Formation SQL4

PostgreSQL : L'état de l'art

DALIBO
L'expertise PostgreSQL

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL : L'état de l'art

Formation SQL4

TITRE : PostgreSQL : L'état de l'art
SOUS-TITRE : Formation SQL4

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-08-1

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

Licence Creative Commons BY-NC-SA 2.0 FR **5**

1 Licence Creative Commons CC-BY-NC-SA **10**

2 Découvrir PostgreSQL **11**
2.1 Préambule . . . 11
2.2 Un peu d'histoire... . . . 13
2.3 Les versions . . . 18
2.4 Concepts de base . . . 29
2.5 Fonctionnalités . . . 35
2.6 Sponsors & Références . . . 48
2.7 Conclusion . . . 51

3 Richesses de l'écosystème PostgreSQL **53**
3.1 Préambule . . . 53
3.2 Les projets satellites . . . 54
3.3 Comparatifs . . . 65
3.4 À la rencontre de la communauté . . . 71
3.5 Conclusion . . . 79

4 Outils graphiques et console **80**
4.1 Préambule . . . 80
4.2 Outils console de PostgreSQL . . . 81
4.3 La console psql . . . 86
4.4 Écriture de scripts shell . . . 105
4.5 Outils graphiques . . . 120
4.6 Conclusion . . . 136
4.7 Travaux Pratiques . . . 137

5 Architectures de Haute-Disponibilité **140**
5.1 Préambule . . . 140
5.2 Rappels théoriques . . . 142
5.3 Réplication interne physique . . . 146
5.4 Réplication interne logique . . . 152
5.5 Réplication externe . . . 154
5.6 Réplication bas niveau . . . 169
5.7 Conclusion . . . 170

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.

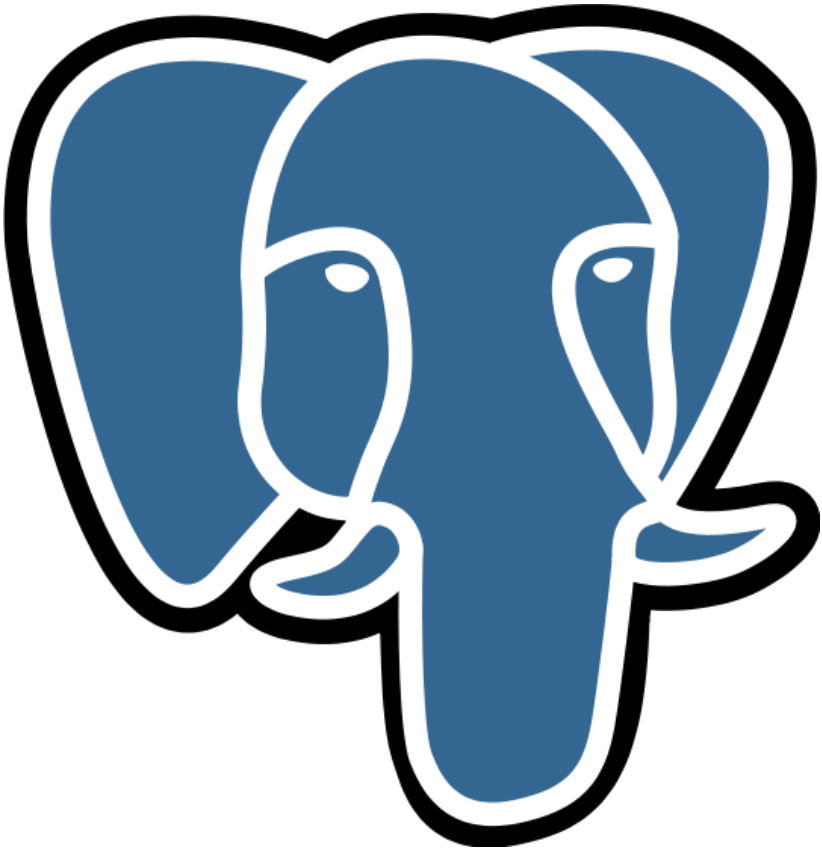
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

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

17.12

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.

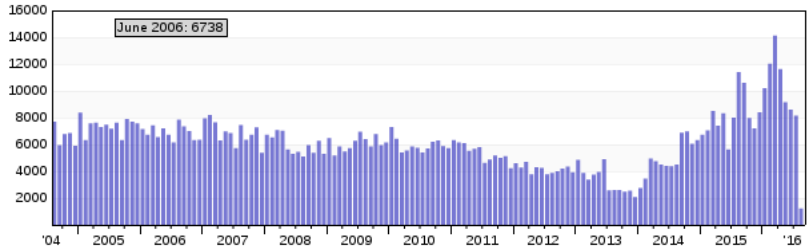
En 2008, douze ans après la création du projet, des associations d' utilisateurs apparaissent pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. PostgreSQL UK organise une journée de conférences à Londres, PostgreSQL Fr en organise une à Toulouse. Des « sur-groupes » apparaissent 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

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.

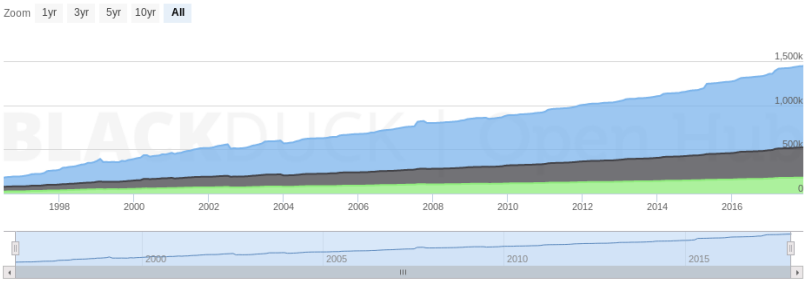
Pour aller plus loin : On peut également visualiser l'évolution des contributions de la communauté PostgreSQL grâce au projet Code Swarm[11] .

---

[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

17.12

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

- 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/

17.12

  - 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/

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

17.12

- 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

- 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

17.12

- 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

- 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

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.

17.12

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.

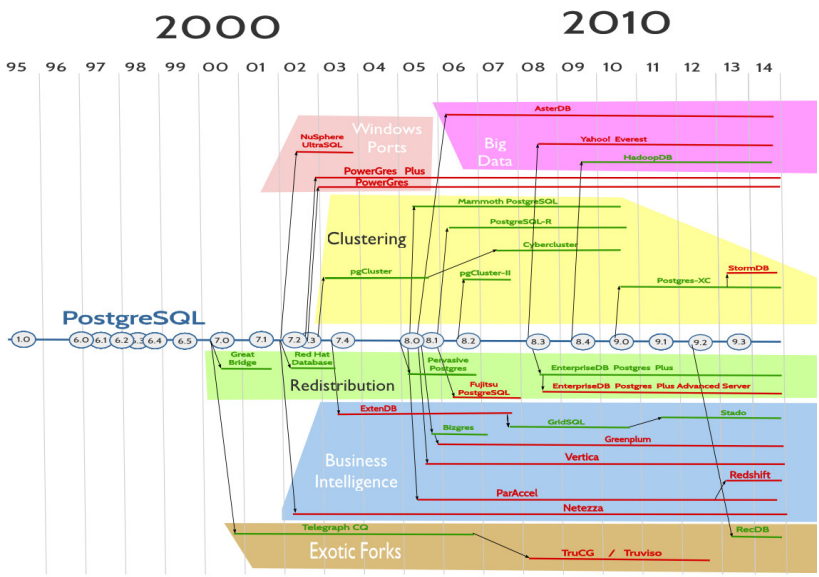
Liste exhaustive des « forks »[38] .

