Module DE

Outils graphiques et console

25.03

Table des matières

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

1/ Outils graphiques et console **5**
1.1 Préambule . . . 6
1.1.1 Plan . . . 6
1.2 Outils console de PostgreSQL . . . 7
1.2.1 Outils : Gestion des bases . . . 7
1.2.2 Outils : Sauvegarde / Restauration . . . 8
1.2.3 Outils : Maintenance . . . 9
1.2.4 Outils : Maintenance de l'instance . . . 9
1.2.5 Autres outils en ligne de commande . . . 10
1.3 Chaînes de connexion . . . 11
1.3.1 Paramètres . . . 11
1.3.2 Autres variables d'environnement . . . 14
1.3.3 Chaînes libpq clés/valeur . . . 14
1.3.4 Chaînes URI . . . 15
1.3.5 Connexion avec choix automatique du serveur . . . 16
1.3.6 Authentification d'un client (outils console) . . . 17
1.4 La console psql . . . 19
1.4.1 Obtenir de l'aide et quitter . . . 19
1.4.2 Gestion de la connexion . . . 20
1.4.3 Catalogue système : objets utilisateurs . . . 22
1.4.4 Catalogue système : rôles et accès . . . 25
1.4.5 Visualiser le code des objets . . . 27
1.4.6 Configuration . . . 28
1.4.7 Exécuter des requêtes . . . 29
1.4.8 Afficher le résultat d'une requête . . . 31
1.4.9 Afficher les détails d'une requête . . . 33
1.4.10 Exécuter le résultat d'une requête . . . 33
1.4.11 Manipuler le tampon de requêtes . . . 35
1.4.12 Entrées/sorties . . . 36
1.4.13 Gestion de l'environnement système . . . 37
1.4.14 Variables internes psql . . . 38
1.4.15 Variables utilisateur psql . . . 40
1.4.16 Tests conditionnels . . . 41

1.4.17 Personnaliser psql . . . 42
1.5 Écriture de scripts shell . . . 45
1.5.1 Exécuter un script SQL avec psql . . . 45
1.5.2 Gestion des transactions . . . 46
1.5.3 Écrire un script SQL . . . 47
1.5.4 Les blocs anonymes . . . 48
1.5.5 Utiliser des variables . . . 49
1.5.6 Gestion des erreurs . . . 50
1.5.7 Formatage des résultats . . . 51
1.5.8 Résultats en pivot (tableau croisé) . . . 52
1.5.9 Formatage dans les scripts SQL . . . 53
1.5.10 Scripts & Crontab . . . 54
1.5.11 Exemple de script de sauvegarde . . . 54
1.6 Outils graphiques . . . 56
1.6.1 temBoard . . . 56
1.6.2 temBoard - PostgreSQL Remote Control . . . 56
1.6.3 temBoard - Vue parc . . . 57
1.6.4 temBoard - Tableau de bord . . . 58
1.6.5 temBoard - Activity . . . 59
1.6.6 temBoard - Supervision . . . 60
1.6.7 temBoard - Configuration . . . 61
1.6.8 temBoard - Maintenance . . . 62
1.6.9 pgAdmin 4 . . . 63
1.6.10 pgAdmin 4 : tableau de bord . . . 64
1.6.11 DBeaver . . . 65
1.6.12 DBeaver : fenêtre principale . . . 66
1.6.13 phpPgAdmin . . . 67
1.6.14 phpPgAdmin : fonctionnalités . . . 67
1.6.15 AdminerEvo . . . 68
1.6.16 AdminerEvo : fonctionnalités . . . 68
1.6.17 pgModeler . . . 69
1.6.18 pgModeler . . . 70
1.7 Conclusion . . . 71
1.7.1 Questions . . . 71
1.8 Quiz . . . 72
1.9 Travaux pratiques . . . 73
1.10 Travaux pratiques (solutions) . . . 76

Les formations Dalibo **85**
Cursus des formations . . . 85
Les livres blancs . . . 86
Téléchargement gratuit . . . 86

Sur ce document

Formation	Module DE
Titre	Outils graphiques et console
Révision	25.03
PDF	https://dali.bo/de_pdf
EPUB	https://dali.bo/de_epub
HTML	https://dali.bo/de_html
Slides	https://dali.bo/de_slides
TP	https://dali.bo/de_tp
TP (solutions)	https://dali.bo/de_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

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

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

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

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

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

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

Marques déposées

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

Versions de PostgreSQL couvertes

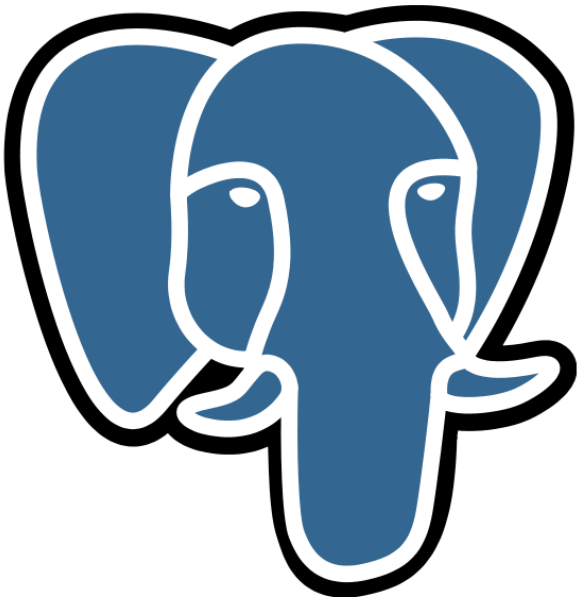
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ Outils graphiques et console



1.1 PRÉAMBULE

Les outils graphiques et console :

- les outils graphiques d'administration
- la console
- les outils de contrôle de l'activité
- les outils DDL
- les outils de maintenance

Ce module nous permet d'approcher le travail au quotidien du DBA et de l'utilisateur de la base de données. Nous verrons les différents outils disponibles, en premier lieu la console texte, `psql`.

1.1.1 Plan

- Outils en ligne de commande de PostgreSQL
- Réaliser des scripts
- Outils graphiques

1.2 OUTILS CONSOLE DE POSTGRESQL

- Plusieurs outils PostgreSQL en ligne de commande existent
  - une console interactive
  - des outils de maintenance
  - des outils de sauvegardes/restauration
  - des outils de gestion des bases

Les outils console de PostgreSQL que nous allons voir sont fournis avec la distribution de PostgreSQL. Ils permettent de tout faire : exécuter des requêtes manuelles, maintenir l'instance, sauvegarder et restaurer les bases.

1.2.1 Outils : Gestion des bases

- `createdb` : ajouter une nouvelle base de données
- `createuser` : ajouter un nouveau compte utilisateur
- `dropdb` : supprimer une base de données
- `dropuser` : supprimer un compte utilisateur

Chacune de ces commandes est un « alias », un raccourci qui permet d'exécuter certaines commandes SQL directement depuis le shell sans se connecter explicitement au serveur. L'option `--echo` permet de voir les ordres SQL envoyés.

Par exemple, la commande système `dropdb` est équivalente à la commande SQL `DROP DATABASE`. L'outil `dropdb` se connecte à la base de données nommée `postgres` et exécute l'ordre SQL et enfin se déconnecte.

La création d'une base se fait en utilisant l'outil `createdb` et en lui indiquant le nom de la nouvelle base, de préférence avec un utilisateur dédié. `createuser` crée ce que l'on appelle un « rôle », et appelle `CREATE ROLE` en SQL. Nous verrons plus loin les droits de connexion, de superutilisateur, etc.

