usename          | u1
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2016-09-01 13:34:32.09651+02
xact_start       |
query_start      |
state_change     |
wait_event_type  |
wait_event       |
state            | disabled
backend_xid      |
backend_xmin     | 3432
query            |
backend_type     | backend client
```

Il est à noter que la requête indiquée dans la colonne `query` est tronquée à 1024 caractères par défaut. Il est possible de configurer cette limite grâce au paramètre `track_activity_query_size`.

Voici un exemple montrant la requête tronquée après 100 caractères (valeur du paramètre `track_activity_query_size`) :

```
b1=# \x
Expanded display is on.
b1=# SELECT current_setting('track_activities'),
```

```
current_setting('track_activity_query_size');
-[ RECORD 1 ]---+----
current_setting | on
current_setting | 100

b1=# SELECT '0123456789', query FROM pg_stat_activity;
-[ RECORD 1 ]----------------------------------------------
?column? | 0123456789
query    | SELECT '0123456789', query FROM pg_stat_activity;

b1=# SELECT '012345678901234567890123456789012345678901234567890123456789'
            '012345678901234567890123456789012345678901234567890123456789ABCDEF',
            query FROM pg_stat_activity;
-[ RECORD 1 ]----------------------------------------------
?column? | 012345678901234567890123456789[...]123456789ABCDEF
query    | SELECT '012345678901234567890123456789[...]1234567890
```

7.5.2 STATISTIQUES - CONFIGURATION 2

- `track_counts`
- `track_io_timing`
- `track_functions`

Le collecteur de statistiques est capable de récupérer des informations sur des nombres de lignes (lues, insérées, mises à jour, supprimées, vivantes, mortes, etc) et sur des nombres de blocs (lus dans le cache de PostgreSQL et en dehors de ce cache). Pour avoir ce type de statistiques, il faut activer le paramètre `track_counts`.

En version 9.2, PostgreSQL peut chronométrer les opérations sur les disques : lectures, écritures, synchronisations. Pour cela, il faut activer le paramètre `track_io_timing`.

Depuis la version 9.0, PostgreSQL sait aussi récupérer des statistiques sur les procédures stockées. Il faut modifier la valeur du paramètre `track_functions` qui, contrairement aux autres paramètres, se trouve être un enum à 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.

7.5.3 STATISTIQUES - CONFIGURATION 3

- `stats_temp_directory`
  - fichier écrit toutes les 500 ms avant la 8.4
  - seulement quand nécessaire à partir de la 8.4

Le collecteur de statistiques fonctionne ainsi :

- il est lancé au démarrage de PostgreSQL (il est d'ailleurs impossible de le désactiver complètement) ;
- il copie le fichier `global/pgstat.stat` dans le répertoire ciblé par le paramètre `stats_temp_directory` ;
- il modifie ce fichier dès que les autres processus lui fournissent des statistiques ;
- à l'arrêt de PostgreSQL, il copie le fichier `pgstat.stat` du répertoire ciblé par le paramètre `stats_temp_directory` dans le répertoire `global`.

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.

7.5.4 INFORMATIONS INTÉRESSANTES À RÉCUPÉRER

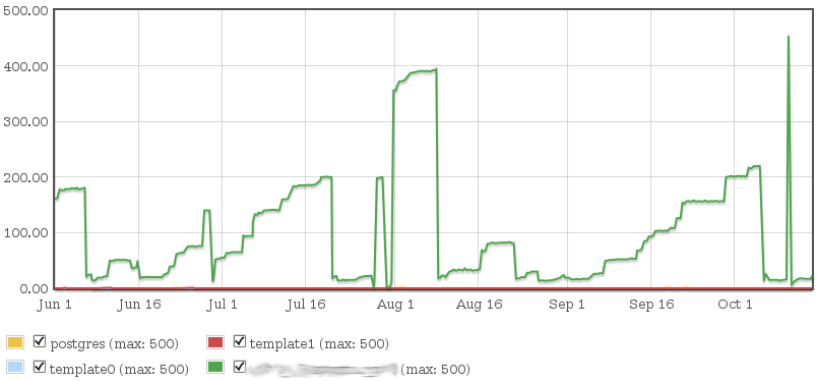
- Sur l'activité
- sur l'instance
- sur les bases
- sur les tables
- sur les index
- sur les fonctions

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 a peu d'intérêt car seulement certaines sont vraiment intéressantes à superviser. Nous allons voir ici quelques exemples.

7.5.5 NOMBRE DE CONNEXIONS PAR BASE

```
SELECT datname, numbackends FROM pg_stat_database GROUP BY 1;
SELECT datname, count(*) FROM pg_stat_activity WHERE datname IS NOT NULL GROUP BY 1;
```



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, numbackends FROM pg_stat_database WHERE usename IS NOT NULL GROUP BY 1;
```

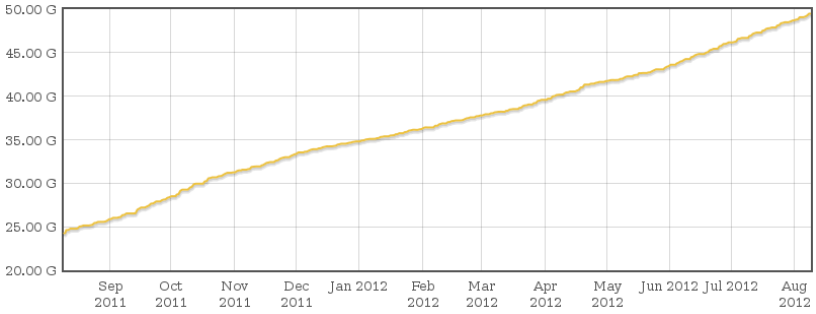
- application cliente :

```
SELECT application_name, numbackends FROM pg_stat_database GROUP BY 1;
```

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.