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

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

17.12

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

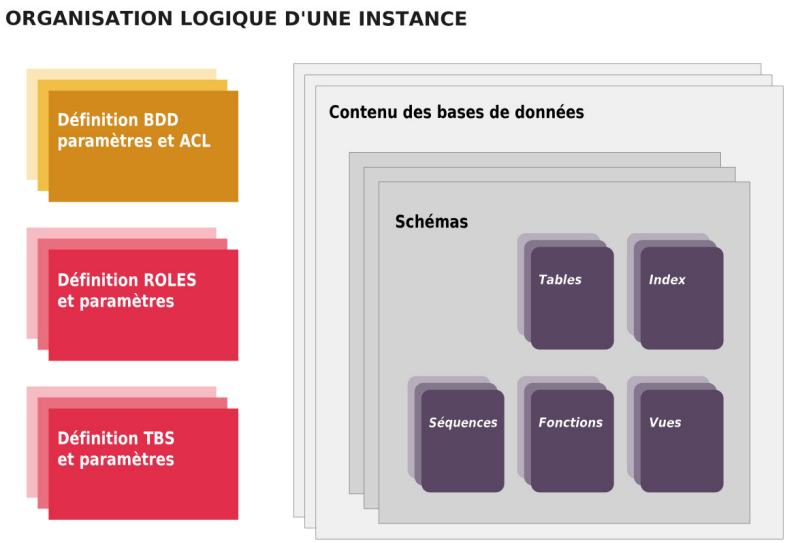
17.12

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

- 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

17.12

```
=> 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

17.12

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

- 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/

17.12

« 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 RICHESSES DE L'ÉCOSYSTÈME POSTGRESQL

3.1 PRÉAMBULE

- Projet horizontal & décentralisé
- La « biodiversité » est une force
- Le meilleur SGBD du marché ?

La communauté est structurée de manière décentralisée et ouverte. Tout le monde peut participer, fournir des patchs, exprimer son opinion. Évidemment, tous les patchs ne seront pas forcément intégrés. Toutes les opinions n'ont pas forcément la même valeur. Le projet est basé sur la méritocratie. Chaque contributeur a un potentiel confiance dépendant de ses précédentes contributions.

Le fait qu'un grand nombre de personnes peut participer fait que des opinions diverses et variées sont prises en compte dans l'évolution de PostgreSQL. Cela ne peut être qu'une bonne chose, même si parfois les discussions sont longues et les décisions moins simples

17.12

à prendre.

3.1.1 AU MENU

- Projets satellites
- Comparatifs
- Communauté
- Avenir

Après un tour d'horizon des logiciels gravitant autour de PostgreSQL et un comparatif avec les autres SGBD dominants, nous verrons le fonctionnement de la communauté et l'avenir du projet.

3.1.2 OBJECTIFS

- Connaître les logiciels connexes
- Exploiter toute la puissance du SGBD
- Participer !

Ce module est destiné aux utilisateurs qui souhaitent repousser les limites d'une utilisation classique.

À l'issue de ce chapitre, vous aurez une vision claire des projets complémentaires qui vous simplifieront la gestion quotidienne de vos bases. Vous connaîtrez les différences entre PostgreSQL et ses concurrents. Enfin, vous serez en mesure de contribuer au projet, si le cœur vous en dit !

3.2 LES PROJETS SATELLITES

- Administration
- Supervision et monitoring
- Migrations
- SIG
- Modélisation

PostgreSQL n'est qu'un moteur de bases de données. Quand vous l'installez, vous n'avez que ce moteur. Vous disposez de quelques outils en ligne de commande (détaillés dans

nos modules « Outils graphiques et consoles » et « Tâches courantes ») mais aucun outil graphique n'est fourni.

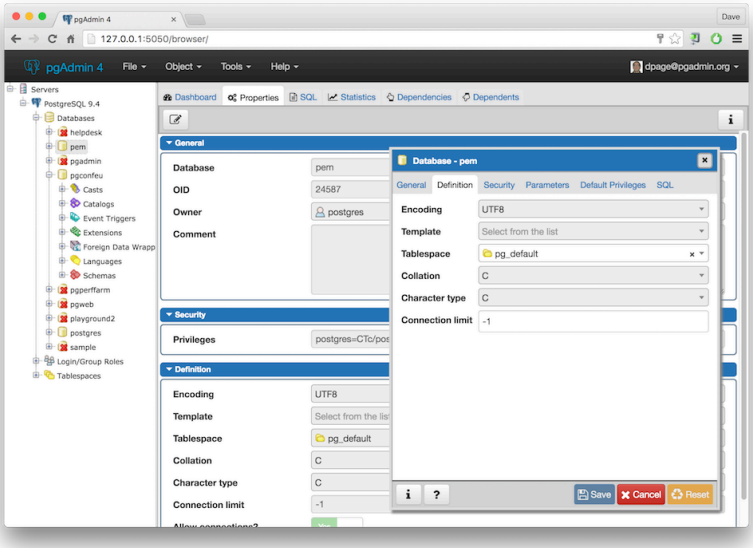
Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

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

3.2.1 PGADMIN IV

- Site officiel : http://www.pgadmin.org/
- Version : 2.0
- Licence : PostgreSQL
- Gestion graphique de l'administration des bases
- Éditeur de requêtes
- Supervision

Logiciel libre d'administration de la base de données PostgreSQL, pgAdmin comprend une interface graphique d'administration, un outil de requêtage SQL, un éditeur de code procédural, un débogueur PL/psql, un éditeur des fichiers de configuration, une fenêtre de statut du serveur et bien plus encore.



pgAdmin est conçu pour répondre à la plupart des besoins, depuis l'écriture de simples requêtes SQL jusqu'au développement de bases de données complexes. L'interface graphique supporte les fonctionnalités de PostgreSQL les plus récentes et facilite l'administration.

Il supporte toutes les versions maintenues de PostgreSQL (il peut supporter des versions qui ne sont plus maintenues, mais ceci dépend fortement de la version de pgAdmin et du bon vouloir des développeurs), ainsi que les versions commerciales de PostgreSQL comme EnterpriseDB et Greenplum.

Il est disponible dans plusieurs langues et est utilisable sur différents systèmes d'exploitation.

Ce logiciel existe actuellement en deux versions :

- pgAdmin III ;
- pgAdmin IV.

Les développeurs de pgAdmin III ont abandonné cette version pour plusieurs raisons :

- le socle technique sur lequel elle reposait n'avançait pas avec son temps ;
- il y avait peu de développeurs connaissant ce socle et prêts à travailler dessus.

Dave Page, leader du projet, a donc décidé de ré-écrire ce projet dans un langage plus connu, plus apprécié, et où il serait possible de trouver plus de développeurs. Il a fini

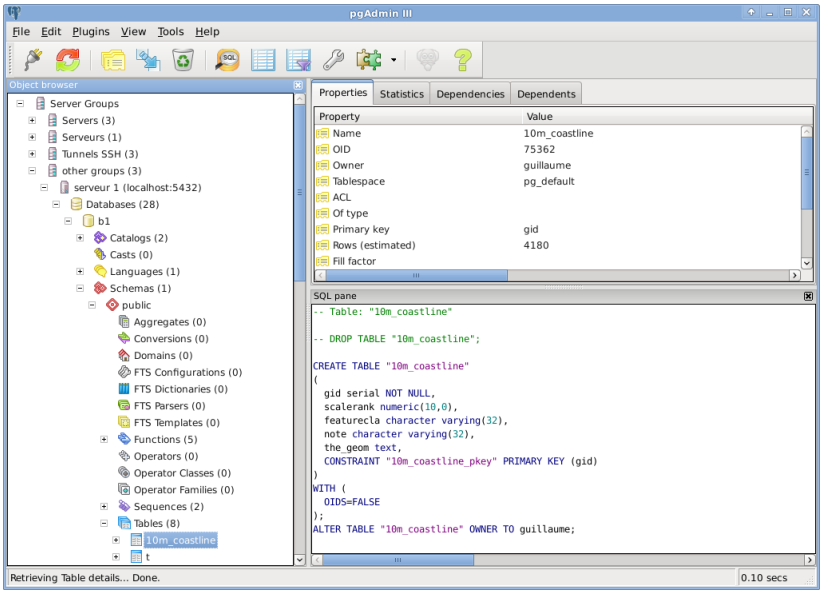
par sélectionner le langage Python et a réécrit, avec son équipe, une grande partie de pgAdmin III sous le nom de pgAdmin IV.

Les versions 1.x de pgAdmin IV ont beaucoup été décriées, par leur manque de fonctionnalités et par leur lenteur. Un grand nombre de ces plaintes ont été entendues et des corrections effectuées. Le résultat est une version 2.0 qui semble tenir ses promesses jusqu'à maintenant.

3.2.2 PGADMIN III

- Site officieux : https://www.openscg.com/bigsql/pgadmin3/
- Version : 1.23
- Licence : PostgreSQL
- Gestion graphique de l'administration des bases
- Éditeur de requêtes / `EXPLAIN` graphique
- Gestion de Slony

pgAdmin III est codé en C++ et utilise la bibliothèque wxWidgets pour être utilisable sur différents systèmes d'exploitation, et donc différents systèmes graphiques.



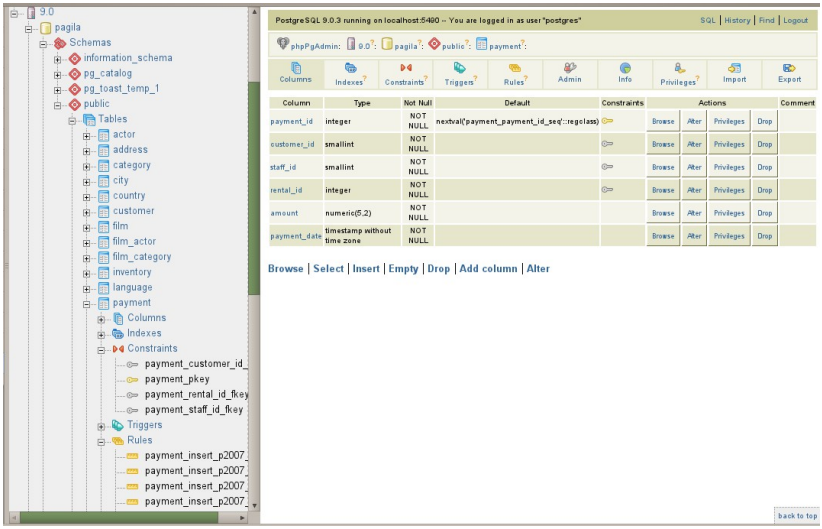
Cette version n'est plus maintenue par la communauté.

La société OpenSCG propose sa propre version, qu'elle appelle pgAdmin 3 LTS. D'après les commits visibles dans le dépôt officiel (https://bitbucket.org/openscg/pgadmin3-lts/commits/all), la seule amélioration est le support de la version 10. Par support, ils entendent que le test vérifiant le numéro de version accepte aussi la version 10. Il n'y a donc aucun support des nouvelles fonctionnalités de la version 10, aucune modification pour améliorer la stabilité, etc. Cette version de pgAdmin a vraisemblablement peu d'avenir devant elle.

3.2.3 PHPPGADMIN

- Site officiel : http://phppgadmin.sourceforge.net/
- Version : 5.1
- Licence : GPL
- Gestion graphique de l'administration des bases
- Interface web
- Mais ne semble plus maintenu
  - pas de nouvelles versions depuis avril 2013
  - pas de commit depuis avril 2016

- Utiliser plutôt pgAdmin IV sur un serveur web



PhpPgAdmin est une application web déportée, multilingue, simple d'emploi permettant la maintenance, la création, l'import-export de bases de données, la gestion des utilisateurs, ainsi que l'exécution de SQL et de scripts.

Cependant, il n'est plus maintenu. La dernière version, 5.1, date d'avril 2013, le dernier commit a été réalisé en avril 2016. Autrement dit, il y a un gros risque qu'il ne soit pas compatible avec PostgreSQL 10 et les versions suivantes, et il n'en supportera de toute façon pas les nouvelles fonctionnalités.

Notre conseil est d'utiliser plutôt pgAdmin IV sur un serveur web. La documentation de pgAdmin indique comment configurer un serveur web Apache.

3.2.4 PGBADGER

- Site officiel : http://projects.dalibo.org/pgbadger/
- Version : 9.2
- Licence : PostgreSQL
- Analyse des traces de durée d'exécution des requêtes
- Analyse des traces du `VACUUM`, des connexions, des checkpoints
- Compatible syslog, stderr, csvlog

pgBadger est un analyseur des journaux applicatifs de PostgreSQL. Il permet de créer

des rapports détaillés depuis ceux-ci. pgBadger est utilisé pour déterminer les requêtes à améliorer en priorité pour accélérer son application basée sur PostgreSQL.

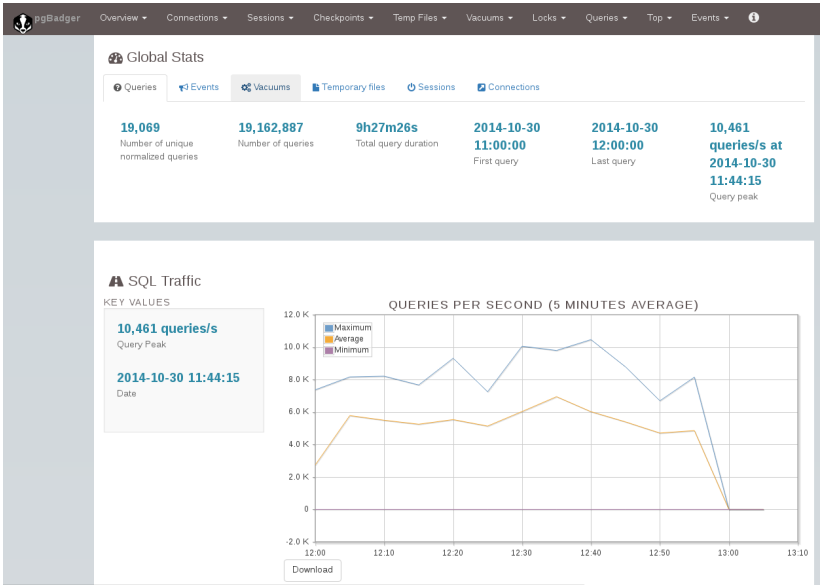


FIGURE 1: CAPTURE PGBADGER

pgBadger est écrit en Perl et est facilement extensible si vous avez besoin de rapports spécifiques.

pgBadger est conçu pour traiter de gros fichiers de logs avec une mémoire réduite.

Pour plus de détails : L'un des principaux développeurs de pgBadger est Gilles Darold, membre de l'équipe dalibo[58] . Le développement de l'outil se poursuit sur github[59] .

---

[58] http://www.dalibo.com
[59] https://github.com/dalibo/pgbadger/

3.2.5 OPM

- *Open PostgreSQL Monitoring*
- Site officiel : http://opm.io/
- Version : 2.4
- Licence : PostgreSQL
- Suite de supervision lancée par Dalibo en septembre 2014
- Projet indépendant mené par OPMDG (*OPM Development Group*)

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 d'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.

3.2.6 POWA

- Site officiel : http://dalibo.github.io/powa/
- Version : 3.1
- Licence : PostgreSQL
- Surveillance de l'activité SQL
- Développé par Dalibo
- Dépôt GitHub
  - archiveur : https://github.com/dalibo/powa-archivist
  - UI web : https://github.com/dalibo/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, sous licence PostgreSQL.

Tout comme pour l'extension standard pg_stat_statements, sa mise en place nécessite la modification du paramètre `shared_preload_libraries`, et donc le redémarrage de l'instance. Il faut également créer une nouvelle base de données dans l'instance. Par ailleurs, PoWA repose sur les statistiques collectées par pg_stat_statements, celui-ci doit donc être également installé.

Une fois installé et configuré, l'outil va récupérer à intervalle régulier les statistiques collectées par pg_stat_statements, les stocker et les historiser.

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 est compatible avec les versions 9.4 et supérieures de PostgreSQL. Pour utiliser PoWA avec PostgreSQL 9.3, il faut utiliser la version 1.2.

3.2.7 ORA2PG

- Site officiel : http://ora2pg.darold.net/
- Version : 18.2
- Licence : GPL
- Migration de la structure d'une base Oracle...
- ... des procédures stockées et des données

Ora2pg est un outil facilitant la migration d'une base Oracle vers une base PostgreSQL. Il s'occupe du schéma, des procédures stockées comme des données, suivant un fichier de configuration.

Pour plus de détails : Le développeur d'ora2pg est Gilles Darold, membre de l'équipe dalibo[60] . Le support s'effectue à travers github[61] .

3.2.8 SQLSERVER2PG

- Site officiel : http://dalibo.github.io/sqlserver2pgsql/
- Version : sans
- Licence : GPL v3
- Convertit un schéma SQL Server en un schéma PostgreSQL
- Produit en option un job Pentaho Data Integrator (Kettle) pour migrer toutes les données de SQL Server vers PostgreSQL

sqlserver2pgsql est un script Perl facilitant la migration d'une base SQL Server vers une base PostgreSQL.

3.2.9 DB2TOPG

- Site officiel : https://github.com/dalibo/db2topg
- Version : sans
- Licence : GPL v3
- Convertit une base DB2 en une base PostgreSQL

sqlserver2pgsql est un ensemble de scripts Perl facilitant la migration d'une base DB2 vers une base PostgreSQL.

[60] http://www.dalibo.com
[61] https://github.com/darold/ora2pg

17.12

Il faut partir d'une sauvegarde SQL de la base DB2. Le script db2topg le convertit en un schéma PostgreSQL. Vous pouvez demander en plus un script pour sauvegarder toutes les données de cette base DB2. Pour migrer les données, il suffit d'exécuter le script produit. Il récupère les fichiers CSV de DB2 qu'il faut ensuite importer dans PostgreSQL (le script deltocopy.pl aide à le faire.

3.2.10 POSTGIS

- Site officiel : http://postgis.net/
- Version : 2.4
- Licence : BSD
- Module spatial pour PostgreSQL
- Conforme aux spécifications de l'OpenGIS Consortium
- Compatible avec MapServer

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

PostGIS a été développé par la société Refractions Research comme une technologie Open-Source de base de données spatiale. Cette société continue à développer PostGIS, soutenue par une communauté active de contributeurs.

La version 2.0 apporte de nombreuses nouveautés attendues par les utilisateurs comme le support des fonctionnalités *raster* et les surfaces tri- dimensionnelles.

3.2.11 PGMODELER

- Site officiel : http://pgmodeler.com.br/
- Version : 0.9
- Licence : GPLv3
- Modélisation de base de données
- Fonctionnalité d'import export

pgmodeler permet de modéliser une base de données. Son intérêt par rapport à d'autres produits concurrents est qu'il est spécialisé pour PostgreSQL. Il en supporte donc toutes les spécificités, comme l'héritage de tables, les types composites, les types tableaux... C'est une excellente solution pour modéliser une base en partant de zéro, ou pour extraire une visualisation graphique d'une base existante.

Il est à noter que, bien que le projet soit open-source, son installation par les sources peut être laborieuse. L'équipe de développement propose des paquets binaires à prix modique.

3.3 COMPARATIFS

- Pas de SGBD universel
- S'inspirer des concurrents plutôt que de les combattre
- PostgreSQL vs.
  - MySQL
  - SQL Server

  - Oracle
  - Informix
  - NoSQL
- Attention aux benchmarks !

Il est toujours difficile de comparer différents SGBD, car ce sont des logiciels complexes et orientés différement.

Avant toute chose, il parait donc nécessaire de rappeler quelques points :

- **Il n'y a pas de SGBD universel !** Aucun logiciel de base de données ne peut couvrir parfaitement l'ensemble des besoins en matière de stockage d'information. On peut reprendre l'image suivante : si l'on pose la question « Quel est le véhicule le plus rapide ? une Ferrari ou un camion de 3,5 tonnes ? », la réponse dépend du contexte ! Si vous devez transporter une personne, alors la Ferrari est le meilleur choix. Si vous devez transporter une tonne de marchandises, alors vous opterez pour le camion !
- **Pour être réellement utiles, les benchmarks doivent être conduits en conditions réelles**. Si vous souhaitez tester différents SGBD, assurez- vous que les tests correspondent à votre cas d'utilisation et à votre matériel.
- **PostgreSQL s'inspire des projets concurrents.** Les autres SGBD sont vus comme des compétiteurs plutôt que comme des ennemis à abattre. La réciproque n'est pas toujours vraie !

3.3.1 POSTGRESQL VS. MYSQL

- Différents moteurs
  - MyISAM ou InnoDB ?
- Points forts de PostgreSQL
  - DDL transactionnel
- Points faibles de PostgreSQL
  - lenteur du `SELECT count(*)` (amélioration en 9.2)
- Quel avenir pour MySQL ?

Pendant des années, MySQL était perçu comme plus rapide et plus simple à utiliser que PostgreSQL. PostgreSQL était vu comme puissant, stable et respectueux des standards mais également lent et compliqué. Comme beaucoup de perceptions héritées du passé, cette vision des choses est complètement fausse. Les deux projets ont évolué et la comparaison est désormais plus complexe que cela.

L'un des principaux soucis de MySQL est son utilisation de différents moteurs qui activent/désactivent certaines fonctionnalités. La liste des fonctionnalités de MySQL est impressionnante mais il faut savoir que, pour bénéficier de certaines fonctionnalités, il faut en abandonner d'autres. Ce n'est pas le cas avec PostgreSQL où toutes les fonctionnalités sont disponibles en permanence, quelle que soit la configuration. Il est par exemple impossible d'utiliser l'indexation Full Text ou les extensions géographiques de MySQL sur autre chose que MyISAM, ce qui empêche l'utilisation de transactions.

Un des manques importants du moteur InnoDB est son incapacité à faire du transactionnel sur les requêtes de modification de schémas (DDL).

Le `SELECT count(*)` est généralement très coûteux avec PostgreSQL. MySQL, avec le moteur MyISAM, conserve sur ce point des performances que PostgreSQL ne peut pas concurrencer : MyISAM stocke un compteur d'enregistrements de la table en entête de celle-ci. Ceci n'est évidemment possible que pour un moteur non transactionnel, puisque deux transactions peuvent « voir » un nombre d'enregistrement différents dans une table suivant ce qui est validé ou non pour chacune d'entre elles. Néanmoins, cette requête est moins coûteuse depuis la version 9.2 de PostgreSQL.