Une création de base depuis le shell peut donc ressembler à ceci :

```
$ createdb --echo --owner erpuser  erp_prod
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE erp_prod OWNER erpuser;
```

Alors qu'une création de rôle peut ressembler à ceci :

```
$ createuser --echo --login --no-superuser erpuser
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE erpuser NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Et si le `pg_hba.conf` le permet :

```
$ psql -U erpuser erp_prod  < script_installation.sql
```

1.2.2 Outils : Sauvegarde / Restauration

- Sauvegarde logique, pour une instance
  - `pg_dumpall` : sauvegarder l'instance PostgreSQL
- Sauvegarde logique, pour une base de données
  - `pg_dump` : sauvegarder une base de données
  - `pg_restore` : restaurer une base de données PostgreSQL
- Sauvegarde physique :
  - `pg_basebackup`
  - avec : `pg_verifybackup` et `pg_combinebackup` (v17)

Ces commandes sont essentielles pour assurer la sécurité des données du serveur.

Comme son nom l'indique, `pg_dumpall` sauvegarde l'instance complète, autrement dit toutes les bases mais aussi les objets globaux. À partir de la version 12, il est cependant possible d'exclure une ou plusieurs bases de cette sauvegarde.

Pour ne sauvegarder qu'une seule base, il est préférable de passer par l'outil `pg_dump`, qui possède plus d'options. Il faut évidemment lui fournir le nom de la base à sauvegarder. Pour sauvegarder notre base b1, il suffit de lancer la commande suivante :

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

Pour la restauration d'une sauvegarde, l'outil habituel est `pg_restore`. `psql` est utilisé pour la restauration d'une sauvegarde faite en mode texte (script SQL).

Ces deux outils réalisent des sauvegardes logiques, donc au niveau des objets logiques (tables, index, etc).

La sauvegarde physique (donc au niveau des fichiers) à chaud est possible avec `pg_basebackup`, qui copie un serveur en fonctionnement, journaux de transaction inclus. Son fonctionnement est nettement plus complexe qu'un simple `pg_dump`. `pg_basebackup` est utilisé par les outils de sauvegarde

PITR, et pour créer des serveurs secondaires. `pg_verifybackup` permet de vérifier une sauvegarde réalisée avec `pg_basebackup`. À partir de PostgreSQL 17, `pg_basebackup` sait effectuer des sauvegardes incrémentales, à recombiner avec `pg_combinebackup`.

1.2.3 Outils : Maintenance

- Maintenance des bases
  - `vacuumdb` : récupérer l'espace inutilisé, statistiques
  - `clusterdb` : réorganiser une table en fonction d'un index
  - `reindexdb` : réindexer

`reindexdb`, là encore, est un alias lançant des ordres `REINDEX`. Une réindexation périodique des index peut être utile. Par exemple, pour lancer une réindexation de la base **b1** en affichant la commande exécutée :

```
$ reindexdb --echo --concurrently b1
SELECT pg_catalog.set_config('search_path', '', false);
REINDEX DATABASE CONCURRENTLY b1;
WARNING:  cannot reindex system catalogs concurrently, skipping all
```

`vacuumdb` permet d'exécuter les différentes variantes du `VACUUM` (`FULL`, `ANALYZE`, `FREEZE`…) depuis le shell, principalement le nettoyage des lignes mortes, la mise à jour des statistiques sur les données, et la reconstruction de tables. L'usage est ponctuel, le démon autovacuum s'occupant de cela en temps normal.

`clusterdb` lance un ordre `CLUSTER`, soit une reconstruction de la table avec tri selon un index. L'usage est très spécifique.

Rappelons que ces opérations posent des verrous qui peuvent être très gênants sur une base active.

1.2.4 Outils : Maintenance de l'instance

- `initdb` : création d'instance
- `pg_ctl` : lancer, arrêter, relancer, promouvoir l'instance
- `pg_upgrade` : migrations majeures
- `pg_config`, `pg_controldata` : configuration

Ces outils sont rarement utilisés directement, car on passe généralement par les outils du système d'exploitation et ceux fournis par les paquets, qui les utilisent. Ils peuvent toutefois servir et il faut les connaître.

`initdb` crée une instance, c'est-à-dire crée tous les fichiers nécessaires dans le répertoire indiqué (`PGDATA`). Les options permettent d'affecter certains paramètres par défaut. La plus importante (car on ne peut corriger plus tard qu'à condition que l'instance soit arrêtée, donc en arrêt de production) est l'option `--data-checksums` activant les sommes de contrôle, dont l'activation est généralement conseillée.

`pg_ctl` est généralement utilisé pour démarrer/arrêter une instance, pour recharger les fichiers de configuration après modification, ou pour promouvoir une instance secondaire en primaire. Toutes les actions possibles sont documentées ici[1].

`pg_upgrade` est utilisée pour convertir une instance existante lors d'une migration entre versions majeures. La version minimale supportée est la 8.4, sauf à partir de pg_upgrade 15 (version 9.2 dans ce cas).

`pg_config` fournit des informations techniques sur les binaires installés (chemins notamment).

`pg_controldata` fournit des informations techniques de base sur une instance.

1.2.5 Autres outils en ligne de commande

- `pgbench` pour des tests
- Outils liés à la réplication/sauvegarde physique, aux tests, analyse…

`pgbench` est l'outil de base pour tester la charge et l'influence de paramètres. Créez les tables de travail avec l'option `-i`, fixez la volumétrie avec `-s`, et lancez `pgbench` en précisant le nombre de clients, de transactions… L'outil vous calcule le nombre de transactions par secondes et diverses informations statistiques. Les requêtes utilisées sont basiques mais vous pouvez fournir les vôtres.

D'autres outils sont liés à l'archivage (`pg_receivewal`) et/ou à la réplication par *log shipping* (`pg_archivecleanup`) ou logique (`pg_recvlogical`), au sauvetage d'instances secondaires (`pg_rewind`), à la vérification de la disponibilité (`pg_isready`), à des tests de la configuration matérielle (`pg_test_fsync`, `pg_test_timing`), ou d'intégrité (`pg_checksums`), à l'analyse des journaux de transactions (`pg_waldump`).

[1] https://www.postgresql.org/docs/current/app-pg-ctl.html

1.3 CHAÎNES DE CONNEXION

Pour se connecter à une base :

- paramètres propres aux outils
- via la libpq
  - variables d'environnement
  - par chaînes clés/valeur
  - par chaînes URI
  - idem en Python,PHP,Perl
- JDBC/.NET/ODBC ont des syntaxes spécifiques

Les types de connexion connus de PostgreSQL et de sa librairie cliente (libpq) sont, au choix, les paramètres explicites (assez variables suivants les outils), les variables d'environnement, les chaînes clés/valeur, et les URI (`postgresql://…`).

Les pilotes PHP[2]…), Perl[3], ou Python[4] utilisent la libpq, comme bien sûr les outils du projet PostgreSQL, et ce qui suit est directement utilisable.

Nous ne traiterons pas ici des syntaxes propres aux outils n'utilisant pas la libpq, comme les pilotes JDBC[5] (dont les URI sont proches mais différente de celles décrites plus bas) ou .NET[6], ou encore les différents pilotes ODBC (comme psqlODBC[7], très limité).

1.3.1 Paramètres

Outils habituels, et très souvent :

```
$ psql -h serveur -d mabase -U nom -p 5432
```

Option	Variable	Valeur par défaut
`-h` HÔTE	$PGHOST	`/tmp` , `/var/run/postgresql`

[2] https://www.php.net/manual/en/function.pg-connect.php
[3] https://metacpan.org/pod/DBD::Pg#data_sources
[4] https://www.psycopg.org/docs/module.html#psycopg2.connect
[5] https://jdbc.postgresql.org/documentation/head/connect.html
[6] https://www.npgsql.org/doc/connection-string-parameters.html
[7] https://odbc.postgresql.org/

Option	Variable	Valeur par défaut
`-p` PORT	$PGPORT	`5432`
`-U` NOM	$PGUSER	nom de l'utilisateur OS
`-d` base	$PGDATABASE	nom de l'utilisateur PG
	$PGOPTIONS	options de connexions

Les options de connexion permettent d'indiquer comment trouver l'instance (serveur, port), puis d'indiquer l'utilisateur et la base de données concernés parmi les différentes de l'instance. Ces deux derniers champs doivent passer le filtre du `pg_hba.conf` du serveur pour que la connexion réussisse.

Lorsque l'une de ces options n'est pas précisée, la bibliothèque cliente PostgreSQL cherche une variable d'environnement correspondante et prend sa valeur. Si elle ne trouve pas de variable, elle se rabat sur une valeur par défaut.

Les paramètres et variables d'environnement qui suivent sont utilisés par les outils du projet, et de nombreux autres outils de la communauté.

La problématique du mot de passe est laissée de côté pour le moment.

Hôte :

Le paramètre `-h <hôte>` ou la variable d'environnement `$PGHOST` permettent de préciser le nom ou l'adresse IP de la machine qui héberge l'instance.

Sans précision, sur Unix, le client se connecte sur la socket Unix, généralement dans `/var/run/postgresql` (défaut sur Debian et Red Hat) ou `/tmp` (défaut de la version compilée). Le réseau n'est alors pas utilisé, et il y a donc une différence entre `-h localhost` (via ::1 ou 127.0.0.1 donc) et `-h /var/run/postgresql` (défaut), ce qui peut avoir un résultat différent selon la configuration du `pg_hba.conf`. Par défaut, l'accès par le réseau exige un mot de passe.

Sous Windows, le comportement par défaut est de se connecter à **localhost**.

Serveur :

`-p <port>` ou `$PGPORT` permettent de préciser le port sur lequel l'instance écoute les connexions entrantes. Sans indication, le port par défaut est le 5432.

Utilisateur :

`-U <nom>` ou `$PGUSER` permettent de préciser le nom de l'utilisateur, connu de PostgreSQL, qui doit avoir été créé préalablement sur l'instance.

Sans indication, le nom d'utilisateur PostgreSQL est le nom de l'utilisateur système connecté.

Base de données :

`-d base` ou `$PGDATABASE` permettent de préciser le nom de la base de données utilisée pour la connexion.

Sans précision, le nom de la base de données de connexion sera celui de l'utilisateur PostgreSQL (qui peut découler de l'utilisateur connecté au système).

Exemples :

- Chaîne de connexion classique, implicitement au port 5432 en local sur le serveur :