7.5.6 TAILLE DES BASES

```
SELECT datname, pg_database_size(oid) FROM pg_database;
```

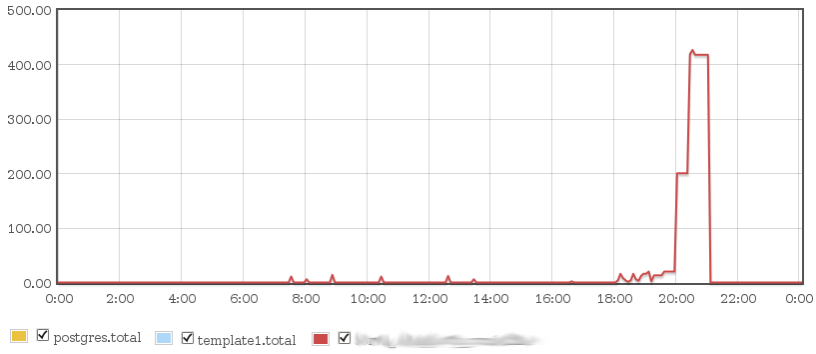


La volumétrie des bases est une autre information fréquemment demandée. L'exécution de cette requête toutes les cinq minutes permettra de suivre l' évolution de la taille des bases.

Le graphe ci-dessus montre une montée en volumétrie assez importante, la base ayant doublé en un an.

7.5.7 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;
```



17.12

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.

7.5.8 ET UN GRAND NOMBRE D'AUTRES INFORMATIONS

- Ratio de lecture du cache (souvent appelé `hitratio`)
- 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.

7.5.9 OUTILS

- Beaucoup d'outils existent pour exploiter les statistiques
- Les plus connus/intéressants sont :
  - Munin
  - Nagios
  - Zabbix
  - OPM
  - pg_stat_statements
  - PoWA

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, généralement contenues dans un script de supervision. Le plus utilisé actuellement est `check_postgres` développé par End Point Corporation, auteurs entre autres de Bucardo. Le script `check_pgactivity`, développé par Dalibo, vise à le remplacer.

OPM (*Open PostgreSQL Monitoring*) est un outil lancé par Dalibo, qui repose sur la mise en place d'un système de collecte, stocke la plupart des statistiques de PostgreSQL et du système d'exploitation sous-jacent dans une base de données PostgreSQL, et permet de les exploiter sous forme de graphiques interactifs et personnalisables .

pg_stat_statements est un module contrib de PostgreSQL, il s'agit donc d'une fonctionnalité qui n'est pas installée par défaut. Il ajoute la collecte de statistiques sur toutes les requêtes exécutées.

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, 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.

7.5.10 OUTILS - MUNIN

- Scripts Perl
- Sondes PostgreSQL incluses depuis la 1.4
- Récupère les statistiques toutes les cinq minutes
- 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.

Dalibo avait créé un projet de sondes PostgreSQL pour la version 1.2 de munin. Ces sondes ont été réécrites par les développeurs de munin et directement intégrées dans la version 1.4 officielle.

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 créés pour faciliter l'accès aux graphes.

7.5.11 OUTILS - NAGIOS

- Outil GPL, sur http://www.nagios.org/
- 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 est possible de le coupler à des sondes dédiées, comme la sonde Bucardo `check_postgres` ou la plus récente sonde Dalibo `check_pgactivity`, pour qu'il puisse récupérer un certain nombre d'informations statistiques de PostgreSQL.

7.5.12 OUTILS - ZABBIX

- Outil GPL, sur http://www.zabbix.com/
- 2.0
- Sonde `check_postgres.pl`
- Template pg-monz
  - http://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.

7.5.13 OUTILS - OPM

- *Open PostgreSQL Monitoring*
- Suite de supervision lancée par Dalibo en septembre 2014
- Projet indépendant mené par OPMDG (*OPM Development Group*)
- Licence PostgreSQL

- http://opm.io/

Open PostgreSQL Monitoring est une suite de supervision lancée par Dalibo. L'objectif est d'avoir un outil permettant de surveiller un grand nombre d'instances PostgreSQL, et à terme de les administrer, de façon similaire à ce que permettent les outils *Oracle Enterprise Manager* ou *SQL Server Management Studio*.

Le projet est publié sous licence PostgreSQL, et un comité a été créé (OPMDG, *OPM Development Group*), à la manière du PGDG, de façon à assurer l 'indépendance du projet.

Le cœur de OPM est une base de données PostgreSQL stockant de nombreuses statistiques concernant les instances et le système d'exploitation des serveurs les hébergeant. Ces données sont ensuite utilisées par l'interface graphique pour les afficher sous forme de graphiques interactifs et personnalisables.

À ce jour la collecte des statistiques nécessite la configuration de Nagios avec le script `check_pgactivity`, mais d'autres systèmes de collecte pourront être ajoutés à l'avenir.

7.5.14 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 un module contrib de PostgreSQL, il s'agit donc d'une fonctionnalité qui n'est pas 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 : http://www.postgresql.org/docs/9.3/static/pgstatstatements.html

---

7.5.15 OUTILS - POWA

- *PostgreSQL Workload Analyzer*
- Effectue des captures des statistiques collectées par pg_stat_statements
- Fournit une interface graphique pour observer en temps réel l'activité des requêtes
- Licence PostgreSQL
- https://github.com/dalibo/powa-archivist
- https://github.com/dalibo/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, sous licence PostgreSQL. Tout comme pour 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. 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.

PoWA 1 et 2 sont compatibles avec les versions 9.3 et supérieures de PostgreSQL.

PoWA 3 est compatible avec les versions 9.4 et supérieures de PostgreSQL.

7.6 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.