Mais, malgré quelques atouts techniques, MySQL a actuellement un gros problème. Le rachat de MySQL AB par Sun puis par Oracle laisse de grosses incertitudes sur l'avenir de MySQL. Étant donné le comportement d'Oracle avec les outils libres, beaucoup de développeurs ont préféré démissionner. Beaucoup ont lancé leur propre fork (version dérivée) de MySQL (Drizzle, MariaDB, Falcon…). Ces forks sont généralement des versions comprenant moins de fonctionnalités que la version 5.5, ou une réimplémentation différente des nouveautés de la version 5.5.

3.3.2 POSTGRESQL VS. SQL SERVER

- Des performances difficile à battre sous Windows
  - … mais PostgreSQL reste plus efficace sous Linux
- Intégration à l'écosystème Microsoft
  - avantage, excellents outils graphiques (Studio)
  - inconvénient, peu de choix dans les outils
- Points faibles de PostgreSQL
  - un partitionnement perfectible (amélioration en 10)
  - pas de vues matérialisées (amélioration en 9.3)

Difficile de battre Microsoft pour fournir un serveur de bases de données plus rapide que SQL Server (créé par Microsoft) sur Windows (créé aussi par Microsoft). Du coup,

17.12

PostgreSQL a de fortes chances d'être moins performant que SQL Server sous Windows. Par contre, une comparaison de SQL Server sur Windows avec PostgreSQL sous Linux donne souvent un résultat inverse. En 2010, Red Hat a publié une étude de performances détaillées montrant la supériorité du couple « PostgreSQL + Linux » face à « SQL Server + Windows » : http://www.redhat.com/pdf/rhel/bmsql-postgres-sqlsrvr-v1.0-1.pdf

SQL Server et PostgreSQL s'opposent aussi sur la philosophie. Le premier propose de très nombreux outils périphériques (console, ETL...), et vous impose tout l'écosystème de Microsoft. PostgreSQL se concentre sur son rôle de SGBD et ne vous impose rien d'autre, ni en terme d'outils, ni en terme de société de support.

Au niveau des fonctionnalités, PostgreSQL ne dispose des vues matérialisées qu'à partir de la version 9.3 alors que SQL Server en dispose depuis bien longtemps. De plus, elles ne sont pas aussi avancées que ce que propose SQL Server. Le partitionnement de PostgreSQL était manuel et très rudimentaire par rapport à ce qui est disponible dans SQL Server. Cependant, la version 10 améliore bien la situation pour PostgreSQL.

3.3.3 POSTGRESQL VS. ORACLE

- PostgreSQL est le SGBD le plus proche d'Oracle
- Points forts de PostgreSQL
  - respect des standards
  - DDL transactionnel
  - pas de gestion et de coût de licence
- Points faibles de PostgreSQL
  - manque certains objets (synonymes, packages)
  - un parallélisme perfectible (amélioration en 9.6)
  - un partitionnement perfectible (amélioration en 10)
  - pas de vues matérialisées (amélioration en 9.3)
  - lenteur du `SELECT count(*)` (amélioration en 9.2)
  - pas d'équivalent de RAC
- Deux produits à présent dans la même catégorie

En terme de fonctionnalités et de fiabilité, PostgreSQL a surtout Oracle comme concurrent, et non pas MySQL (ce qui semblerait le concurrent le plus logique du fait de leur licence libre).

PostgreSQL ne dispose pas de l'intégralité de la pléthore de fonctionnalités d'Oracle. Par exemple, les vues matérialisées ne sont pas disponibles sur PostgreSQL avant la 9.3. Le

partitionnement est possible mais est assez complexe à mettre en place pour les versions antérieures à la 10.

Pour le `SELECT count(*)`, Oracle passe par une lecture de l'index, ce qui améliore fortement les performances. PostgreSQL passe aussi par une lecture d'index à partir de PostgreSQL 9.2, mais les versions précédentes ne pouvaient pas le faire. PostgreSQL ne disposait pas d'un mode d'exécution en parallèle avant la version 9.6 (ce qui ne peut poser problème qu'en contexte d'infocentre).

RAC n'a tout simplement pas d'équivalent dans le monde PostgreSQL.

PostgreSQL ne dispose pas d'équivalent complet à Oracle Enterprise Manager ou à Grid Control, les consoles de supervision d'Oracle.

PostgreSQL rattrape son retard sur beaucoup de fonctionnalités comme le partitionnement, les vues matérialisées, le parallélisme. Les fonctionnalités de Streaming Replication et Hot Standby sont l'équivalent de Oracle Data Guard. Les fonctions de fenêtrage sont très ressemblantes. Les performances sont très proches pour la plupart des cas d'utilisation.

PostgreSQL possède de l'avance sur certaines fonctionnalités, comme l'implémentation de SSI (Serializable Snapshot Isolation) ou comme le support du DDL transactionnel. Une transaction commencée sera automatiquement terminée avec un `COMMIT` au premier ordre DDL arrivé dans cette transaction.

Oracle pâtit également de la politique commerciale de son éditeur quant aux coûts de licence, et aux audits. La tendance du marché est de migrer des bases Oracle vers PostgreSQL, pas l'inverse.

3.3.4 POSTGRESQL VS. INFORMIX

- Informix est un lointain cousin de PostgreSQL
- Points forts de PostgreSQL
  - un vrai `CREATER USER`
  - pas de double quote (`"`) pour les chaînes
- Points faibles de PostgreSQL
  - pas de `synonym`

En 1995, la version « universitaire » de Postgres fut commercialisée sous le nom d'Illustra par une société éponyme dirigée notamment par Michael Stonebraker. Les deux logiciels évoluèrent de manière différente et Illustra a fini par être racheté en 1997 par Informix (maintenant détenu par IBM).

17.12

Il n'y a pas d'équivalent au concept de synonyme dans PostgreSQL. On utilise le `search_path` mais ce n'est qu'un contournement.

Par contre, pour ajouter un utilisateur de bases dans Informix, il faut créer un nouvel utilisateur système. Ce n'est pas le cas dans PostgreSQL, ce qui le sépare de la gestion du système d'exploitation.

La gestion des chaînes de caractères est différente. PostgreSQL n'accepte pas les double guillemets pour entourer les chaînes, conformément à la norme SQL (les double guillemets étant réservés pour spécifier des noms d'objets).

3.3.5 POSTGRESQL VS. NOSQL

- 4 technologies majeures dans le monde NoSQL
  - stockage clé->valeur (Redis, Apache Cassandra, Riak, MongoDB)
  - stockage documents (Apache CouchDB, MongoDB)
  - stockage colonnes (Apache Hbase, Google BigTable)
  - stockage graphes (Neo4j)
- PostgreSQL réunit le monde relationnel et le monde NoSQL
  - stockage clé->valeur : `hstore`
  - stockage documents : `xml`, `json` et `jsonb` (plus performant que MongoDB)
  - procédure stockée en Javascript : PL/V8
  - stockage colonnes : `cstore_fdw`

PostgreSQL n'est pas en reste vis à vis des bases de données NoSQL. PostgreSQL permet de stocker des données au format clé->valeur. Couplé aux index GiST et GIN, ce mode de stockage s'avère comparable à MongoDB (https://wiki.postgresql.org/images/b/b4/Pg-as-nosql-pgday-fosdem-2013.pdf) ou à Redis.

PostgreSQL peut stocker des données au format JSON (depuis la version 9.2, avec de nombreuses fonctions ajoutées en 9.3). L'avantage est que les données seront validées : si une donnée n'est pas conforme au format JSON, elle sera rejetée. Avec le type natif `jsonb` (en 9.4) il est possible d'indexer le JSON, et les performances de PostgreSQL avec ce nouveau type de stockage de documents sont très supérieures à MongoDB (http://blogs.enterprisedb.com/2014/09/24/postgres-outperforms-mongodb-and-ushers-in-new-developer-reality/).

Couplé au langage PL/V8, le type de données JSON permet de traiter efficacement ce type de données. PL/V8 permet d'écrire des fonctions en langage Javascript, elles seront exécutées avec l'interpréteur V8 écrit par Google. Ces fonctions sont bien entendus indexables par PostgreSQL si elles respectent un certain nombre de pré-requis.

Le stockage colonne pour PostgreSQL consiste actuellement en une extension *Foreign Data Wrapper* nommée `cstore_fdw`. Elle est développée par la société CitusData (http://www.citusdata.com/blog/76-postgresql-columnar-store-for-analytics).

Malgré cela, PostgreSQL conserve les fonctionnalités qui ont fait sa réputation et que les moteurs NoSQL ont dû abandonner :

- un langage de requête puissant ;
- un optimiseur de requêtes sophistiqué ;
- la normalisation des données ;
- les jointures ;
- l'intégrité référentielle ;
- la durabilité.

Voici enfin un lien vers une excellente présentation sur les différences entre PostgreSQL et les solutions NoSQL : http://fr.slideshare.net/EnterpriseDB/the-nosql-way-in-postgres

ainsi que l'avis de Bruce Momjian sur le choix du NoSQL pour de nouveaux projets[62]

3.4 À LA RENCONTRE DE LA COMMUNAUTÉ

- Cartographie du projet
- Pourquoi participer
- Comment participer

3.4.1 POSTGRESQL, UN PROJET MONDIAL

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

Quelques faits :

- Le projet est principalement anglophone.
- Il existe une très grande communauté au Japon.
- La communauté francophone est très dynamique mais on trouve très peu de développeurs francophones.
- La communauté hispanophone est naissante.
- Les développeurs du noyau (*core hackers*) vivent en Europe, au Japon, en Russie et en Amérique du Nord.

[62] https://momjian.us/main/blogs/pgblog/2017.html#June_12_2017

17.12

FIGURE 2: CARTE DES HACKERS

3.4.2 POSTGRESQL CORE TEAM

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

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

La « Core Team » est un ensemble de personnes doté d'un pouvoir assez limité. Ils peuvent décider de la sortie d'une version. Ce sont les personnes qui sont immédiatement au courant des failles de sécurité du serveur PostgreSQL. Tout le reste des décisions est pris par la communauté dans son ensemble après discussion, généralement sur la liste pgsql-hackers.

Détails sur les membres actuels de la *core team* :

- **Peter Eisentraut**, 2nd Quadrant, Philadelphie (États-Unis) - développement du moteur (internationalisation, SQL/Med, etc.)
- **Tom Lane**, Crunchy Data, Pittsburgh (USA) - certainement le développeur le plus aguerri

17.12

- **Bruce Momjian**, EnterpriseDB, Philadelphia (USA) - a lancé le projet, fait principalement de la promotion
- **Dave Page**, EnterpriseDB, Oxfordshire (Angleterre) - leader du projet pgAdmin, administration des serveurs
- **Magnus Hagander**, Redpill Linpro, Stockholm (Suède) - développeur PostgreSQL, administration des serveurs, président de PostgreSQL Europe

---

3.4.3 CONTRIBUTEURS

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

- développement des projets satellites (Slony, pgAdmin, ...) ;
- promotion du logiciel ;
- administration des serveurs ;
- rédaction de documentation ;
- conférences ;
- organisation de groupes locaux.

Le *PGDG* a fêté son 10e anniversaire à Toronto en juillet 2006. Ce « PostgreSQL Anniversary Summit » a réuni pas moins de 80 membres actifs du projet.

PGCon2009 a réuni 180 membres actifs à Ottawa.

Voir la liste des contributeurs officiels[63] .

3.4.4 UTILISATEURS

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

Il est impossible de connaître précisément le nombre d'utlisateurs de PostgreSQL. On sait toutefois que ce nombre est en constante augmentation.

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

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

3.4.5 POURQUOI PARTICIPER

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

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

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

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

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

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

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

3.4.6 SERVEURS

- Site officiel : http://www.postgresql.org/
- La doc : http://www.postgresql.org/docs/
- Actualité : http://planet.postgresql.org/
- Des extensions : http://pgxn.org/

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

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

Le site PGXN est l'équivalent pour PostgreSQL du CPAN de Perl, une collection en ligne de librairies et extensions accessibles depuis la ligne de commande. Il remplace petit à petit le site pgfoundry.org. Ce dernier est de plus en plus délaissé. Il s'agissait d'un SourceForge dédié à PostgreSQL, mais le manque d'administration fait que les développeurs l'abandonnent de plus en plus (ils vont généralement sur github ou sur le vrai SourceForge). Actuellement, il est déconseillé d'ouvrir un projet sur pgfoundry.org.

3.4.7 SERVEURS FRANCOPHONES

- Site officiel : http://www.postgresql.fr/
- La doc : http://docs.postgresql.fr/
- Forum : http://forums.postgresql.fr/
- Actualité : http://planete.postgresql.fr/
- Wiki : http://wiki.postgresql.fr/

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

3.4.8 LISTES DE DISCUSSIONS / LISTES D'ANNONCES

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

Les mailing-lists sont les outils principaux de gouvernance du projet. Toute l' activité de la communauté (bugs, promotion, entraide, décisions) est accessible par ce canal.

Pour s'inscrire ou consulter les archives : http://www.postgresql.org/community/lists/

Si vous avez une question ou un problème, la réponse se trouve probablement dans les archives ! Pourquoi ne pas utiliser un moteur de recherche spécifique ?

- http://www.nabble.com/
- http://markmail.org/

N'hésitez pas à rejoindre ces listes.

Les listes de diffusion sont régies par des règles de politesse et de bonne conduite. Avant de poser une question, nous vous conseillons de consulter le guide suivant : http://www.linux-france.org/article/these/smart-questions/smart-questions-fr.html

---

3.4.9 IRC

- Réseau Freenode
- IRC anglophone
  - `#postgresql`
  - `#postgresql-eu`
- IRC francophone
  - `#postgresqlfr`

Le point d'entrée principal pour le réseau Freenode est le serveur : `irc.freenode.net`. La majorité des développeurs sont disponibles sur IRC et peuvent répondre à vos questions.

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

17.12

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

3.4.10 WIKI

- http://wiki.postgresql.org/
- http://wiki.postgresql.org/wiki/Fran%C3%A7ais

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

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

Il existe une partie spécifiquement en français, indiquant la liste des documents écrits en français sur PostgreSQL. Elle n'est pas exhaustive et souffre fréquemment d'un manque de mises à jour.

3.4.11 L'AVENIR DE POSTGRESQL

- PostgreSQL 10 est sortie en septembre 2017
- Grandes orientations :
  - réplication logique
  - meilleur parallélisme
  - gros volumes
- Prochaine version, la 11
- Stabilité économique
- Le futur de PostgreSQL dépend de vous !

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

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

- etc.

PostgreSQL est là pour durer. Il n'y a pas qu'une seule entreprise derrière ce projet. Il y en a plusieurs, petites et grosses sociétés, qui s'impliquent pour faire avancer le projet.

3.5 CONCLUSION

- Beaucoup de projets complémentaires
- Une communauté active
- Concurrent solide face aux SGBD propriétaires
- De nombreuses conversions en cours vers PostgreSQL

3.5.1 BIBLIOGRAPHIE

- Organisation du projet PostgreSQL, Guillaume Lelarge, 2010[64]

Iconographie :

La photo initiale est sous licence CC-BY-SA[65] : http://www.flickr.com/photos/st3f4n/675708572/

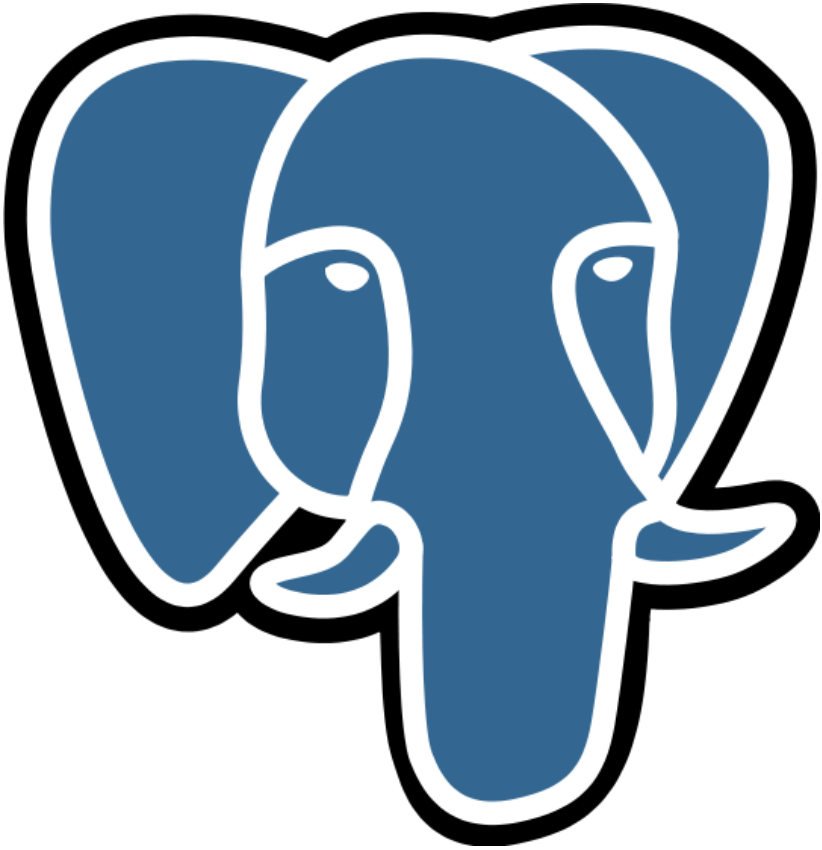
3.5.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[64] http://www.dalibo.org/organisation_du_projet_postgresql
[65] http://creativecommons.org/licenses/by-sa/2.0/deed.fr

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[66] .

4.2.5 AUTHENTIFICATION D'UN CLIENT

- En interactif

[66] 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)
```