```
$ psql -U jeanpierre -d comptabilite
```

- Connexion à un serveur distant pour une sauvegarde :

```
$ pg_dump -h serveur3 -p 5435 -U postgres -d basecritique -f fichier.dump
```

- Connexion sans paramètre via l'utilisateur système **postgres**, et donc implicitement en tant qu'utilisateur **postgres** de l'instance à la base **postgres** (qui existent généralement par défaut).

```
$ sudo -iu postgres  psql
```

Dans les configurations par défaut courantes, cette commande est généralement la seule qui fonctionne sur une instance fraîchement installée.

- Utilisation des variables d'environnement pour alléger la syntaxe dans un petit script de maintenance :

```
#! /bin/bash
export PGHOST=/var/run/postgresql
export PGPORT=5435
export PGDATABASE=comptabilite
export PGUSER=superutilisateur
# liste des bases
psql -l
# nettoyage
vacuumdb
# une sauvegarde
pg_dump -f fichier.dump
```

Raccourcis

Généralement, préciser `-d` n'est pas nécessaire quand la base de données est le premier argument non nommé de la ligne de commande. Par exemple :

```
$ psql -U jeanpierre comptabilite
```

```
$ sudo -iu postgres  vacuumdb nomdelabase
```

Il faut faire attention à quelques différences entre les outils : pour `pgbench`, `-d` désigne le mode debug et le nom de la base est un argument non nommé ; alors que pour `pg_restore`, le `-d` est nécessaire pour restaurer vers une base.

Variable d'environnement PGOPTIONS

La variable d'environnement `$PGOPTIONS` permet de positionner la plupart des paramètres de sessions disponibles, qui surchargent les valeurs par défaut.

Par exemple, pour exécuter une requête avec un paramétrage différent de `work_mem` (mémoire de tri) :

```
$ PGOPTIONS="-c work_mem=100MB" psql -p 5433 -h serveur nombase  < grosse_requete.sql
```

ou pour importer une sauvegarde sans être freiné par un serveur secondaire synchrone :

```
$ PGOPTIONS="-c synchronous_commit=local" pg_restore -d nombase sauvegarde.dump
```

1.3.2 Autres variables d'environnement

- $PGAPPNAME
- $PGSSLMODE
- $PGPASSWORD
- …

Il existe aussi :

- `$PGSSLMODE` pour définir le niveau de chiffrement SSL de la connexion avec les valeurs `disable`, `prefer` (le défaut), `require`… (il existe d'autres variables d'environnement pour une gestion fine du SSL) ;
- `$PGAPPNAME` permet de définir une chaîne de caractère arbitraire pour identifier la connexion dans les outils et vues d'administration (paramètre de session `application_name`) : mettez-y par exemple le nom du script ;
- `$PGPASSWORD` peut stocker le mot de passe pour ne pas avoir à l'entrer à la connexion (voir plus loin).

Toutes ces variables d'environnement, ainsi que de nombreuses autres, et leurs diverses valeurs possibles, sont documentées[8].

1.3.3 Chaînes libpq clés/valeur

```
psql "host=serveur1  user=jeanpierre  dbname=comptabilite"

psql -d "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
     sslmode=require  application_name='chargement'
     options='-c work_mem=30MB' "
```

En lieu du nom de la base, une chaîne peut inclure tous les paramètres nécessaires. Cette syntaxe permet de fournir plusieurs paramètres d'un coup.

[8] https://docs.postgresql.fr/current/libpq-envars.html

Les paramètres disponibles sont listés dans la documentation[9]. On retrouvera de nombreux paramètres équivalents aux variables d'environnement[10], ainsi que d'autres.

Dans cet exemple, on se connecte en exigeant le SSL, en positionnant `application_name` pour mieux repérer la session, et en modifiant la mémoire de tri, ainsi que le paramétrage de la synchronisation sur disque pour accélérer les choses :

```
$ psql "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
       sslmode=require  application_name='chargement'
       options='-c work_mem=99MB' -c synchronous_commit=off' " \
       <  script_chargement.sql
```

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

1.3.4 Chaînes URI

```
psql -d "postgresql://jeanpierre@serveur1:5432/comptabilite"

psql \
"postgres://jeanpierre@serveur1/comptabilite?sslmode=require\
&options=-c%20synchronous_commit%3Doff"

psql -d postgresql://serveur1/comptabilite?user=jeanpierre\&port=5432
```

Une autre possibilité existe : des chaînes sous forme URI comme il en existe pour beaucoup de pilotes et d'outils. La syntaxe est de la forme :

```
postgresql://[user[:motdepasse]@][lieu][:port][/dbname][?param1=valeur21&param2=valeur2…]
```

Là encore cette chaîne remplace le nom de la base dans les outils habituels. `postgres://` et `postgresql://` sont tous deux acceptés.

Cette syntaxe est très souple. Une difficulté réside dans la gestion des caractères spéciaux, signes `=` et des espaces :

```
# Ces deux appels sont équivalents
$ psql -d postgresql:///var/lib/postgresql?dbname=pgbench\&user=pgbench\&port=5436
$ psql -h /var/lib/postgresql -d pgbench -U pgbench -p 5436

# Ces deux appels sont équivalents
$ psql "postgresql://bob@vm1/proddb?&options=-c%20synchronous_commit%3Doff"
$ psql -U bob -h vm1 -d proddb -c 'synchronous_commit=off'
```

[9] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-PARAMKEYWORDS
[10] https://docs.postgresql.fr/current/libpq-envars.html

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

1.3.5 Connexion avec choix automatique du serveur

```
psql "host=serveur1,serveur2,serveur3
      port=5432,5433,5434
      user=jeanpierre  dbname=comptabilite
      target_session_attrs=read-write
      load_balance_hosts=random"   # v16
```

Il est possible d'indiquer plusieurs hôtes et ports. Jusque PostgreSQL 15 inclus, l'hôte sélectionné est le premier qui répond avec les conditions demandées. À partir de PostgreSQL 16, on peut ajouter `load_balance_hosts=random` pour que la libpq choisisse un serveur au hasard, pour une répartition de charge basique, mais très simple à mettre en place.

Si l'authentification ne passe pas, la connexion tombera en erreur. Il est possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule grâce au paramètre `target_session_attrs`.

Par exemple, on se connectera ainsi au premier serveur de la liste ouvert en écriture et disponible parmi les trois précisés :

```
psql postgresql://jeanpierre@serveur1:5432,serveur2:5433,serveur3:5434/\
comptabilite?target_session_attrs=read-write
```

qui équivaut à :

```
psql "host=serveur1,serveur2,serveur3 port=5432,5433,5434
      target_session_attrs=read-write  user=jeanpierre  dbname=comptabilite"
```

Depuis la version 14, dans le but de faciliter la connexion aux différents serveurs d'une grappe, `target_session_attrs` possède d'autres options[11] que `read-write`, à savoir : `any`, `read-only`, `primary`, `standby`, `prefer-standby`. Par exemple, la commande suivante permet de se connecter à un secondaire au hasard parmi ceux qui répondent. `serveur3` est doublé pour prendre une charge double des autres. S'il ne reste plus que le primaire, la connexion se fera à celui-ci.