`\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"[67]

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

[67] 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

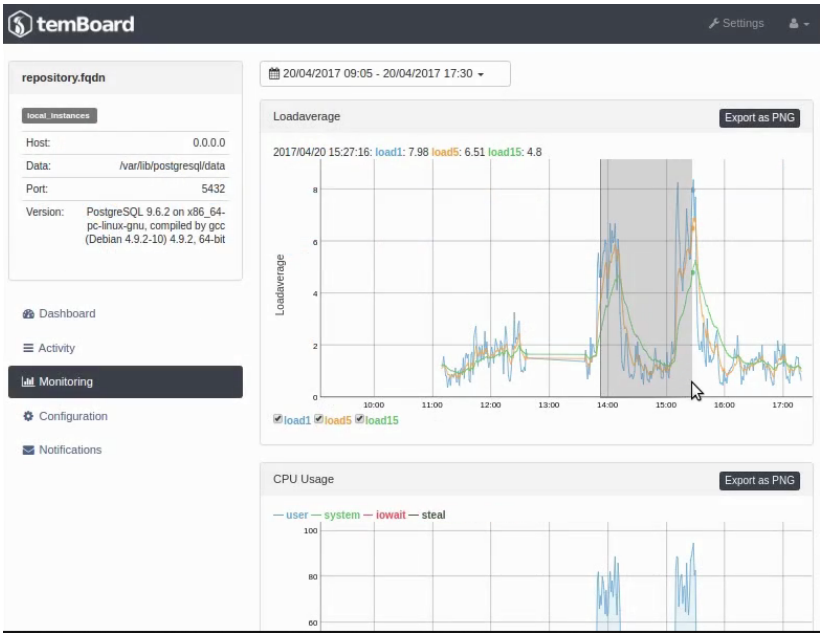
4.5.2 TEMBOARD - POSTGRESQL REMOTE CONTROL

- 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.

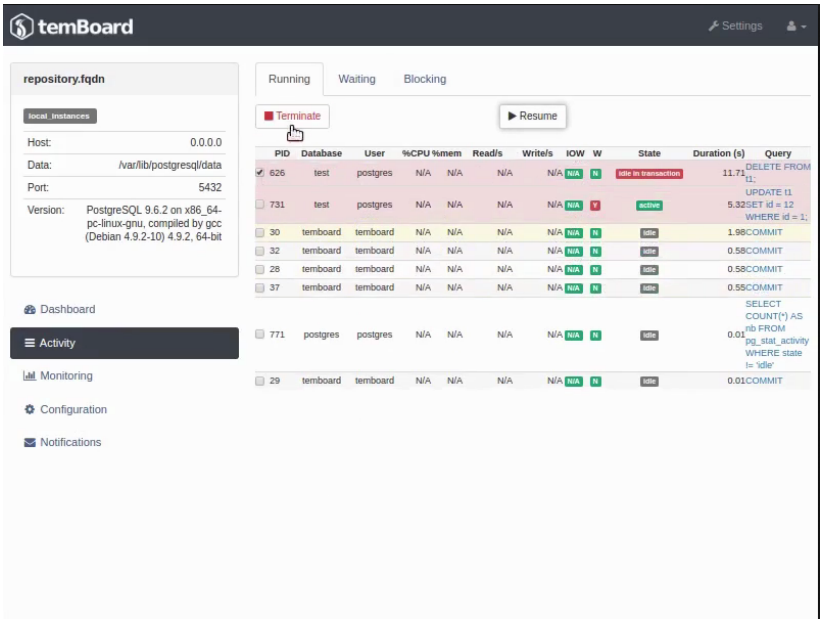
Le serveur web est installé de façon centralisée et un agent est déployé pour chaque instance.

4.5.3 TEMBOARD - MONITORING



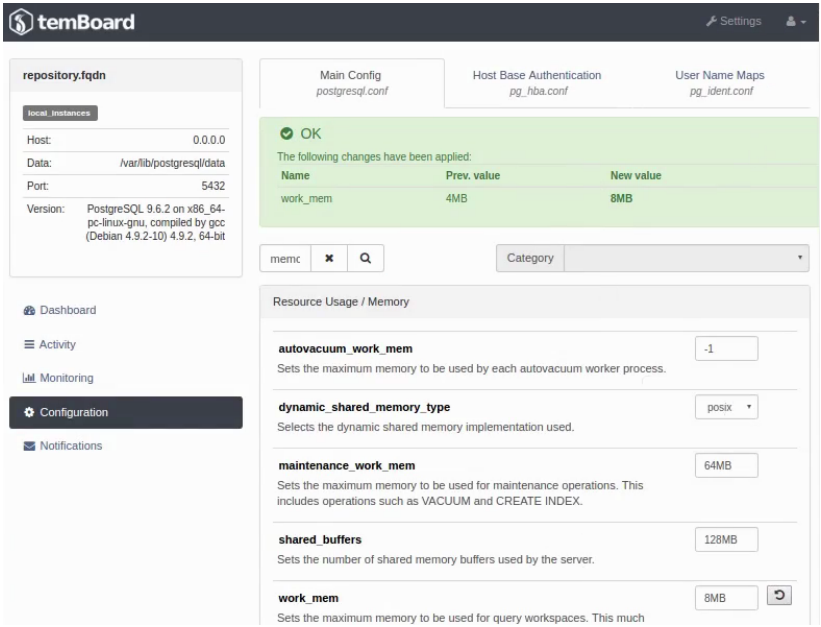
La section **Monitoring** permet de visualiser les graphiques historisés au niveau du système d'exploitation (CPU, mémoire, ...) ainsi qu'au niveau de l'instance PostgreSQL.

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)

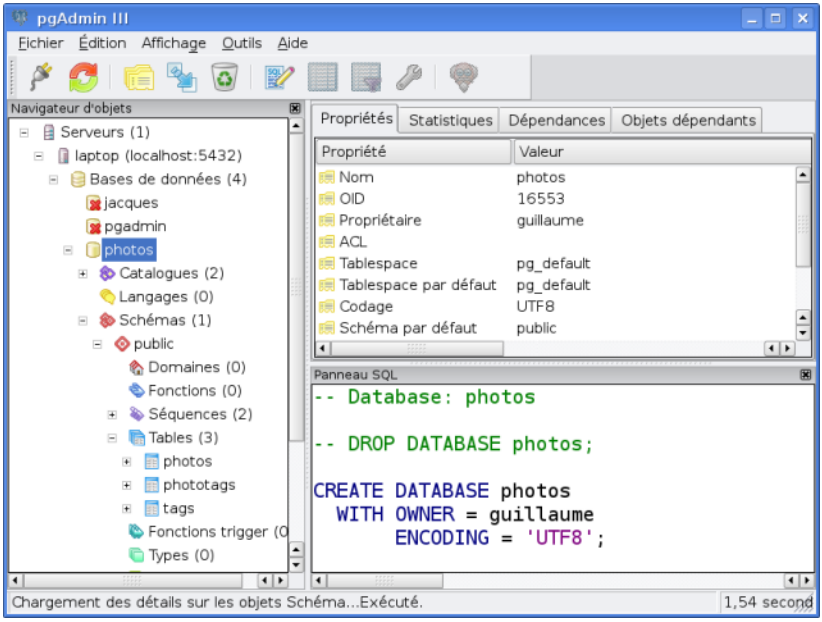
- Sauvegarde et restauration d'une base
- Gestion des rôles
- Création/modification/suppression de tous les objets PostgreSQL
- Possibilité de voir/cacher les objets systèmes
- Éditeur de requête

17.12

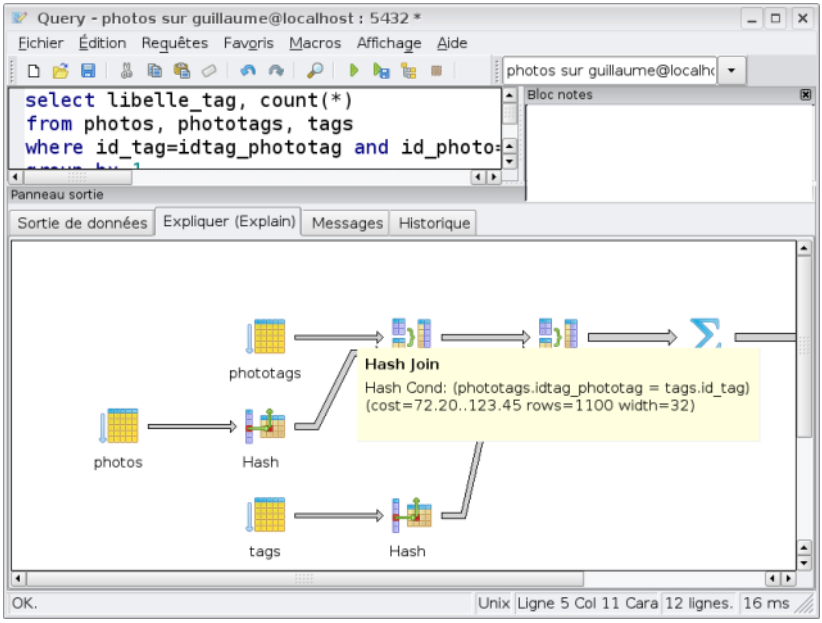
Les objets gérables par `pgAdmin` sont :

- 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

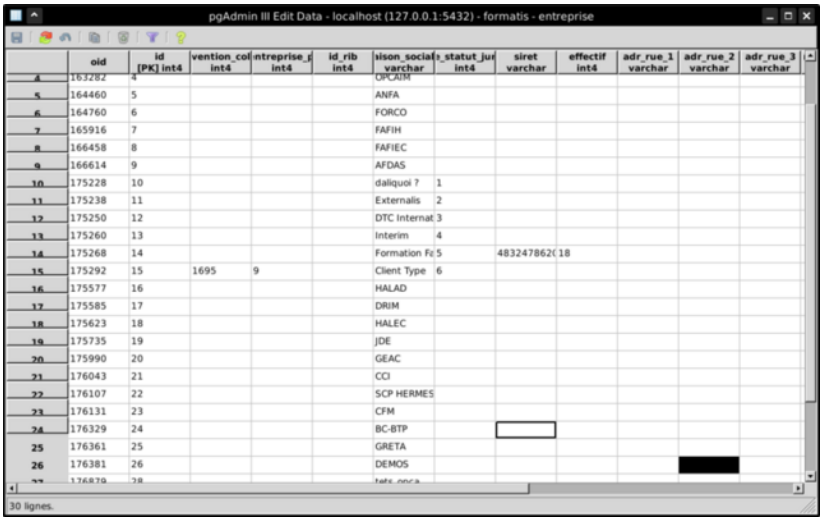


L'éditeur de requête permet de :

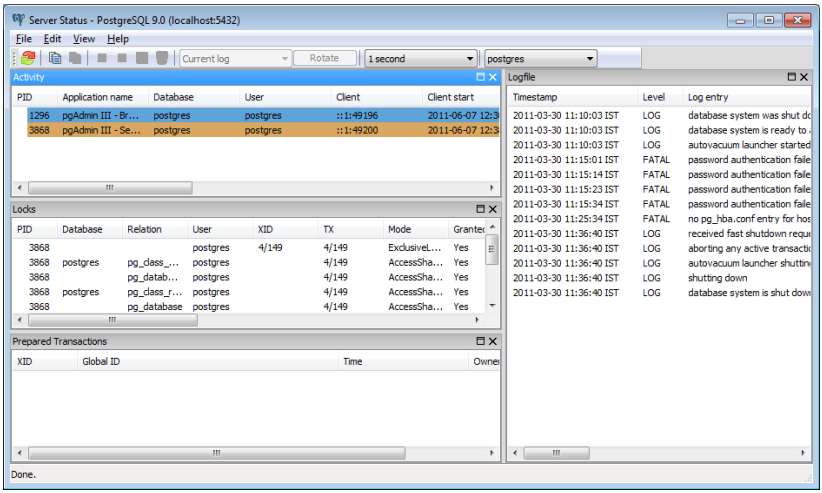
- lire/écrire un fichier de requêtes ;
- exécuter une requête ;
- sauvegarder le résultat d'une requête dans un fichier ;
- exécuter un `EXPLAIN` (avec les options `verbose` et `analyze`) sur une requête.