```
psql "host=serveur1,serveur2,serveur3,serveur3 port=5432,5433,5434,5434
      target_session_attrs=prefer-standby
      load_balance_hosts=random
      user=jeanpierre  dbname=comptabilite"
```

[11] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-CONNECT-TARGET-SESSION-ATTRS

1.3.6 Authentification d'un client (outils console)

- En interactif (`psql`)
  - `-W | --password`
  - `-w | --no-password`
- Variable `$PGPASSWORD`
- À préférer : fichier `.pgpass`
  - `chmod 600 .pgpass`
  - `nom_hote:port:database:nomutilisateur:motdepasse`

Options -W et -w de psql

L'option `-W` oblige à saisir le mot de passe de l'utilisateur. C'est le comportement par défaut si le serveur demande un mot de passe. Si les accès aux serveurs sont configurés sans mot de passe et que cette option est utilisée, le mot de passe sera demandé et fourni à PostgreSQL lors de la demande de connexion. Mais PostgreSQL ne vérifiera pas s'il est bon si la méthode d'authentification ne le réclame pas.

L'option `-w` empêche la saisie d'un mot de passe. Si le serveur a une méthode d'authentification qui nécessite un mot de passe, ce dernier devra être fourni par le fichier `.pgpass` ou par la variable d'environnement `$PGPASSWORD`. Dans tous les autres cas, la connexion échoue.

Variable $PGPASSWORD

Si `psql` détecte une variable `$PGPASSWORD` initialisée, il se servira de son contenu comme mot de passe qui sera soumis pour l'authentification du client.

Fichier .pgpass

Le fichier `.pgpass`, situé dans le répertoire personnel de l'utilisateur ou celui référencé par `$PGPASSFILE`, est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\pgpass.conf` (où `%APPDATA%` fait référence au sous-répertoire `Application Data` du profil de l'utilisateur).

Ce fichier devra être composé de lignes du format :

```
nom_hote:port:nom_base:nom_utilisateur:mot_de_passe
```

Chacun des quatre premiers champs pourraient être une valeur littérale ou `*` (qui correspond à tout). La première ligne réalisant une correspondance pour les paramètres de connexion sera utilisée (du

coup, placez les entrées plus spécifiques en premier lorsque vous utilisez des jokers). Si une entrée a besoin de contenir `*` ou `\`, échappez ce caractère avec `\`. Un nom d'hôte `localhost` correspond à la fois aux connexions `host` (`TCP`) et aux connexions `local` (`socket` de domaine *Unix*) provenant de la machine locale.

Les droits sur `.pgpass` doivent interdire l'accès aux autres et au groupe ; réalisez ceci avec la commande :

```
chmod 0600 ~/.pgpass
```

Attention : si les droits du fichier sont moins stricts, le fichier sera ignoré !

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

1.4 LA CONSOLE PSQL

- Un outil simple pour
  - les opérations courantes
  - les tâches de maintenance
  - l'exécution des scripts
  - les tests

```
postgres$ psql
  base=#
```

La console `psql` permet d'effectuer l'ensemble des tâches courantes d'un utilisateur de bases de données. Si ces tâches peuvent souvent être effectuées à l'aide d'un outil graphique, la console présente l'avantage de pouvoir être utilisée en l'absence d'environnement graphique ou de scripter les opérations à effectuer sur la base de données. Elle a la garantie d'être également toujours disponible.

Nous verrons également qu'il est possible d'administrer la base de données depuis cette console.

Enfin, elle permet de tester l'activité du serveur, l'existence d'une base, la présence d'un langage de programmation…

1.4.1 Obtenir de l'aide et quitter

- Obtenir de l'aide sur les commandes internes `psql`
  - `\?`
- Obtenir de l'aide sur les ordres SQL
  - `\h <COMMANDE>`
- Quitter
  - `\q` ou `ctrl-D`
  - `quit` ou `exit` (v11)

`\?` liste les commandes propres à `psql` (par exemple `\d` ou `\du` pour voir les tables ou les utilisateurs), trop nombreuses pour pouvoir être mémorisées.

`\h <COMMANDE>` affiche l'aide en ligne des commandes SQL. Sans argument, la liste des commandes disponibles est affichée. La version 12 ajoute en plus l'URL vers la page web documentant cette commande.

Exemple :

```
postgres=# \h ALTER TA

Commande :     ALTER TABLE
Description : modifier la définition d'une table
Syntaxe :
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
…
URL: https://www.postgresql.org/docs/15/sql-altertable.html
```

`\q` ou `Ctrl-D` permettent de quitter `psql`. Depuis la version 11, il est aussi possible d'utiliser `quit` ou `exit`.

1.4.2 Gestion de la connexion

- Modifier le mot de passe d'un utilisateur
  - `\password  nomutilisateur`
- Quelle est la connexion courante ?
  - `\conninfo`
  - `SELECT current_user,session_user,system_user;`
- Se connecter à une autre base :
  - `\c ma base`
  - `\c mabase  utilisateur serveur 5432`

`\c` permet de changer d'utilisateur et/ou de base de données sans quitter le client. `\conninfo` permet de savoir où l'on est connecté.

Exemple :

```
CREATE DATABASE formation;
CREATE DATABASE prod;
CREATE USER stagiaire1;
CREATE USER stagiaire2;
CREATE USER admin;
```

```
postgres=# \c formation stagiaire1

You are now connected to database "formation" as user "stagiaire1".

formation=> \c - stagiaire2

You are now connected to database "formation" as user "stagiaire2".

formation=> \c prod admin

You are now connected to database "prod" as user "admin".

prod=> \conninfo

You are connected to database "prod" as user "admin"
                  on host "localhost" (address "::1") at port "5412".
```

Au niveau SQL, un équivalent partiel de `\conninfo` serait :

```
SELECT  current_user,
        current_catalog,                          -- base
        inet_server_addr(), inet_server_port(),   -- serveur
        system_user                               -- depuis PG16
        \gx

-[ RECORD 1 ]----+-----------------------
current_user     | dalibo
current_catalog  | pgbench
inet_server_addr | 192.168.74.5
inet_server_port | 5433
system_user      | scram-sha-256:postgres
```

Un superutilisateur pourra affecter un mot de passe à un autre utilisateur grâce à la commande `\password`, qui en fait encapsule un `ALTER USER … PASSWORD …`. Le gros intérêt de `\password` est d'envoyer le mot de passe chiffré au serveur. Ainsi, même si les traces contiennent toutes les requêtes SQL exécutées, il est impossible de retrouver les mots de passe via le fichier de traces. Ce n'est pas le cas avec un `CREATE ROLE` ou un `ALTER ROLE` manuel (à moins de chiffrer soi-même le mot de passe).

1.4.3 Catalogue système : objets utilisateurs

Lister :

- les bases de données
  - `\l` , `\l+`
- les tables
  - `\d` , `\d+` , `\dt` , `\dt+`
- les index
  - `\di` , `\di+`
- les schémas
  - `\dn`
- les fonctions & procédures
  - `\df[+]`
- etc…

Ces commandes permettent d'obtenir des informations sur les objets utilisateurs de tout type stockés dans la base de données. Pour les commandes qui acceptent un motif, celui-ci permet de restreindre les résultats retournés à ceux dont le nom d'opérateur correspond au motif précisé.

Le client psql en version 15 est compatible avec toutes les versions du serveur depuis la 9.2 incluse.

`\l` dresse la liste des bases de données sur le serveur. Avec `\l+`, les commentaires, les tablespaces par défaut et les tailles des bases sont également affichés, ce qui peut être très pratique.

`\dt` affiche les tables, `\di` les index, `\dn` les schémas, `\ds` les séquences, `\dv` les vues, etc. Là encore, on peut ajouter `+` pour d'autres informations comme la taille, et même `S` pour inclure les objets système normalement masqués.

Exemple :

Si l'on crée une simple base grâce à `pgbench` avec un utilisateur dédié :

```
$ psql
```

```
=# CREATE ROLE testeur LOGIN ;
=# \password testeur
Saisir le nouveau mot de passe :
Saisir le mot de passe à nouveau :
```

Utilisateurs en place :

```
=# \du
                              Liste des rôles
         Nom du rôle          |        Attributs        |       Membre de
------------------------------+-------------------------+------------------
 dalibo                       |                         | {}
 nagios                       | Superutilisateur        | {}
 postgres                     | Superutilisateur, (...) | {}
 testeur                      |                         | {}
```

Création de la base :

```
CREATE DATABASE pgbench OWNER testeur ;
```

Bases de données en place :

```
=# \l
                            Liste des bases de données
      Nom       | Propriétaire | Encodage | Collationnement | Type caract. | …
----------------+--------------+----------+-----------------+--------------+----
 pgbench        | testeur      | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 postgres       | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
```

Création des tables de cette nouvelle base :

```
$ pgbench --initialize --scale=10 -U testeur -h localhost pgbench
```

Connexion à la nouvelle base :

```
$ psql -h localhost -U testeur -d pgbench
```

Les tables :

```
=# \d
               Liste des relations
 Schéma |       Nom        | Type  | Propriétaire
--------+------------------+-------+--------------
 public | pgbench_accounts | table | testeur
 public | pgbench_branches | table | testeur
 public | pgbench_history  | table | testeur
 public | pgbench_tellers  | table | testeur
(4 lignes)

=# \dt+ pgbench_*s
                          Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Persistence |  M…  | Taille |
--------+------------------+-------+--------------+-------------+------+--------+--
 public | pgbench_accounts | table | testeur      | permanent   | heap | 128 MB |
 public | pgbench_branches | table | testeur      | permanent   | heap | 40 kB  |
 public | pgbench_tellers  | table | testeur      | permanent   | heap | 40 kB  |
(3 lignes)
```

Une table (affichage réduit pour des raisons de mise en page) :

```
=# \d+ pgbench_accounts
                          Table « public.pgbench_accounts »
 Colonne  |     Type      | Col..nt | NULL-able | … | Stockage | Compr. | …
----------+---------------+---------+-----------+---+----------+--------+--
```

```
 aid      | integer       |         | not null  |   | plain    |        |
 bid      | integer       |         |           |   | plain    |        |
 abalance | integer       |         |           |   | plain    |        |
 filler   | character(84) |         |           |   | extended |        |
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Méthode d'accès : heap
Options: fillfactor=100
```

Les index :

```
=> \di
                          Liste des relations
 Schéma |          Nom          | Type  | Propriétaire |      Table
--------+-----------------------+-------+--------------+------------------
 public | pgbench_accounts_pkey | index | testeur      | pgbench_accounts
 public | pgbench_branches_pkey | index | testeur      | pgbench_branches
 public | pgbench_tellers_pkey  | index | testeur      | pgbench_tellers
(3 lignes)
```

Les schémas, utilisateur ou système :

```
=> \dn+
                          Liste des schémas
  Nom   | Propriétaire |    Droits d´accès    |      Description
--------+--------------+----------------------+------------------------
 public | postgres     | postgres=UC/postgres+| standard public schema
        |              | =UC/postgres         |
(1 ligne)

=> \dnS
         Liste des schémas
        Nom         | Propriétaire
--------------------+--------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | postgres
(4 lignes)
```

Les tablespaces (ici ceux par défaut) :

```
=> \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
(2 lignes)
```

1.4.4 Catalogue système : rôles et accès

- Lister les rôles (utilisateurs et groupes)
  - `\du[+]`
- Lister les droits d'accès
  - `\dp`
- Lister les droits d'accès par défaut
  - `\ddp`
  - `ALTER DEFAULT PRIVILEGES`
- Lister les configurations par rôle et par base
  - `\drds`

On a vu que `\du` (`u` pour *user*) affiche les rôles existants. Rappelons qu'un rôle peut être aussi bien un utilisateur (si le rôle a le droit de `LOGIN`) qu'un groupe d'utilisateurs, voire les deux à la fois.

Dans les versions antérieures à la 8.1, il n'y avait pas de rôles, et les groupes et les utilisateurs étaient deux notions distinctes. Certaines commandes ont conservé le terme de *user*, mais il s'agit bien de rôles dans tous les cas.

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

Dans cet exemple, le rôle **admin** devient membre du rôle système **pg_signal_backend** :

```
=# GRANT pg_signal_backend TO admin;

=# \du
                                List of roles
 Nom du rôle |                   Attributs         |      Membre de
-------------+-----------------------------------+---------------------
 admin       |                                   | {pg_signal_backend}
 postgres    | Superuser, Create role, Create DB,  |
             | Replication, Bypass RLS             | {}
 …
```

Toujours dans la base **pgbench** :

```
=# GRANT SELECT ON TABLE pgbench_accounts TO dalibo ;
```

```
=# GRANT ALL ON TABLE pgbench_history TO dalibo ;
=# \z
```

```
=> \z
                                  Droits d'accès
 Schéma |       Nom        | Type  |     Droits d'accès      | Droits … | …
--------+------------------+-------+-------------------------+----------+---
 public | pgbench_accounts | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=r/testeur        |          |
 public | pgbench_branches | table |                         |          |
 public | pgbench_history  | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=arwdDxt/testeur  |          |
 public | pgbench_tellers  | table |                         |          |
(4 lignes)
```

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

```
=# ALTER DEFAULT PRIVILEGES GRANT ALL ON TABLES TO dalibo ;
```

```
=# \ddp
                Droits d´accès par défaut
 Propriétaire | Schéma | Type  |     Droits d´accès
--------------+--------+-------+------------------------
 testeur      |        | table | dalibo=arwdDxt/testeur +
              |        |       | testeur=arwdDxt/testeur
(1 ligne)
```

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

```
=# ALTER DATABASE pgbench SET work_mem TO '15MB';
=# ALTER ROLE testeur SET log_min_duration_statement TO '0';
```

```
=# \drds
              Liste des paramètres
  Rôle   | Base de données |         Réglages
---------+-----------------+--------------------------
 testeur |                 | log_min_duration_statement=0
         | pgbench         | work_mem=15MB
…
```

1.4.5 Visualiser le code des objets

- Voir les vues ou les fonctions & procédures
  - `\dv`, `\df`
- Code d'une vue
  - `\sv`
- Code d'une procédure stockée
  - `\sf`

Ceci permet de visualiser le code de certains objets sans avoir besoin de l'éditer. Par exemple avec cette vue système :

```
=# \dv pg_tables
             Liste des relations
   Schéma   |    Nom    | Type | Propriétaire
------------+-----------+------+--------------
 pg_catalog | pg_tables | vue  | postgres
(1 ligne)


=# \sv+ pg_tables
1       CREATE OR REPLACE VIEW pg_catalog.pg_tables AS
2        SELECT n.nspname AS schemaname,
3           c.relname AS tablename,
4           pg_get_userbyid(c.relowner) AS tableowner,
5           t.spcname AS tablespace,
6           c.relhasindex AS hasindexes,
7           c.relhasrules AS hasrules,
8           c.relhastriggers AS hastriggers,
9           c.relrowsecurity AS rowsecurity
10         FROM pg_class c
11           LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
12           LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
13        WHERE c.relkind = ANY (ARRAY['r'::"char", 'p'::"char"])
```

Ou cette fonction :

```
=# CREATE FUNCTION nb_sessions_actives () RETURNS int
LANGUAGE sql
AS $$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$$ ;
```

```
=# \df nb_sessions_actives
                                        Liste des fonctions
 Schéma |         Nom          | Type … du résultat | Type …des paramètres | Type
--------+----------------------+--------------------+----------------------+------
 public | nb_sessions_actives | integer            |                      | func
(1 ligne)
```

```
=# \sf nb_sessions_actives
CREATE OR REPLACE FUNCTION public.nb_sessions_actives()
 RETURNS integer
 LANGUAGE sql
AS $function$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$function$
```

Il est même possible de lancer un éditeur depuis `psql` pour modifier directement la vue ou la fonction avec respectivement `\ev` ou `\ef`.

1.4.6 Configuration

- Lister les paramètres correspondants au motif indiqué
  - `\dconfig` (v15+)

La méta-commande `\dconfig` permet de récupérer la configuration d'un paramètre. Par exemple :

```
b1=# \dconfig log_r*
  List of configuration parameters
         Parameter          | Value
----------------------------+-------
 log_recovery_conflict_waits | off
 log_replication_commands    | off
 log_rotation_age            | 1d
 log_rotation_size           | 0
(4 rows)
```

En ajoutant le signe `+`, il est possible d'obtenir plus d'informations :

```
b1=# \dconfig+ log_r*
                      List of configuration parameters
         Parameter          | Value |  Type   |  Context  | Access privileges
----------------------------+-------+---------+-----------+-------------------
 log_recovery_conflict_waits | off   | bool    | sighup    |
 log_replication_commands    | off   | bool    | superuser |
 log_rotation_age            | 1d    | integer | sighup    |
 log_rotation_size           | 0     | integer | sighup    |
(4 rows)
```

1.4.7 Exécuter des requêtes

- Exécuter une requête :

```
SELECT * FROM pg_tables ;
SELECT * FROM pg_tables \g
SELECT * FROM pg_tables \gx    -- une ligne par champ
INSERT INTO … VALUES (1) \; INSERT INTO … VALUES (2) ; -- 1 transaction
```

- Rappel des requêtes:
  - flèche vers le haut
  - `\g`
  - `Ctrl-R` suivi d'un extrait de texte représentatif

Les requêtes SQL doivent se terminer par `;` ou, pour marquer la parenté de PostgreSQL avec Ingres, `\g`. Cette dernière commande permet de relancer un ordre.

Depuis version 10, il est aussi possible d'utiliser `\gx` pour bénéficier de l'affichage étendu sans avoir à jouer avec `\x on` et `\x off`.

La console `psql`, lorsqu'elle est compilée avec la bibliothèque `libreadline` ou la bibliothèque `libedit` (cas des distributions Linux courantes), dispose des mêmes possibilités de rappel de commande que le shell bash.

Exemple :