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

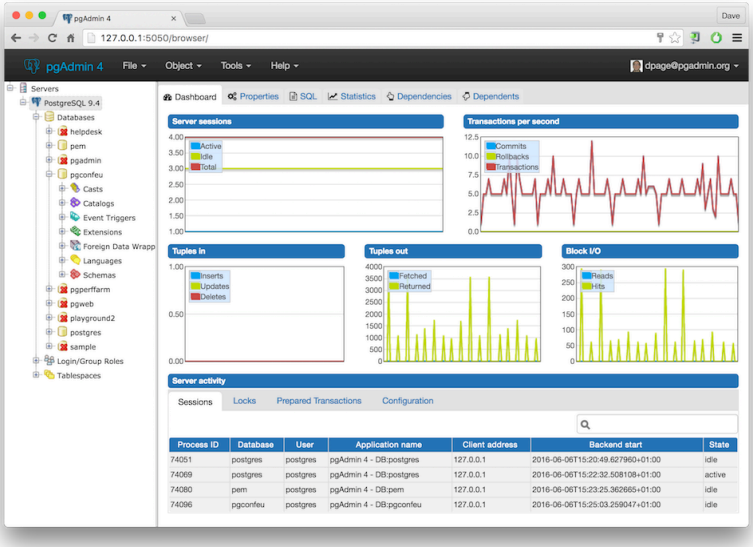


4.5.14 PGADMIN IV

- 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

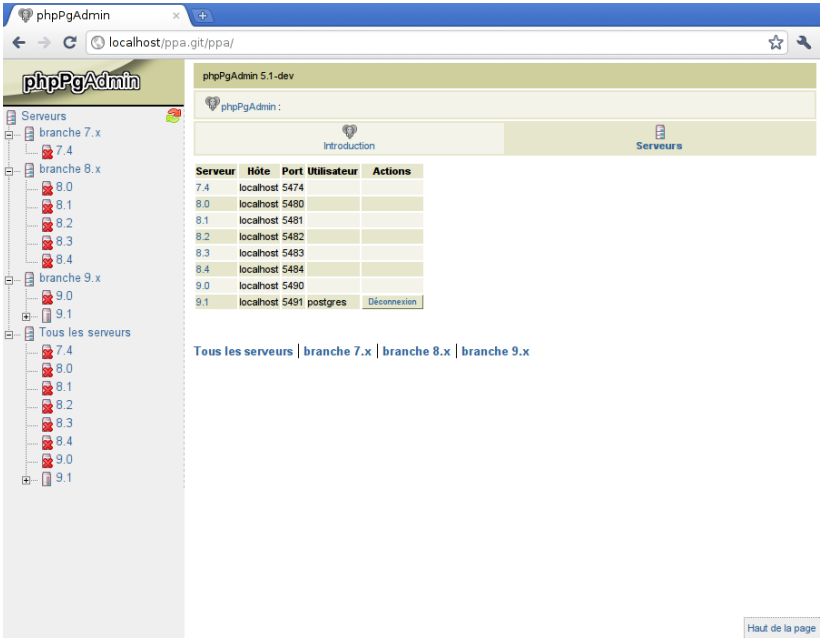
phpPgAdmin a certains avantages:

- 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

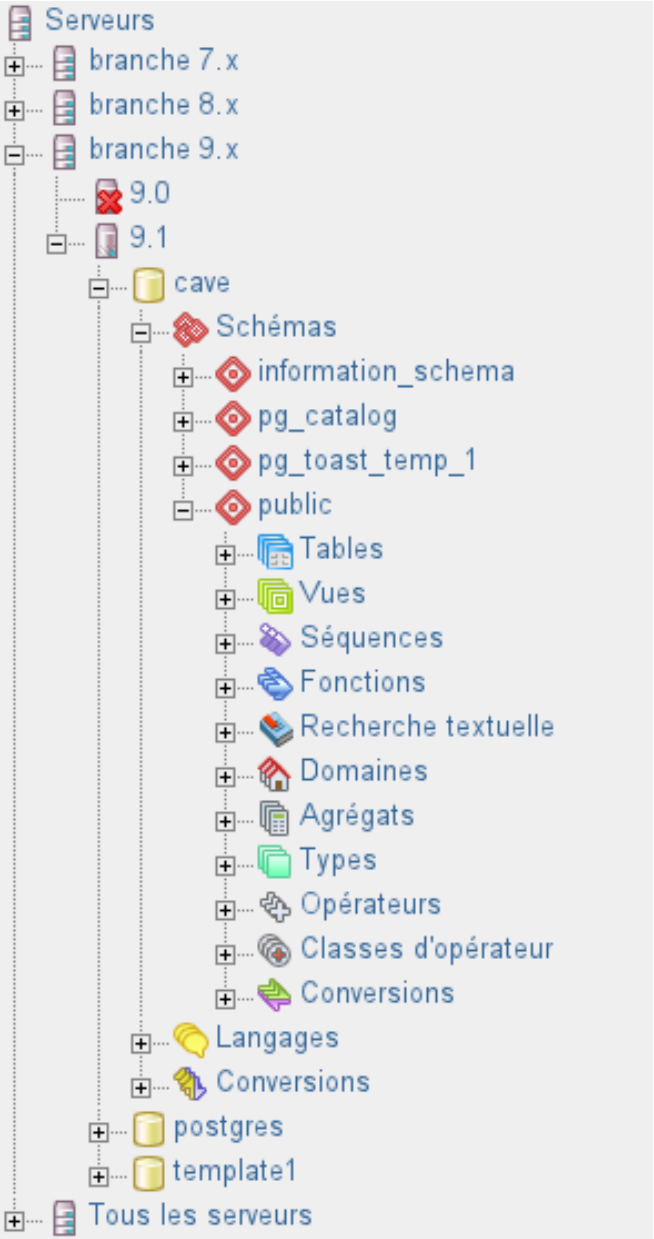


phpPgAdmin se compose de deux volets:

- 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:

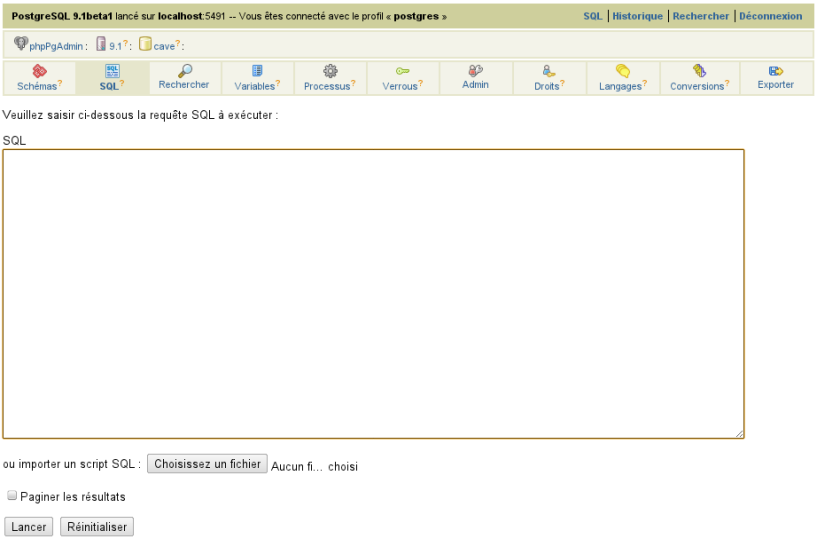
```
// Groups allow administrators to logicaly group servers together under group
// nodes in the left browser tree
$conf['srv_groups'][0]['desc'] = "branche 7.x";
$conf['srv_groups'][0]['servers'] = '0';
$conf['srv_groups'][1]['desc'] = 'branche 8.x';
$conf['srv_groups'][1]['servers'] = '1,2,3,4,5';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



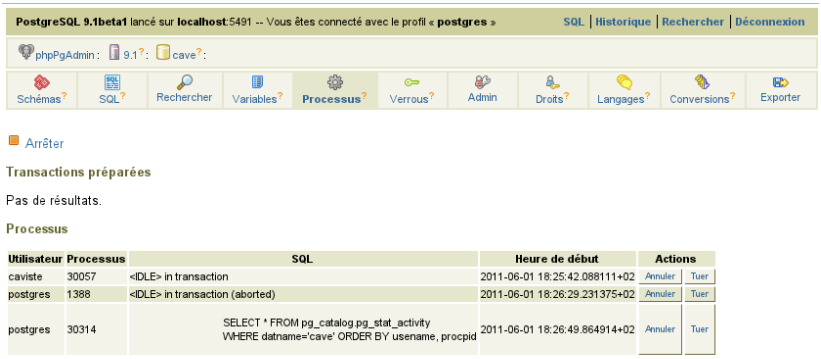
L’exécution de requêtes SQL depuis phpPgAdmin se fait depuis l’onglet ”SQL” accessible au niveau de chaque base de données ou d’un de leurs schémas.

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 ARCHITECTURES DE HAUTE-DISPONIBILITÉ

FIGURE 3: POSTGRESQL

5.1 PRÉAMBULE

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

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

5.1.1 AU MENU

- Rappels théoriques
- Réplication interne
  - réplication physique
  - réplication logique
- 5 logiciels externes de réplication
- Alternatives

Dans cette présentation, nous reviendrons rapidement sur la classification des solutions de réplication, qui sont souvent utilisés dans un but de haute disponibilité, mais pas uniquement.

PostgreSQL dispose d'une réplication physique basée sur le rejeu des journaux de transactions par un serveur dit « en Standby ». Nous présenterons ainsi les techniques dites de « Warm Standby » et de « Hot Standby ».

Il dispose aussi depuis la version 10 d'une réplication logique. Elle sera aussi présentée.

Nous détaillerons ensuite les projets de réplication autour de PostgreSQL les plus en vue actuellement.

5.1.2 OBJECTIFS

- Connaître les principaux outils de réplication
- Identifier les différences entre les solutions proposées
- Choisir le système le mieux adapté à votre besoin

La communauté PostgreSQL propose plusieurs réponses aux problématiques de réplication. Le but de cette présentation est de vous apporter les connaissances nécessaires pour comparer chaque solution et comprendre les différences fondamentales qui les séparent.

À l'issue de cette présentation, vous serez capable de choisir le système de réplication qui correspond le mieux à vos besoins et aux contraintes de votre environnement de production.

5.2 RAPPELS THÉORIQUES

- Cluster
- Réplication
  - synchrone / asynchrone
  - symétrique / asymétrique
  - diffusion des modifications

Le domaine de la haute-disponibilité est couvert par un bon nombre de termes qu 'il est préférable de définir avant de continuer.

5.2.1 CLUSTER

- Dans la terminologie PostgreSQL
  - groupe autonome de bases de données
  - i.e. une instance
- Dans la terminologie haute-disponibilité et/ou réplication
  - groupe de serveurs

Toute la documentation (anglophone) de PostgreSQL parle de cluster dans le contexte d'un serveur PostgreSQL seul. Dans ce contexte, le cluster est un groupe de bases de données, groupe étant la traduction directe de cluster.

Dans le domaine de la haute-disponibilité et de la réplication, un cluster désigne un groupe de serveurs. Par exemple, un groupe d'un serveur maître et de ses deux serveurs esclaves compose un cluster de réplication.

5.2.2 RÉPLICATION ASYNCHRONE ASYMÉTRIQUE

- Asymétrique
  - écritures sur un serveur maître unique
  - lectures sur le maître et/ou les esclaves
- Asynchrone
  - les écritures sur les esclaves sont différées
  - perte de données possible en cas de crash du maître
- Quelques exemples
  - streaming replication, Slony, Londiste, Bucardo

Dans la réplication asymétrique, seul le maître accepte des écritures, et les esclaves ne sont accessibles qu'en lecture.

Dans la réplication asynchrone, les écritures sont faites sur le maître et, avant qu'elles ne soient poussées vers l'esclave, le client a un retour lui indiquant que l'écriture s'est bien passée. La mise à jour des tables répliquées **est différée** (asynchrone). Elle est réalisée par un programmateur de tâches, possédant une horloge. Des points de synchronisation sont utilisés pour propager les changements.

L'inconvénient de ce système est que, si un crash intervient sur le maître après que le client ait eu la réponse du succès de l'écriture mais avant que les données ne soient poussées sur l'esclave, certaines données validées sur le maître ne seront pas disponibles sur l'esclave. Autrement dit, il existe une fenêtre, plus ou moins importante, de perte de données.

5.2.3 RÉPLICATION ASYNCHRONE SYMÉTRIQUE

- Symétrique
  - écritures sur les différents maîtres
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents maîtres
- Asynchrone
  - les écritures sur les esclaves sont différées
  - perte de données possible en cas de crash du maître

Dans la réplication symétrique, tous les serveurs sont accessibles aux utilisateurs, aussi bien en lecture qu'en écriture. La réplication asynchrone, comme indiquée précédemment, met en attente l'envoi des modifications sur les esclaves, donc il y a toujours un risque de perte de données si le maître tombe sans avoir eu le temps d'envoyer les données à au moins un esclave

Ce mode de réplication ne respecte généralement pas les propriétés `ACID` (atomicité, cohérence, isolation, durabilité) car si une copie échoue sur l'autre maître alors que la transaction a déjà été validée, on peut alors arriver dans une situation où les données sont incohérentes entre les serveurs.

Généralement, ce type de système doit proposer un gestionnaire de conflits, de préférence personnalisable.

5.2.4 RÉPLICATION SYNCHRONE ASYMÉTRIQUE

- Asymétrique
  - écritures sur un serveur maître unique
  - lectures sur le maître et/ou les esclaves
- Synchrone
  - les écritures sur les esclaves sont immédiates
  - le client sait si sa commande a réussi sur l'esclave

Dans la réplication asymétrique, seul le maître accepte des écritures, et les esclaves ne sont accessibles qu'en lecture.

Dans la réplication synchrone, le client envoie sa requête en écriture sur le maître, le maître l'écrit sur son disque, il envoie les données à l' esclave, attend que ce dernier l'écrive sur son disque. Si tout ce processus s' est bien passé, le client est averti que l'écriture a été réalisée avec succès. On utilise généralement un mécanisme dit de **Two Phase Commit** ou "Validation en deux phases", qui assure qu'une transaction est validée sur tous les nœuds dans la même transaction. Les propriétés `ACID` sont dans ce cas respectées.

Le gros avantage de ce système est qu'il n'y a pas de risque de perte de données quand le maître s'arrête brutalement et qu'on doive repartir sur l' esclave. L'inconvénient majeur est que cela ralentit fortement les écritures.

Ce type de réplication garantit que l'esclave a bien écrit la transaction dans ses journaux et qu'elle a été synchronisé sur disque (fsync). En revanche elle ne garantit pas que l'esclave a bien rejoué la transaction. Il peut se passer un laps de temps très court où une lecture sur l'esclave serait différente du maître (le temps que l'esclave rejoue la transaction).

5.2.5 RÉPLICATION SYNCHRONE SYMÉTRIQUE

- Symétrique
  - écritures sur les différents maîtres
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents maîtres
- Synchrone
  - les écritures sur les esclaves sont immédiates
  - le client sait si sa commande a réussi sur l'esclave
  - risque important de lenteur

Ce système est le plus intéressant en théorie. L'utilisateur peut se connecter à n'importe quel serveur pour des lectures et des écritures. Il n'y a pas de risques de perte de don-

nées vu que la commande ne réussit que si les données sont bien enregistrées sur tous les serveurs. Autrement dit, c'est le meilleur système de réplication et de répartition de charge.

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

Postgres-X2 (anciennement appelé "Postgres-XC") n'est pas un choix pérenne, il s'agit d'un fork de PostgreSQL 9.3 ce qui signifie qu'à l'horizon 2018, sa base de code sera périmée. Par ailleurs, c'est un projet qui nécessite un investissement très important en terme de matériel et maintenance opérationnelle.

Pgpool semblait prometteur, mais certaines fonctions (notamment le load-balancing et la réplication) se révèlent souvent complexes à mettre en œuvre, difficile à stabiliser et limitées à des cas d'utilisation très spécifiques. Malgré son ancienneté il y a encore beaucoup de corrections de bugs à chaque mise à jour.

5.2.6 DIFFUSION DES MODIFICATIONS

- 4 types de récupération des informations de réplication
- Par requêtes
  - diffusion de la requête
- Par triggers
  - diffusion des données résultant de l'opération
- Par journaux, physique
  - diffusion des blocs disques modifiés
- Par journaux, logique
  - extraction et diffusion des données résultant de l'opération depuis les journaux

La récupération des données de réplication se fait de différentes façons suivant l'outil utilisé.

La diffusion de l'opération de mise à jour (donc **le SQL lui-même**) est très flexible et compatible avec toutes les versions. Cependant, cela pose la problématique des opérations dites non déterministes. L'insertion de la valeur `now()` exécutée sur différents serveurs fera que les données seront différentes, généralement très légèrement différentes, mais différentes malgré tout. L'outil pgPool, qui implémente cette méthode de réplication, est capable de récupérer l'appel à la fonction `now()` pour la remplacer par la date et l'heure. Il est capable de le faire car il connaît les différentes fonctions de date et heure proposées

en standard par PostgreSQL. Cependant, il ne connaît pas les fonctions utilisateurs qui pourraient faire de même. Il est donc préférable de renvoyer les données, plutôt que les requêtes.

Le renvoi du résultat peut se faire de deux façons : soit en récupérant les nouvelles données avec un trigger, soit en récupérant les nouvelles données dans les journaux de transactions.

Cette première solution est utilisée par un certain nombre d'outils externes de réplication, comme Slony, Londiste ou Bucardo. Les fonctions triggers étant écrites en C, cela assure de bonnes performances. Cependant, seules les modifications des données sont attrapables avec des triggers. Les modifications de la structure de la base ne le sont pas (l'ajout des triggers sur événement en 9.3 est une avancée intéressante pour permettre ce genre de fonctionnalités dans le futur). Autrement dit, l'ajout d'une table, l'ajout d' une colonne demande une administration plus poussée, non automatisables.

La deuxième solution (par journaux de transactions) est bien plus intéressante car les journaux contiennent toutes les modifications, données comme structures. De ce fait, une fois mise en place, elle ne demande pas une grosse administration.

Depuis PostgreSQL 9.4, un nouveau niveau `logical` a été ajouté dans le paramétrage des journaux de transactions (paramètre `wal_level`). Couplé à l'utilisation des slots de réplication (nouveau paramètre `max_replication_slots`), il permet le décodage logique des modifications de données correspondant aux blocs modifiés dans les journaux de transactions. L' objectif était de permettre la reconstitution d'un ordre SQL permettant d'obtenir le même résultat, ce qui permettrait la mise en place d'une réplication logique des résultats entièrement intégrée, donc sans triggers. Ceci est disponible depuis la version 10 de PostgreSQL.

5.3 RÉPLICATION INTERNE PHYSIQUE

- Réplication
  - asymétrique
  - asynchrone ou synchrone
- Esclaves
  - non disponibles (Warm Standby)
  - disponibles en lecture seule (Hot Standby, à partir de la 9.0)
  - cascade possible à partir de la 9.2
  - retard possible à partir de la 9.4

Ce mode de réplication est par défaut **asynchrone** et **asymétrique**. Le mode synchrone est disponible à partir de la version 9.1. Il est même possible de sélectionner le mode synchrone/asynchrone pour chaque esclave individuellement.

Il fonctionne par l'envoi des enregistrements des journaux de transactions, soit par envoi de fichiers complets (on parle de `Log Shipping`), soit par envoi de groupes d'enregistrements en flux (on parle là de `Streaming Replication`), puisqu'il s'agit d'une réplication par diffusion de journaux.

La différence entre `Warm Standby` et `Hot Standby` est très simple :

- un serveur esclave en Warm Standby est un serveur de secours sur lequel il n'est pas possible de se connecter ;
- un serveur esclave en Hot Standby accepte les connexions et permet l'exécution de requêtes en lecture seule.

À partir de la version 9.2, un esclave peut récupérer les informations de réplication d'un autre esclave. À partir de la 9.4, il peut appliquer les informations de réplication après un délai configurable.

5.3.1 LOG SHIPPING

- But
  - envoyer les journaux de transactions à un esclave
- Première solution disponible (dès 2006)
- Gros inconvénient
  - il est possible de perdre un journal de transactions entier

Le Log Shipping permet d'envoyer les journaux de transactions terminés sur un autre serveur. Ce dernier peut être un serveur esclave, en Warm Standby ou en Hot Standby, prêt à les rejouer.

Cependant, son gros inconvénient vient du fait qu'il faut attendre qu'un journal soit complètement écrit pour qu'il soit propagé vers l'esclave. Autrement dit, il est possible de perdre les transactions contenues dans le journal de transactions en cours en cas de failover. Sans même ce problème, cela veut aussi dire que le retard de l'esclave sur le maître peut être assez important, ce qui est gênant dans le cas d'un Hot Standby qu'on peut utiliser en lecture seule, par exemple dans le cadre d'une répartition de la charge de lecture.

5.3.2 STREAMING REPLICATION

- But
  - avoir un retard moins important sur le maître
- Rejouer **les enregistrements de transactions** du maître par **paquets**
  - paquets plus petits qu'un journal de transactions
- Solution idéalement couplée au `Hot Standby`

L'objectif du mécanisme de `Streaming Replication` est d'avoir un esclave qui accuse moins de retard. En effet, dans le cas du `Log Shipping`, il faut attendre qu'un journal soit complètement rempli avant qu'il ne soit envoyé à l'esclave. Un journal peut contenir plusieurs centaines de transactions, ce qui veut dire qu'en cas de crash du maître, si ce dernier n'a pas eu le temps de transférer le dernier journal, on peut avoir perdu plusieurs centaines de transactions. Le `Streaming Replication` diminue ce retard en envoyant les enregistrements des journaux de transactions par groupe bien inférieur à un journal complet. Il introduit aussi deux processus gérant le transfert entre le maître et l'esclave. Ainsi, en cas de perte du maître, la perte de données est très faible.

Les délais de réplication entre le maître et l'esclave sont très courts. Couplée au `Hot Standby`, cette technologie a rendu obsolète certains systèmes de réplication, utilisés bien souvent avec deux nœuds (un maître et un esclave) : une modification sur le maître sera en effet très rapidement visible sur un esclave, en lecture seule. Néanmoins, cette solution a ses propres inconvénients : réplication de l'instance complète, architecture forcément identique entre les serveurs du cluster, etc.

5.3.3 WARM STANDBY

- Intégré à PostgreSQL depuis 2006
- Serveur de secours en cas de panne
- L'esclave est identique au maitre
  - à quelques transactions près

Le Warm Standby existe depuis la version `8.2`, sortie le 5 décembre 2006. La robustesse de ce mécanisme simple est prouvée depuis longtemps.

Les journaux de transactions (généralement appelés `WAL`, pour *Write Ahead Log*) sont immédiatement envoyés au serveur secondaire après leur écriture. Le serveur secondaire est dans un mode spécial d'attente (le mode de restauration), et lorsqu'un journal de transactions est reçu, il est automatiquement appliqué à l'esclave.

Étant donné que le serveur distant n'applique que les journaux de transactions qu'il reçoit, il y a toujours un risque de pertes de données en cas de panne majeure sur le serveur primaire avant l'envoi du journal de transactions en cours. On peut cependant moduler le risque de trois façons:

- sauf en cas d'avarie très grave sur le serveur primaire, le journal de transactions courant peut être récupéré et appliqué sur le serveur secondaire ;
- on peut réduire la fenêtre temporelle de la réplication en modifiant la valeur de la clé de configuration `archive_timeout`... au delà du délai exprimé avec cette variable de configuration, le serveur change de journal de transactions, provoquant l'archivage du précédent. On peut par exemple envisager un `archive_timeout` à 30 secondes, et ainsi obtenir une réplication à 30 secondes près ;
- on peut utiliser l'outil `pg_receivewal` (apparu en 9.2, et nommé `pg_receivexlog` jusqu'en 9.6) pour créer à distance les journaux de transactions d'après le flux de réplication.

5.3.4 HOT STANDBY

- Évolution du `Warm Standby` apparue en 9.0
- Basé sur le même mécanisme
- Le serveur secondaire est ouvert aux connexions
  - et aux requêtes en lecture seule
- Différentes configurations
  - asynchrone ou synchrone
  - application immédiate ou retardée

Le `Hot Standby` est une évolution du `Warm Standby` en ce sens qu'il comble le gros défaut du `Warm Standby`. Un esclave en `Hot Standby` accepte les connexions des utilisateurs. Il permet aussi d'exécuter des requêtes en lecture seule.

5.3.5 EXEMPLE

Cet exemple montre un serveur maître en Streaming Replication vers un serveur esclave. Ce dernier est en plus en Hot Standby. De ce fait, les utilisateurs peuvent se connecter sur l'esclave pour les requêtes en lecture et sur le maître pour des lectures comme des écritures. Cela permet une forme de répartition de charge sur les lectures, la répartition étant gérée par le serveur d'applications ou par un outil spécialisé.

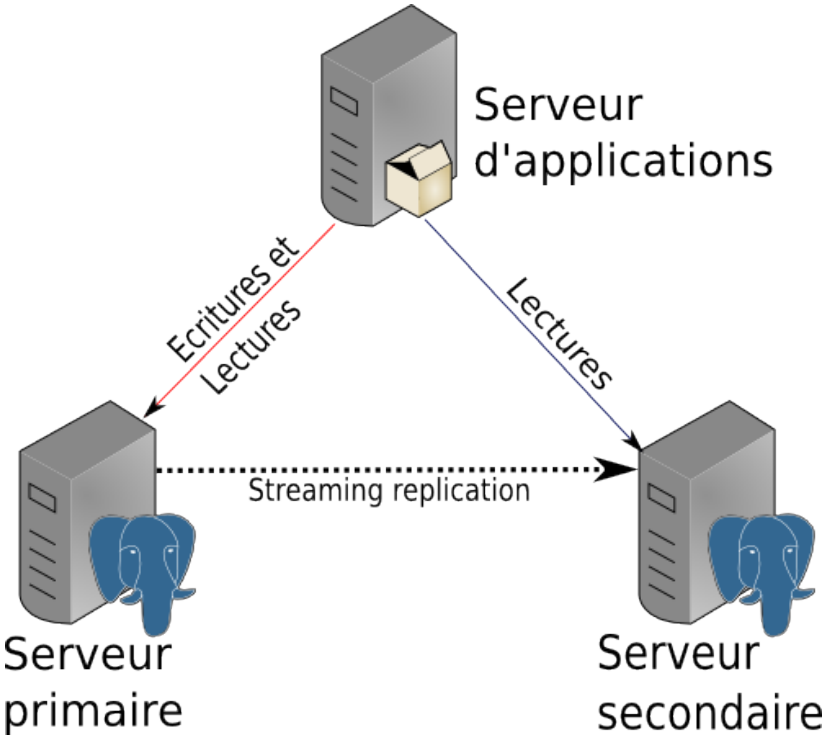
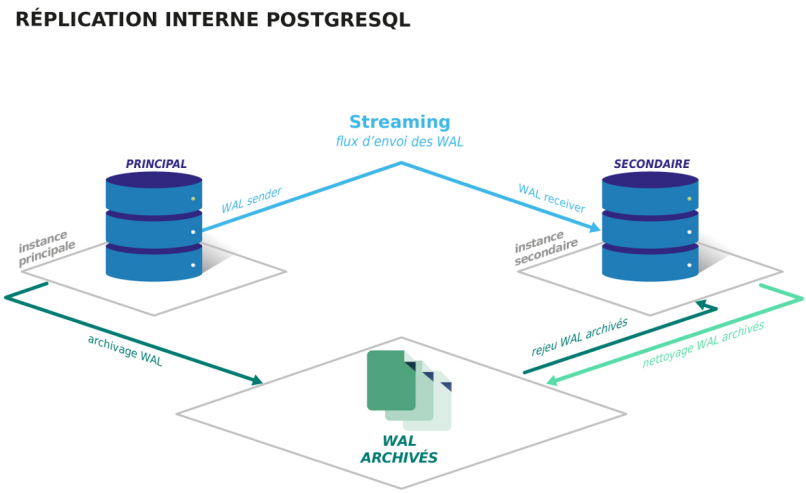
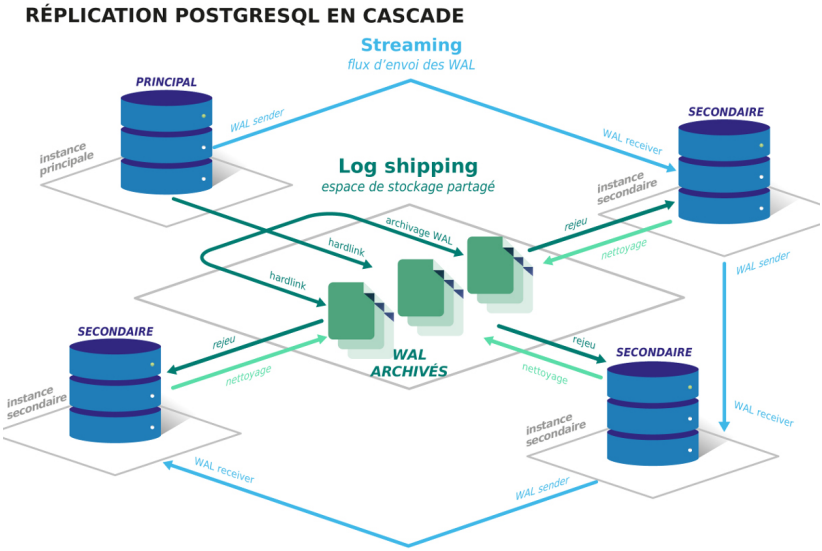


FIGURE 4: EXEMPLE D'ARCHITECTURE

5.3.6 RÉPLICATION INTERNE



5.3.7 RÉPLICATION EN CASCADE



5.4 RÉPLICATION INTERNE LOGIQUE

- Réplique les changements
  - sur une seule base de données
  - d'un ensemble de tables défini
- Uniquement INSERT / UPDATE / DELETE
  - pas les DDL, ni les TRUNCATE

Contrairement à la réplication physique, la réplication logique ne réplique pas les blocs de données. Elle décode le **résultat** des requêtes qui sont transmis au secondaire. Celui-ci applique les modifications SQL issues du flux de réplication logique.

La réplication logique utilise un système de publication/abonnement avec un ou plusieurs abonnés qui s'abonnent à une ou plusieurs publications d'un nœud particulier.

Une publication peut être définie sur n'importe quel serveur primaire de réplication physique. Le nœud sur laquelle la publication est définie est nommé éditeur. Le nœud

où un abonnement a été défini est nommé abonné.

Une publication est un ensemble de modifications générées par une table ou un groupe de table. Chaque publication existe au sein d'une seule base de données.

Un abonnement définit la connexion à une autre base de données et un ensemble de publications (une ou plus) auxquelles l'abonné veut souscrire.

5.4.1 FONCTIONNEMENT

- Création d'une publication sur un serveur
- Souscription d'un autre serveur à cette publication

- Une publication est créée sur le serveur éditeur.
- L'abonné souscrit à cette publication, c'est un « souscripteur ».
- Un processus spécial est lancé : le « bgworker logical replication ». Il va se connecter à un slot de réplication sur le serveur éditeur.
- Le serveur éditeur va procéder à un décodage logique des journaux de transaction pour extraire les résultats des ordres SQL.
- Le flux logique est transmis à l'abonné qui les applique sur les tables.

5.4.2 LIMITATIONS

- Non répliqués :
  - Schémas
  - Séquences
  - *Large objects*
- Pas de publication des tables parentes du partitionnement
- Ne convient pas comme *fail-over*

Le schéma de la base de données ainsi que les commandes DDL ne sont pas répliquées, ci-inclus l'ordre `TRUNCATE`. Le schéma initial peut être créé en utilisant par exemple `pg_dump --schema-only`. Il faudra dès lors répliquer manuellement les changements de structure.

Il n'est pas obligatoire de conserver strictement la même structure des deux côtés. Afin de conserver sa cohérence, la réplication s'arrêtera en cas de conflit.

Il est d'ailleurs nécessaire d'avoir des contraintes de type `PRIMARY KEY` ou `UNIQUE` et `NOT NULL` pour permettre la propagation des ordres `UPDATE` et `DELETE`.

17.12

Les triggers des tables abonnées ne seront pas déclenchés par les modifications reçues via la réplication.

En cas d'utilisation du partitionnement, il n'est pas possible d'ajouter des tables parentes dans la publication.

Les *large objects* ne sont pas répliqués. Les séquences non plus, y compris celles utilisées des colonnes de type `serial`.

De manière générale, il serait possible d'utiliser la réplication logique en vue d'un *fail-over* en propageant manuellement les mises à jour de séquences et de schéma. La réplication physique est cependant plus appropriée pour cela.

La réplication logique vise d'autres objectifs, tels la génération de rapports sur une instance séparée ou la mise à jour de version majeure de PostgreSQL avec une indisponibilité minimale.

5.5 RÉPLICATION EXTERNE

- Un très large choix !
- Quel solution choisir ?
  - pgPool
  - Slony / Bucardo / Londiste
  - Postgres-XC

On dénombre plus de 15 projets de réplication externe pour PostgreSQL. Jusqu' en 2010, PostgreSQL ne disposait pas d'un système de réplication évolué, ce qui explique en partie une telle profusion de solutions. Bien sûr, l'arrivée de la réplication interne physique (avec les technologies `Hot Standby` et `Streaming Replication`) change la donne mais ne remet pas en cause l'existence de tous ces projets. Par contre, l'arrivée de la réplication logique en version 10 risque de les mettre à mal.

Dans cette partie, nous ferons un zoom sur cinq logiciels de réplication :

- pgpool, réplication au niveau SQL
- Slony, réplication asynchrone et asymétrique
- Londiste, réplication asynchrone et asymétrique
- Bucardo, réplication asynchrone et symétrique
- Postgres-XC, réplication synchrone et symétrique

La liste exhaustive est trop longue pour que l'on puisse évoquer en détail chacune des solutions, surtout que certaines sont considérées maintenant comme obsolètes ou tout

du moins non maintenues. Voici les plus connues :

- PGCluster
- Mammoth Replicator
- Daffodil Replicator
- RubyRep
- pg_comparator

L'essentiel est donc de trouver le logiciel adapté à votre besoin !

Plus de détails à cette adresse[68] .

5.5.1 PGPOOL : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Synchrone / Symétrique
- Réplication des requêtes SQL
- Site web[69]

pgPool est un outil libre développé principalement par la société SRA OSS. Il propose un grand nombre de fonctionnalités tournant autour de la haute-disponibilité : pooler de connexions, répartition de charges et réplication. La réplication est en mode synchrone et symétrique. pgPool récupère la requête, la renvoie sur tous les serveurs et attend la réponse de chaque serveur avant de communiquer la réponse au client.

5.5.2 PGPOOL : FONCTIONNALITÉS

- Réplication basée sur les requêtes
- Capable de récupérer certaines fonctions non déterministes

pgPool récupère la requête à exécuter sur chaque serveur. Avant de l' envoyer, il l'analyse pour voir si elle fait appel à des fonctions non déterministes comme celles de récupération de la date et de l'heure : `now()` , `current_timestamp()`, etc. pgPool est capable de faire l'appel à `now()` dans une requête séparée, puis de remplacer l'appel à la fonction par le résultat de la requête séparée. Pour la requête