```
postgres=# SELECT * FROM pg_tablespace LIMIT 5;

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# SELECT * FROM pg_tablespace LIMIT 5\g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# \g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
```

```
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |


postgres=# \gx


-[ RECORD 1 ]----------
oid        | 1663
spcname    | pg_default
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 2 ]----------
oid        | 1664
spcname    | pg_global
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 3 ]----------
oid        | 16502
spcname    | ts1
spcowner   | 10
spcacl     |
spcoptions |
```

Plusieurs ordres sur la même ligne séparés par des `;` seront exécutés et affichés à la suite. Par contre, s'ils sont séparés par `\;`, ils seront envoyés ensemble et implicitement dans une même transaction (sauf utilisation explicite de `BEGIN` / `COMMIT` / `ROLLBACK`). Avant la version 15, ne sera affiché que le résultat du dernier ordre.

Dans cet exemple, la division par zéro fait tomber en erreur et annule l'insertion de la valeur 2 car les deux sont dans la même transaction :

```
=# CREATE TABLE demo_insertion (i float);
CREATE TABLE
=# INSERT INTO demo_insertion VALUES(1.0); INSERT INTO demo_insertion VALUES(1.0/0);
INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion \; INSERT INTO demo_insertion VALUES (2.0) \;
   INSERT INTO demo_insertion VALUES (2.0/0) ;
 i
 1
(1 ligne)

INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion ;
 i
 1
(1 ligne)
```

1.4.8 Afficher le résultat d'une requête

- `\x` pour afficher un champ par ligne
- Affichage par paginateur si l'écran ne suffit pas
- Préférer `less` :
  - `set PAGER='less -S'`
- Ou outil dédié :
  - `\setenv PAGER 'pspg'`

En mode interactif, `psql` cherche d'abord à afficher directement le résultat :

```
postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

      relname      | reltype | relchecks |  oid  |  oid
-------------------+---------+-----------+-------+-------
 pg_statistic      |   11319 |         0 |  2619 |  2619
 t3                |   16421 |         0 | 16419 | 16419
 capitaines_id_seq |   16403 |         0 | 16402 | 16402
 t1                |   16415 |         0 | 16413 | 16413
 t2                |   16418 |         0 | 16416 | 16416
```

S'il y a trop de colonnes, on peut préférer n'avoir qu'un champ par ligne grâce au commutateur `\x` :

```
postgres=# \x on

Expanded display is on.

postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

-[ RECORD 1 ]----------------
relname   | pg_statistic
reltype   | 11319
relchecks | 0
oid       | 2619
oid       | 2619
-[ RECORD 2 ]----------------
relname   | t3
reltype   | 16421
relchecks | 0
oid       | 16419
oid       | 16419
-[ RECORD 3 ]----------------
relname   | capitaines_id_seq
reltype   | 16403
relchecks | 0
```

```
oid       | 16402
oid       | 16402
```

`\x on` et `\x off` sont alternativement appelés si l'on tape plusieurs fois `\x`. `\x auto` délègue à `psql` la décision du meilleur affichage, en général à bon escient. `\gx` à la place de `;` bascule l'affichage pour une seule requête.

S'il n'y a pas la place à l'écran, `psql` appelle le paginateur par défaut du système. Il s'agit souvent de `more`, parfois de `less`. Ce dernier est bien plus puissant, permet de parcourir le résultat en avant et en arrière, avec la souris, de chercher une chaîne de caractères, de tronquer les lignes trop longues (avec l'option `-S`) pour naviguer latéralement.

Le paramétrage du paginateur s'effectue par des variables d'environnement :

```
export PAGER='less -S'
psql
```

ou :

```
PAGER='less -S'  psql
```

ou dans `psql` directement, ou `.psqlrc` :

```
\setenv PAGER 'less -S'
```

Mais `less` est généraliste. Un paginateur dédié à l'affichage de résultats de requêtes a été récemment développé par Pavel Stěhule[12] et son paquet figure dans les principales distributions.

Pour les gens qui consultent souvent des données dans les tables depuis la console, `pspg` permet de naviguer dans les lignes avec la souris en figeant les entêtes ou quelques colonnes ; de poser des signets sur des lignes ; de sauvegarder les lignes. (Pour plus de détail, voir cette présentation[13] et la page Github du projet[14]). La mise en place est similaire :

```
\setenv PAGER 'pspg'
```

À l'inverse, la pagination se désactive complètement avec :

```
\pset pager
```

(et bien sûr en mode non interactif, par exemple quand la sortie de `psql` est redirigée vers un fichier).

[12] http://okbob.blogspot.fr/2017/07/i-hope-so-every-who-uses-psql-uses-less.html
[13] http://blog-postgresql.verite.pro/2017/11/21/test-pspg.html
[14] https://github.com/okbob/pspg

1.4.9 Afficher les détails d'une requête

- `\gdesc`
- Afficher la liste des colonnes correspondant au résultat d'exécution d'une requête
  - noms
  - type de données

Après avoir exécuté une requête, ou même à la place de l'exécution, il est possible de connaître le nom des colonnes en résultat ainsi que leur type de données.

Requête :

```
SELECT nspname, relname
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

Description des colonnes :

```
postgres=# \gdesc

 Column  | Type
---------+------
 nspname | name
 relname | name
```

Ou sans exécution :

```
postgres=# SELECT * FROM generate_series (1, 1000) \gdesc

     Column      |  Type
-----------------+---------
 generate_series | integer
```

1.4.10 Exécuter le résultat d'une requête

- Exécuter le résultat d'une requête
  - `\gexec`

Parfois, une requête permet de créer des requêtes sur certains objets. Par exemple, si nous souhaitons exécuter un `VACUUM` sur toutes les tables du schéma `public`, nous allons récupérer la liste des tables avec cette requête :

```
SELECT nspname, relname FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
 nspname |     relname
---------+------------------
 public  | pgbench_branches
 public  | pgbench_tellers
 public  | pgbench_accounts
 public  | pgbench_history
(4 lignes)
```

Plutôt que d'éditer manuellement cette liste de tables pour créer les ordres SQL nécessaires, autant modifier la requête pour qu'elle prépare elle-même les ordres SQL :

```
SELECT 'VACUUM ' || quote_ident(nspname) || '.' || quote_ident(relname)
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
           ?column?
--------------------------------
 VACUUM public.pgbench_branches
 VACUUM public.pgbench_tellers
 VACUUM public.pgbench_accounts
 VACUUM public.pgbench_history
(4 lignes)
```

Une fois que nous avons vérifié la validité des requêtes SQL, il ne reste plus qu'à les exécuter. C'est ce que permet la commande `\gexec` :

```
=# \gexec
```

```
VACUUM
VACUUM
VACUUM
VACUUM
```

1.4.11 Manipuler le tampon de requêtes

- Éditer
  - dernière requête : `\e`
  - vue : `\ev nom_vue`
  - fonction PL/pgSQL : `\ef nom_fonction`
- Historique :
  - `\s`

`\e nomfichier.sql` édite le tampon de requête courant ou un fichier existant indiqué à l'aide d'un éditeur externe.

L'éditeur désigné avec les variables d'environnement `$EDITOR` ou `$PSQL_EDITOR` notamment. Sur Unix, c'est généralement par défaut une variante de `vi` mais n'importe quel éditeur fait l'affaire :

```
PSQL_EDITOR=nano  psql
```

ou dans `psql` ou dans le `.psqlrc` :

```
\setenv PSQL_EDITOR 'gedit'
```

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

`\r` efface la requête qui est en cours d'écriture. La précédente requête reste accessible avec `\p`.

`\w nomfichier.sql` provoque l'écriture du tampon de requête dans le fichier indiqué (à modifier par la suite avec `\e` par exemple).

`\ev v_mavue` édite la vue indiquée. Sans argument, cette commande affiche le squelette de création d'une nouvelle vue.

`\ef f_mafonction` est l'équivalent pour une fonction.

`\s` affiche l'historique des commandes effectuées dans la session. `\s historique.log` sauvegarde cet historique dans le fichier indiqué.