```
INSERT INTO t1 VALUES (2,now());
```

voici ce qu'il exécute sur le premier serveur :

[68] http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
[69] http://www.pgpool.net

```
LOG:  duration: 0.064 ms  statement: BEGIN
LOG:  duration: 0.143 ms  statement: SELECT now()
LOG:  duration: 0.243 ms  statement: INSERT INTO "t1" VALUES
        (2,"pg_catalog"."timestamptz"('2013-01-21 17:24:37.346359+01'::text))
LOG:  duration: 6.090 ms  statement: COMMIT
```

Il a bien remplacé l' appel à `now()` par le résultat de la précédente requête. Et voici ce qu'il exécute sur le deuxième serveur :

```
LOG:  duration: 0.037 ms  statement: BEGIN
LOG:  duration: 0.215 ms  statement: INSERT INTO "t1" VALUES
        (2,"pg_catalog"."timestamptz"('2013-01-21 17:24:37.346359+01'::text))
LOG:  duration: 161.343 ms  statement: COMMIT
```

Cependant, ce n'est fonctionnel que pour certaines fonctions que pgPool connaît, pas pour les autres (pas pour `random()` par exemple). De plus, si la fonction `now()` est la valeur par défaut de la colonne, pgPool ne sait pas le gérer. Il sera possible de voir une légère différence entre les différents serveurs :

```
-- sur le premier serveur
  2 | 2013-01-21 17:26:59.814946+01
-- sur le deuxième serveur
  2 | 2013-01-21 17:26:59.815214+01
```

Même si la différence est minime, il s'agissait d'un cas simple où les deux serveurs PostgreSQL sont sur la même machine. Avec une machine séparée, la différence peut être très importante, surtout si les horloges des deux machines ne sont pas synchronisées.

5.5.3 PGPOOL : TECHNIQUE

- pgPool est à l'origine un *pooler* de connexions
- Configuration propre
- Plus ou moins transparent pour les applications
- Réplication de toutes les requêtes
  - y compris le `DDL`

Au départ, pgPool n'était qu'un pooler de connexions. Il a ensuite évolué pour intégrer d'autres fonctionnalités, comme le répartiteur de charges, la réplication, le cache de requête, le partitionnement, un peu de pacemaker.

C'est un logiciel supplémentaire qui se fait passer pour un serveur PostgreSQL. Il est très simple à mettre en place, sa configuration est plutôt aisée et il est plus ou moins transpar-

ent pour les applications. Cependant, il faut vous attendre à devoir modifier vos applications suivant les fonctionnalités demandés.

Contrairement aux autres outils de réplication (notamment ceux par trigger), il réplique directement toutes les requêtes, y compris les DDL.

5.5.4 PGPOOL : LIMITES

- pgPool est un *SPOF*
  - SPOF => Single Point Of Failure
- Réplication basée sur la réplication des requêtes `SQL`
  - obligation de passer par pgpool
- Prise en main et fonctionnement complexe
- Authentification en mode réplication
  - pas de `md5`
  - pas de méthodes externes (LDAP, Radius, etc.)
- Effet « couteau suisse »

pgPool est le serveur sur lequel toutes les applications vont se connecter à PostgreSQL. S'il tombe, les bases de données ne sont plus accessibles. Il faudra donc veiller à ce qu'un autre service pgpool-II existe sur une autre machine et à mettre en place un système de bascule automatique. Cela est généralement fait avec des infrastructures redondantes basées sur `heartbeat` /`pacemaker`, `lvm`, etc.

De plus, il faut veiller à ce que les bases de données ne puissent être accédées que via pgPool. Si jamais un utilisateur peut se connecter sur une base sans passer par pgPool, il pourrait y exécuter des requêtes en écriture qui, de ce fait, ne seront exécutées que sur ce serveur. Cela provoquerait une désynchronisation des données entre les différents serveurs.

Comme pgPool agit en proxy, il est impossible d'utiliser une authentification `md5` ou des authentifications basées sur des méthodes externes comme LDAP, Radius, etc. De même, la gestion des certificats peut poser de gros soucis.

Pgpool semblait prometteur mais certaines fonctions (notamment le load-balancing et la réplication) se révèlent souvent complexes à mettre en œuvre, difficile à stabiliser et limitées à des cas d'utilisation très spécifiques. Malgré son ancienneté il y a encore beaucoup de corrections de bugs à chaque mise à jour. Sa maîtrise demande du temps et nécessite d' écrire les scripts de gestion de Haute Disponibilité.

Dans la communauté PostgreSQL, les critiques sont récurrentes à l'encontre de pgPool

: beaucoup lui reprochent de tout faire un peu. pgPool est en effet à la fois un pooler de connexions, un outil de réplication, un outil d'exécution de requêtes en parallèle, un outil de répartition de charge, un simili-pacemaker, etc. Ces critiques sont plutôt fondées. Ceux qui ne s'intéressent qu'au mode de pooling peuvent toujours opter pour l'excellent PgBouncer de Skype.

5.5.5 PGPOOL : UTILISATIONS

- Base de données de secours
- Répartition de charge en lecture

L'utilisation principale de pgPool actuellement réside en sa fonctionnalité de répartition de charge. Son mode réplication est très souvent déconseillé, de meilleures implémentations de la réplication existant ailleurs.

Vous trouverez plus d'informations dans cet article[70] .

5.5.6 SLONY : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Asymétrique
- Diffusion des résultats (triggers)
- Site web[71]

Slony est un très ancien projet libre de réplication pour PostgreSQL. C'était l'outil de choix avant l'arrivée de la réplication en interne dans PostgreSQL.

5.5.7 SLONY : FONCTIONNALITÉS

- Réplication de tables sélectionnées
- Procédures de bascule
  - switchover / switchback
  - failover / failback

[70] http://www.dalibo.org/hs44_pgpoolii_la_replication_par_duplication_des_requetes
[71] http://slony.info/

Contrairement à la réplication interne de PostgreSQL qui réplique forcément tout (ce qui est un avantage et un inconvénient), Slony vous laisse choisir les tables que vous voulez répliquer. Cela a pour conséquence que, si vous ajoutez une table, il faudra en plus dire à Slony s'il est nécessaire ou non de la répliquer.

Les procédures de bascule chez Slony sont très simples. Il est ainsi possible de basculer un maître et son esclave autant de fois qu'on le souhaite, très rapidement, sans avoir à reconstruire quoi que ce soit.

5.5.8 SLONY : TECHNIQUE

- Réplication basée sur des triggers
- Démons externes, écrits en C
- Le maître est un **provider**
- Les esclaves sont des **subscribers**

Slony est un système de réplication asynchrone/asymétrique, donc un seul maître et un ou plusieurs esclaves mis à jour à intervalle régulier. La récupération des données modifiées se fait par des triggers, qui stockent les modifications dans les tables systèmes de Slony avant leur transfert vers les esclaves. Un système de démon récupère les données pour les envoyer sur les esclaves et les applique.

Les démons et les triggers sont écrits en C, ce qui permet à Slony d'être très performant.

Au niveau du vocabulaire utilisé, le maître est souvent appelé un « provider » (il fournit les données aux esclaves) et les esclaves sont souvent des « subscribers » (ils s'abonnent au flux de réplication pour récupérer les données modifiées).

5.5.9 SLONY : POINTS FORTS

- Choix des tables à répliquer
- Indépendance des versions de PostgreSQL
- Technique de propagation des *DDL*
- Robustesse

Slony dispose de nombreux points forts qui font défaut à la réplication interne de PostgreSQL.

Il permet de ne répliquer qu'un sous-ensemble des objets d'une instance : pas forcément toutes les bases, pas forcément toutes les tables d'une base particulière, etc.

17.12

Le maître et les esclaves n'ont pas besoin d'utiliser la même version majeure de PostgreSQL. Il est donc possible de mettre à jour en plusieurs étapes (plutôt que tous les serveurs à la fois). Cela facilite aussi le passage à une version majeure ultérieure.

Même si la réplication des DDL est impossible, leur envoi aux différents serveurs est possible grâce à un outil fourni. Tous les systèmes de réplication par triggers ne peuvent pas en dire autant.

5.5.10 SLONY : LIMITES

- Le réseau doit être fiable : peu de *lag*, pas ou peu de coupures
- Supervision délicate
- Modifications de schémas complexes

Slony peut survivre avec un réseau coupé. Cependant, il n'aime pas quand le réseau passe son temps à être disponible puis indisponible. Les démons slon ont tendance à croire qu'ils sont toujours connectés alors que ce n'est plus le cas.

Superviser Slony n'est possible que via une table statistique appelée `sl_status`. Elle fournit principalement deux informations : le retard en nombre d'événements de synchronisation et la date de la dernière synchronisation.

Enfin, la modification de la structure d'une base, même si elle est simplifiée avec le script fourni, n'est pas simple, en tout cas beaucoup moins simple que d 'exécuter une requête DDL seule.

5.5.11 SLONY : UTILISATIONS

- Clusters en cascade
- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc)

La réplication proposée par Slony est surtout intéressante pour les besoins complexes, comme mettre des serveurs en cascade (quoique PostgreSQL est capable de le faire depuis la version 9.2).

Pour avoir un esclave permettant la création de rapports, Slony peut se révéler plus intéressant car il est possible d'avoir des tables de travail en écriture sur l'esclave avec Slony. Il est aussi possible d'ajouter des index sur l'esclave qui ne seront pas présents sur le maître

(on évite donc la charge de maintenance des index par le maître, tout en permettant de bonnes performances pour la création des rapports).

Pour plus d'informations sur Slony, n'hésitez pas à lire un de nos articles disponibles sur notre site[72] . Le thème des réplications complexes a aussi été abordé lors d'un PostgreSQL Sessions[73] .

5.5.12 LONDISTE : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Asymétrique
- Réplication des résultats
- Site web[74]

Londiste est un projet libre, conçu et développé par la société Skype. Il s 'agit là-aussi de réplication asynchrone/asymétrique, par trigger, tout comme Slony.

5.5.13 LONDISTE : FONCTIONNALITÉS

- Réplication de tables sélectionnées
- Procédures de bascule
  - switchover / switchback
  - failover / failback

Tout comme Slony, Londiste offre davantage de finesse concernant les tables à répliquer. De la même manière, il faut explicitement indiquer les tables à répliquer.

Les procédures de bascule sont tout aussi simples que celles de Slony. Il est ainsi possible de basculer un maître et son esclave autant de fois qu'on le souhaite, très rapidement, sans avoir à reconstruire quoi que ce soit.

5.5.14 LONDISTE : TECHNIQUE

- Réplication basée sur des triggers

[72] http://www.dalibo.org/hs44_slony_la_replication_des_donnees_par_trigger
[73] http://www.dalibo.org/replications_sophistiquees_avec_slony
[74] http://skytools.projects.pgfoundry.org/skytools-3.0/doc/londiste3.html

- Démons externes, écrits en Python
- Utilise un autre outil provenant des *Skytools* : PgQ
- 1 maître / N esclaves

Londiste est un outil qui ressemble beaucoup à Slony dans ces caractéristiques. La différence principale tient dans le fait qu'il ne gère pas de groupes de tables. C'est donc aussi de la réplication par triggers (attention, pas de triggers sur l'esclave qui empêchent les modifications sur les tables répliquées). Les triggers récupèrent les modifications et les stockent dans des tables systèmes. Un démon récupère les données dans les tables systèmes pour les envoyer et les écrire sur l'esclave. Pour une table, il n'y a qu'un maître et un ou plusieurs esclaves.

5.5.15 LONDISTE : POINTS FORTS

- PgQ est robuste, fiable et flexible
- Pas de groupes de réplication, mais des tables appartenant à différentes *queues*

On peut ainsi avoir des tables dans le serveur "maître" qui alimentent la *queue* principale à laquelle les différents "esclaves" auront souscrit, mais aussi d'autres *queues* qui vont alimenter certaines autres tables du "maître".

Cela rend donc possible la mise en place de réplications "croisées".

5.5.16 LONDISTE : LIMITES

- Technique de propagation des *DDL* basique
- Très peu de fonctionnalités.

Pour la réplication des requêtes DDL, c'est encore plus basique que Slony. Il faut soi-même envoyer les requêtes DDL sur les différents serveurs du cluster d'instances. Néanmoins depuis la version 3 de Londiste, il existe une commande permettant d'exécuter un script SQL sur un ou plusieurs nœuds.

Londiste est aussi plus simple à mettre en place et à maîtriser, tout simplement parce qu'il propose moins de fonctionnalités que Slony.

5.5.17 LONDISTE : UTILISATIONS

- Clusters en cascade
- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc)

L'utilisation de Londiste porte sur les mêmes cas que Slony. Le choix entre les deux se fera plutôt sur un avis personnel.

Pour plus d'informations sur Londiste, voir cet article[75] .

5.5.18 BUCARDO : CARTE D'IDENTITÉ

- Projet libre (BSD)
- Asynchrone / Symétrique ...
- ... ou Asynchrone / Asymétrique
- Réplication des résultats (dits *deltas*)
- Site web[76]

Ce projet libre a été créé par la société End Point Corporation pour le besoin d'un client particulier. Ils ont fini par décider de libérer le code qu 'ils continuent à maintenir.

C'est un des rares outils à proposer du multi-maître. Jusqu'à la version 5, seuls deux nœuds peuvent être en maître. À partir de la version 5, il sera possible d'avoir plus de deux serveurs maîtres. Il fonctionne aussi en utilisant des triggers mais sa mise en place est vraiment différente par rapport à Slony et Londiste.

5.5.19 BUCARDO : FONCTIONNALITÉS

- Réplication maître-maitre ou maître-esclave
- Répartition des écritures
- Failover manuel
- Plusieurs méthodes de résolution des conflits

Bucardo offre plusieurs types de réplication. « fullcopy » et « pushdelta » permettent de répliquer les données en mode maître-esclave, « swap » est le mode de synchronisation multi-maître.

[75] http://www.dalibo.org/hs44_londiste_la_replicaton_vue_par_skype
[76] http://bucardo.org/

17.12

Bucardo ne s'attèle qu'à la réplication des données, c'est à l' administrateur de réagir en cas de panne et de réaliser les opérations de bascule et de remise en réplication.

5.5.20 BUCARDO : TECHNIQUE

- Réplication basée sur des triggers
- Démons externes, écrits en Perl
- Maître / maître (1 seul couple)
  - ou maître / esclave(s)

En mode maître-esclave, le principe de fonctionnement de Bucardo est très proche de Slony : des jeux de réplication appelés « sync » utilisent des triggers, un service en Perl se charge alors de propager les modifications.

En mode maître-maître, le type de réplication « sync » indique au service en Perl de réaliser la réplication dans les deux sens avec la gestion des conflits.

5.5.21 BUCARDO : POINTS FORTS

- Basé sur un(des) set(s) de réplication et non sur un(des) schéma(s)
- Simplicité d'utilisation
- Résolution standard des conflits

Bucardo introduit les concepts suivants pour la résolution des conflits en réplication multi-maître :

- `source` : la base de données d'origine gagne toujours
- `target` : la base de destination gagne toujours
- `random` : l'une des deux bases est choisie au hasard comme étant la gagnante
- `latest` : la ligne modifiée le plus récemment gagne
- `abort` : la réplication est arrêtée
- `skip` : aucune décision ni action n'est prise

Il est également possible de créer son propre gestionnaire de résolution de conflit personnalisé.

5.5.22 BUCARDO : LIMITES

- Aucune technique de propagation des *DDL*
  - (en cours de développement)
- Limité à deux nœuds en mode multi-maîtres
- Le réseau doit être fiable
  - peu de retard, pas ou peu de coupures
- Sous Linux/Unix uniquement
- Un seul développeur sur le projet
- Manque de fiabilité de l'outil

Le projet est porté par pratiquement un seul développeur, Greg Sabino Mulane, développeur très connu (et apprécié) dans la communauté PostgreSQL. Cela explique un développement en dent de scie, même si la correction des bugs est généralement très rapide.

La propagation des DDL n'est pas prise en compte. Il faut donc, comme pour Slony, exécuter les DDL sur chaque serveur séparément.

5.5.23 BUCARDO : UTILISATIONS

- Cluster maître/maître simple
- Base de données de secours

La mise en place de Bucardo est intéressante pratiquement uniquement quand on veut mettre en place un cluster maître/maître. En dehors de cela, il est préférable de se baser sur des solutions comme Slony ou Bucardo.

5.5.24 POSTGRES-XC : CARTE D'IDENTITÉ

- Projet libre (Licence PostgreSQL)
- Réplication multi-maîtres synchrone
- Réplication des résultats
- Ancien site web[77]
- Projet renommé en postgres-x2[78]

[77] http://postgres-xc.sourceforge.net
[78] https://github.com/postgres-x2/postgres-x2

17.12

Postgres-XC est une solution de réplication multi-maîtres synchrones et de répartition des données entre plusieurs serveurs de données. Le projet a été initié en 2010 par NTT en libérant un projet de recherche interne. EnterpriseDB s'ajoute alors à NTT pour soutenir le développement de Postgres-XC. La version 1.0, basée sur PostgreSQL 9.1, est sortie en juin 2012. La version 1.1 est basée sur PostgreSQL 9.2, tandis que la version 1.2 est basée sur PostgreSQL 9.3. Une nouvelle version basée sur PostgreSQL 9.4 est prévue pour suivre l'arrivée à l'état stable de cette version majeure du projet.

En 2015, lors du passage du projet sur GitHub, le projet a été renommé en `postgres-x2`.

5.5.25 POSTGRES-XC : FONCTIONNALITÉS

- Réplication multi-maîtres synchrone
- Distribution des données entre serveurs
  - partitionnement horizontal
- Répartition de charge
- Haute-disponibilité

Le principal intérêt de Postgres-XC est sa capacité à gérer plusieurs serveurs, tous en maître. En ce sens, il rejoint Bucardo, sauf qu'il est capable de prendre en compte un plus grand nombre de serveurs.

Il dispose de requêtes SQL permettant de distribuer les données entre plusieurs nœuds, ce qui permet un partitionnement horizontal facilité.

5.5.26 POSTGRES-XC : TECHNIQUE

- *Global Transaction Manager*
  - gestion des XID et conflits
- *Coordinator*
  - Catalogue / Répartiteur
- *Datanode*
  - Stockage

L'architecture de Postgres-XC est divisé en trois éléments.

En premier lieu, le *Global Transaction Manager* (GTM) est le service de gestion global des transactions. Il gère les conflits et fournit les identifiants de transactions.

Ensuite, le *Coordinator* fournit le point d'entrée pour les sessions clientes. Il équilibre la charge entre les *datanodes*, agrège les données réparties au besoin, et communique avec les autres *coordinators* par l'intermédiaire du GTM, il conserve le catalogue du cluster Postgres-XC.

Enfin, un *datanode* est un serveur PostgreSQL modifié qui exécute les requêtes fournies par les *coordinators* et stocke la totalité ou une partie des données selon la configuration.

5.5.27 POSTGRES-XC : POINTS FORTS

- Installation relativement aisée et bien documentée
- Intégration au cœur de PostgreSQL
- Ajout d'ordres DDL spécifiques

L'intégration au sein même du code de PostgreSQL est garant de performances optimales. Cependant, c'est aussi à double tranchant, comme cela sera exposé plus tard.

Les fonctionnalités de réplication de PostgreSQL sont, par exemple, mises à profit pour rendre une plate-forme distribuée (partitionnement horizontal) hautement-disponible.

Les tests et benchmarks produits au moment de la sortie de la version 1.0 indique une bonne tenue en charge jusqu'à 10 nœuds. La limite théorique est placée à 20 nœuds par le projet.

Le choix de la réplication ou du partitionnement se fait au niveau des ordres SQL de type DDL, ainsi une seule API permet de gérer l'organisation des données.

5.5.28 POSTGRES-XC : LIMITES

- Version dérivée de PostgreSQL
  - basée sur la version 9.3
- Chaque ordre SQL doit être porté
- Fonctionnalités manquantes
  - triggers, SQL/MED, savepoints, SSI
  - contraintes globales, détection globale des deadlock

L'inconvénient majeur de créer une version dérivée de PostgreSQL (fork) concerne le suivi des modifications du code initial : en plus de fournir ses fonctionnalités propres, Postgres-XC doit intégrer les nouvelles fonctionnalités introduites dans PostgreSQL.

17.12

Toutes les fonctionnalités de PostgreSQL n'ont pas encore été portées dans Postgres-XC, comme le support des extensions ou encore les triggers. Il est donc indispensable de valider les fonctionnalités disponibles par la lecture des notes de version et des maquettes de test avant d'adopter la solution.

Il faut donc être conscient que PostgreSQL et Postgres-XC sont deux projets distincts, même s'ils sont très liés. Autrement dit, une application fonctionnant sur PostgreSQL ne fonctionnera pas forcément sur Postgres-XC.

5.5.29 POSTGRES-XC : UTILISATIONS

- Très grandes bases transactionnelles sous forte charge
- Bases spécialisées
- Peu de cas d'utilisation connus en production

Postgres-XC est un projet prometteur, la documentation est fournie et à jour, une roadmap est disponible. Il est surtout soutenu par deux acteurs incontournables de l'écosystème PostgreSQL.

Il s'agit quand même d'un projet jeune qu'il est intéressant de laisser mûrir.

5.5.30 SOLUTIONS OBSOLÈTES

- Postgres-R
- PGCluster
- replicator
- cybercluster

Comme beaucoup de logiciels, certains deviennent obsolètes pour diverses raisons. La maintenance d'un logiciel est une tâche qui représente un coup humain non négligeable.

Parmi les solutions de réplication externe pour PostgreSQL, certains projets ne sont plus développés ou maintenus faute de main d'œuvre, comme Postgres-R. D' autres ont perdu leur intérêt suite à l'arrivée de la réplication interne (Hot-Standby et Streaming Replication) en version 9.0.

5.6 RÉPLICATION BAS NIVEAU

- RAID
- DRBD
- SAN Mirroring
- À prendre évidemment en compte...

Cette présentation est destinée à détailler les solutions logicielles de réplication pour PostgreSQL, uniquement. On peut tout de même évoquer les solutions de réplication de "bas niveaux", voire matérielles.

De nombreuses techniques matérielles viennent en complément essentiel des technologies de réplication utilisées dans la haute disponibilité. Leur utilisation est généralement obligatoire, du `RAID` en passant par les `SAN` et autres techniques pour redonder l'alimentation, la mémoire, les processeurs, etc...

5.6.1 RAID

- Obligatoire
- Fiabilité d'un serveur.
- RAID 1 ou RAID 10
- Lectures plus rapides
  - dépend du nombre de disques impliqués

Un système RAID1 ou RAID10 permet d'écrire les mêmes données sur plusieurs disques en même temps. Si un disque meurt, il est possible d'utiliser l'autre disque pour continuer. C'est de la réplication bas-niveau. Le disque défectueux peut être remplacé sans interruption de service. Ce n'est pas une réplication entre serveur mais cela contribue à la haute-disponibilité du système.

Le système RAID10 est plus intéressant pour les fichiers de données alors qu' un système RAID1 est suffisant pour les journaux de transactions.

5.6.2 DRBD

- Simple / synchrone / Bien documenté
- Lent / Esclave inaccessible / Linux uniquement

17.12

DRBD est un outil capable de répliquer le contenu d'un périphérique blocs. En ce sens, ce n'est pas un outil spécialisé pour PostgreSQL contrairement aux autres outils vus dans ce module. Il peut très bien servir à répliquer des serveurs de fichiers ou de mails. Il réplique les données en temps réel et de façon transparente, pendant que les applications modifient leur fichiers sur un périphérique. Il peut fonctionner de façon synchrone ou asynchrone. Tout ça en fait donc un outil intéressant pour répliquer le répertoire des données de PostgreSQL.

Pour plus de détails, consulter cet article[79] .

DRBD est un système simple à mettre en place. Son gros avantage est la possibilité d' avoir une réplication synchrone, son inconvénient direct est sa lenteur et la non-disponibilité des esclaves.

5.6.3 SAN MIRRORING

- Comparable à DRBD
- Solution intégrée
- Manque de transparence

La plupart des constructeurs de baie de stockage propose des systèmes de réplication automatisé avec des mécanismes de failover / failback parfois sophistiqués. Ces solutions présentent généralement les mêmes caractéristiques que DRBD. Ces technologies ont en revanche le défaut d'être opaques et de nécessiter une main d'œuvre hautement qualifiée.

5.7 CONCLUSION

Points essentiels :

- bien définir son besoin
- identifier tous les *SPOF*
- superviser son *cluster*
- tester régulièrement les procédures de *Failover* (Loi de Murphy...)

---

[79] http://www.dalibo.org/hs45_drbd_la_replication_des_blocs_disques

5.7.1 BIBLIOGRAPHIE

- Doc officielle : Chapitre 25
- Hors série « Haute-disponibilité avec PostgreSQL » dans GNU/Linux France Magazine
- Article de blog « Hot Standby : Un exemple concret »
- « 25. Haute disponibilité, répartition de charge et réplication[80] ». PGDG, 2010
- « Haute-disponibilité avec PostgreSQL[81] ». Guillaume Lelarge, 2009
- « Hot Standby : Un exemple concret[82] ». Damien Clochard, 2010

Iconographie :

- La photo initiale[83] est sous licence CC-BY.

5.7.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[80] http://docs.postgresql.fr/9.0/high-availability.html
[81] http://www.dalibo.org/haute_disponibilite_avec_postgresql
[82] http://blog.taadeem.net/index.php?post/2010/07/19/Hot-Standby-:-un-exemple-concret
[83] http://www.flickr.com/photos/epsos/3574411866/