1.4.12 Entrées/sorties

- Charger et exécuter un script SQL
  - `\i fichier.sql`
- Rediriger la sortie dans un fichier
  - `\o resultat.out`
- Écrire un texte sur la sortie standard
  - `\echo "Texte…"`

`\i fichier.sql` lance l'exécution des commandes placées dans le fichier passé en argument. `\ir` fait la même chose sauf que le chemin est relatif au chemin courant.

`\o resultat.out` envoie les résultats de la requête vers le fichier indiqué (voire vers une commande UNIX via un *pipe*).

Exemple :

```
=# \o tables.txt
=# SELECT * FROM pg_tables ;
```

(Si l'on intercale `\H`, on peut avoir un formatage en HTML.)

Attention : cela va concerner toutes les commandes suivantes. Entrer `\o` pour revenir au mode normal.

`\echo "Texte"` affiche le texte passé en argument sur la sortie standard. Ce peut être utile entre les étapes d'un script.

1.4.13 Gestion de l'environnement système

- Chronométrer les requêtes
  - `\timing on`
- Exécuter une commande OS
  - `\! ls -l` (sur le client !)
- Changer de répertoire courant
  - `\cd /tmp`
- Affecter la valeur d'une variable d'environnement (v15+)
  - `\getenv toto PATH`

`\timing on` active le chronométrage de toutes les commandes. C'est très utile mais alourdit l'affichage. Sans argument, la valeur actuelle est basculée de `on` à `off` et vice-versa.

```
=# \timing on
Chronométrage activé.

=# VACUUM ;
VACUUM
Temps : 26,263 ms
```

`\! <commande>` ouvre un shell interactif en l'absence d'argument ou exécute la commande indiquée **sur le client** (pas le serveur !) :

`\cd` (et non `\! cd` !) permet de changer de répertoire courant, là encore **sur le client**. Cela peut servir pour définir le chemin d'un script à exécuter ou d'un futur export.

`\getenv` permet de récupérer la valeur d'une valeur d'environnement système et de l'affecter à une variable.

Exemple :

```
\! cat nomfichier.out
\! ls -l /tmp
\! mkdir /home/dalibo/travail
\cd /home/dalibo/travail
\! pwd
/home/dalibo/travail
```

1.4.14 Variables internes psql

- Positionner des variables internes
  - `\set NOMVAR nouvelle_valeur`
- Variables internes usuelles
  - `ON_ERROR_STOP` : `on` / `off`
  - `ON_ERROR_ROLLBACK` : `on` / `off` / `interactive`
  - `ROW_COUNT` : nombre de lignes renvoyées par la dernière requête (v11)
  - `ERROR` : `true` si dernière requête en erreur (v11)
- Ne pas confondre avec `SET` (au niveau du serveur) !

Les variables déclarées avec `\set` sont positionnées au niveau de `psql`, outil client. Elles ne sont pas connues du serveur et n'existent pas dans les autres outils clients (pgAdmin, DBeaver…).
Il ne faut pas les confondre avec les paramètres définis sur le serveur au niveau de la session avec `SET`. Ceux-ci sont transmis directement au serveur quand ils sont entrés dans un outil client, quel qu'il soit.

`\set` affiche les variables internes et utilisateur.

`\set NOMVAR nouvelle_valeur` permet d'affecter une valeur.

La liste des variables prédéfinies est disponible dans la documentation de psql[15]. Beaucoup modifient le comportement de `psql`.

Exemple :

```
postgres=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'b1'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
FETCH_COUNT = '0'
HIDE_TABLEAM = 'off'
HIDE_TOAST_COMPRESSION = 'off'
HISTCONTROL = 'none'
```

[15] https://docs.postgresql.fr/current/app-psql.html#app-psql-variables

```
HISTSIZE = '500'
HOST = '/var/run/postgresql'
IGNOREEOF = '0'
LAST_ERROR_MESSAGE = ''
LAST_ERROR_SQLSTATE = '00000'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5432'
PROMPT1 = '%/%R%x%# '
PROMPT2 = '%/%R%x%# '
PROMPT3 = '>> '
QUIET = 'off'
SERVER_VERSION_NAME = '15.1'
SERVER_VERSION_NUM = '150001'
SHOW_ALL_RESULTS = 'on'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 15.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
↪  20210514 (Red Hat 8.5.0-15), 64-bit'
VERSION_NAME = '15.1'
VERSION_NUM = '150001'
```

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

La version 11 ajoute quelques variables internes. `ROW_COUNT` indique le nombre de lignes de résultat de la dernière requête exécutée :

```
=# SELECT * FROM pg_namespace;

      nspname       | nspowner |               nspacl
--------------------+----------+-------------------------------------
 pg_toast           |       10 |
 pg_temp_1          |       10 |
 pg_toast_temp_1    |       10 |
 pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
 public             |       10 | {postgres=UC/postgres,=UC/postgres}
 information_schema |       10 | {postgres=UC/postgres,=U/postgres}

=# \echo :ROW_COUNT
6

=# SELECT :ROW_COUNT ;
6
```

alors que `ERROR` est un booléen indiquant si la dernière requête était en erreur ou pas :

```
=# CREATE TABLE t1(id integer);
CREATE TABLE

=# CREATE TABLE t1(id integer);

ERROR:  relation "t1" already exists
```

```
postgres=# \echo :ERROR
true
postgres=# CREATE TABLE t2(id integer);
CREATE TABLE
postgres=# \echo :ERROR
false
```

1.4.15 Variables utilisateur psql

- Définir une variable utilisateur
  - `\set NOMVAR nouvelle_valeur`
- Invalider une variable
  - `\unset NOMVAR`
- Stockage du résultat d'une requête :
  - si résultat est une valeur unique
  - Exemple :

```
SELECT now() AS maintenant \gset
SELECT :'maintenant' ;
```

En dehors des variables internes évoquées dans le chapitre précédent, il est possible à un utilisateur de `psql` de définir ses propres variables.

```
-- initialiser avec une constante
\set active 'y'
\echo :active
y
-- initialiser avec le résultat d'une commande système
\set uptime `uptime`
\echo :uptime

 09:38:58 up 53 min,  1 user,  load average: 0,12, 0,07, 0,07
```

Et pour supprimer la variable :

```
\unset uptime
```

Il est possible de stocker le résultat d'une requête dans une variable pour sa réutilisation dans un script avec la commande `\gset`. Le nom de la variable est celui de la colonne ou de son alias. La valeur retournée par la requête doit être unique sous peine d'erreur.

```
SELECT now() AS "curdate" \gset
\echo :curdate
```

```
2020-08-16 10:53:51.795582+02
```

Il est possible aussi de donner un préfixe au nom de la variable :

```
SELECT now() AS "curdate" \gset v_
\echo :v_curdate

2020-08-16 10:54:20.484344+02
```

1.4.16 Tests conditionnels

- `\if`
- `\elif`
- `\else`
- `\endif`

Ces quatre instructions permettent de tester la valeur de variables `psql`, ce qui permet d'aller bien plus loin dans l'écriture de scripts SQL. Le client doit être en version 10 au moins (pas forcément le serveur).

Par exemple, si on souhaite savoir si on se trouve sur un serveur standby ou sur un serveur primaire, il suffit de configurer la variable `PROMPT1` à partir du résultat de l'interrogation de la fonction `pg_is_in_recovery()`. Pour cela, il faut enregistrer ce code dans le fichier `.psqlrc` :

```
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
    \set PROMPT1 'standby %x$ '
\else
    \set PROMPT1 'primaire %x$ '
\endif
```

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

```
psql (15.1)
Type "help" for help.

primaire $
```

alors qu'on obtient sur un serveur secondaire :

```
psql (15.1)
Type "help" for help.

standby $
```

1.4.17 Personnaliser psql

- Fichier de configuration `~/.psqlrc`
  - voire `~/.psqlrc-X.Y` ou `~/.psqlrc-X`
  - ignoré avec `-X`
- Exemple de `.psqlrc` :

```
\set ON_ERROR_ROLLBACK interactive  -- paramétrage de session
\timing on
\set PROMPT1 '%M:%> %n@%/%R%#%x'    -- invite
\set cfg 'SHOW ALL ;'               -- requête utilisable avec :cfg
\set cls '\\! clear;'               -- nettoyer l'écran avec :cls
```

`psql` est personnalisable par le biais de plusieurs variables internes. Il est possible de pérenniser ces personnalisations par le biais d'un fichier `~/.psqlrc` (ou `%APPDATA%\postgresql\psqlrc.conf` sous Windows, ou dans un répertoire désigné par `$PSQLRC`). Il peut exister des fichiers par version (par exemple `~/.psqlrc-15` ou `~/.psqlrc-15.0`), voire un fichier global[16].

Modifier l'invite est utile pour toujours savoir où et comment l'on est connecté. Tous les détails sont dans la documentation[17]. Par exemple, ajouter `%>` dans l'invite affiche le port. On peut aussi afficher toutes les variables listées par `\set` (par exemple ainsi : `%:PORT:` ou `%:USER:`). En cas de reconnexion, psql en régénère certaines, mais pas toutes.

Exemple de fichier `.psqlrc` :

```
\set QUIET 1
\timing on
\pset pager on
\setenv pager
\pset null 'ø'
-- Mots clés autocomplétés en majuscule
\set COMP_KEYWORD_CASE upper
-- Affichage
\x auto
\pset linestyle 'unicode'
\pset border 2
\pset unicode_border_linestyle double
\pset unicode_column_linestyle double
\pset unicode_header_linestyle double
-- Prompt dynamique
-- %> indique le port, %m le serveur, %n l'utilisateur, %/ la base…
```

[16] https://docs.postgresql.fr/current/app-psql.html#id-1.9.4.20.10
[17] https://docs.postgresql.fr/current/app-psql.html#app-psql-prompting

```
\set PROMPT1 '%m:%> %[%033[1;33;40m%]%n@%/%R%[%033[0m%]%#%x '
-- serveur secondaire ? (NB : non mis à jour lors d'une reconnexion !)
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
   \set PROMPT1 :PROMPT1 '(standby) '
\else
   \set PROMPT1 :PROMPT1
\endif
\set QUIET 0
```

```
$ psql -h serveur -p5435 -U jeanpierre -d mabase

[serveur]:5435 jeanpierre@postgres=# (standby) SELECT pi(), now(), null ;
```

pi	now	?column?
3.141592653589793	2022-01-07 16:08:39.925262+01	ø

```
(1 ligne)
```

Il est aussi possible d'y rajouter du paramétrage de session avec `SET` pour adapter le fuseau horaire, par exemple.

Des requêtes très courantes peuvent être ajoutées dans le `.psqlrc`, par exemple celles-ci :

```
-- Paramètres en cours avec leur source
-- Ceci impérativement sur une seule ligne !
\set config 'SELECT name, current_setting(name), CASE source WHEN $$configuration
 ↪  file$$ THEN
regexp_replace(sourcefile, $$^/.*/$$, $$$$)||$$:$$||sourceline ELSE source END FROM
 ↪  pg_settings
WHERE source <> $$default$$ OR name LIKE $$%.%$$;'
```

(Requête inspirée de Christoph Berg[18]).

```
\set extensions 'SELECT * FROM pg_available_extensions ORDER BY name ;'
```

…utilisables ainsi dans `psql` :

```
=# :config

=# :extensions
```

Attention : le `.psqlrc` n'est exécuté qu'au démarrage de `psql`, mais pas lors d'une reconnexion avec `\c` ! Les prompts dynamiques à base de variables utilisateur sont donc susceptibles d'être alors faux ! Pour relancer le script, utiliser :

```
\i ~/.psqlrc
```

[18] https://www.postgresql.org/message-id/YlFQLzlPi4QD0wSi%40msg.df7cb.de

Dans un script, il vaut mieux ignorer ce fichier de configuration grâce à `--no-psqlrc` (`-X`) pour revenir à l'environnement par défaut et éviter de polluer l'affichage :

```
$ psql -X -f script.sql
$ psql -X -At -c 'SELECT name, setting FROM pg_settings ;'
```

1.5 ÉCRITURE DE SCRIPTS SHELL

- Script SQL
- Script Shell
- Exemple sauvegarde

1.5.1 Exécuter un script SQL avec psql

- Avec `-c` :

```
psql -c 'SELECT * FROM matable' -c 'SELECT fonction(123)' ;
```

- Avec un script :

```
psql -f nom_fichier.sql
```

```
psql < nom_fichier.sql
```

- Depuis `psql` :
  - `\i nom_fichier.sql`

L'option `-c` permet de spécifier du SQL en ligne de commande sans avoir besoin de faire un script. Plusieurs ordres seront enchaînés dans la même session.

Il est généralement préférable d'enregistrer les ordres dans des fichiers si on veut les exécuter plusieurs fois sans se tromper. L'option `-f` est très utile dans ce cas. La redirection avec `<` est une alternative répandue.

1.5.2 Gestion des transactions

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

Par défaut, `psql` est en mode « autocommit », c'est-à-dire que tous les ordres SQL sont automatiquement validés après leur exécution.

Pour exécuter une suite d'ordres SQL dans une seule et même transaction, il faut soit ouvrir explicitement une transaction avec `BEGIN;` et la valider avec `COMMIT;` ou l'annuler avec `ROLLBACK;`. Les autres outils clients sont généralement dans ce même cas.

L'ordre

```
=# \set AUTOCOMMIT off
```

a pour effet d'insérer systématiquement un `BEGIN` avant chaque ordre s'il n'y a pas déjà une transaction ouverte ; il faudra valider ensuite avec `COMMIT` **avant** de déconnecter. Il est déconseillé de changer le comportement par défaut (`on`), même s'il peut désorienter au premier abord des personnes ayant connu une base de données concurrente.

Une autre possibilité est d'utiliser `psql -1` ou `psql --single-transation` : les ordres sont automatiquement encadrés d'un `BEGIN` et d'un `COMMIT`. La présence d'ordres `BEGIN`, `COMMIT` ou `ROLLBACK` explicites modifiera ce comportement en conséquence.

1.5.3 Écrire un script SQL

- Attention à l'encodage des caractères
  - `\encoding`
  - `SET client_encoding`
- Écriture des requêtes

L'encodage a moins d'importance depuis qu'UTF-8 s'est généralisé, mais il y a encore parfois des problèmes dans de vieilles bases ou certains vieux outils.

Rappelons que les bases modernes devraient toutes utiliser l'encodage UTF-8 (c'est le défaut).

`\encoding [ENCODAGE]` permet, en l'absence d'argument, d'afficher l'encodage du client. En présence d'un argument, il permet de préciser l'encodage du client.

Exemple :

```
postgres=# \encoding
UTF8
postgres=# \encoding LATIN9
postgres=# \encoding
LATIN9
```

Cela a le même effet que d'utiliser l'ordre SQL `SET client_encoding TO LATIN9;`.

En terme de présentation, il est commun d'écrire les mots clés SQL en majuscules et d'écrire les noms des objets et fonctions manipulés dans les requêtes en minuscule. Le langage SQL est un langage au même titre que Java ou PHP, la présentation est importante pour la lisibilité des requêtes, même si les variations personnelles sont nombreuses.

1.5.4 Les blocs anonymes

- Bloc procédural anonyme en PL/pgSQL :

```
DO $$
DECLARE r record;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        RAISE NOTICE 'Analyze %.%', r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END$$;
```

Les blocs anonymes sont utiles pour des petits scripts ponctuels qui nécessitent des boucles ou du conditionnel, voire du transactionnel, sans avoir à créer une fonction ou une procédure. Ils ne renvoient rien. Ils sont habituellement en PL/pgSQL mais tout langage procédural installé est possible.

L'exemple ci-dessus lance un `ANALYZE` sur toutes les tables où les statistiques n'ont pas été calculées d'après la vue système, et donne aussi un exemple de SQL dynamique. Le résultat est par exemple :

```
NOTICE:  Analyze public.pgbench_history
NOTICE:  Analyze public.pgbench_tellers
NOTICE:  Analyze public.pgbench_accounts
NOTICE:  Analyze public.pgbench_branches
DO
Temps : 141,208 ms
```

(Pour ce genre de SQL dynamique, si l'on est sous `psql` , il est souvent plus pratique d'utiliser `\gexec`[19].)

Noter que les ordres constituent une transaction unique, à moins de rajouter des `COMMIT` ou `ROLLBACK` explicitement (ce n'est autorisé qu'à partir de la version 11).

[19] https://docs.postgresql.fr/current/app-psql.html#R2-APP-PSQL-4

1.5.5 Utiliser des variables

```
\set nom_table 'ma_table'
SELECT * FROM :"nom_table";

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

\prompt 'invite' nom_variable
\unset variable

psql -v VARIABLE=valeur
```

psql permet de manipuler des variables internes personnalisées dans les scripts. Ces variables peuvent être particulièrement utiles pour passer des noms d'objets ou des termes à utiliser dans une requête par le biais des options de ligne de commande (`-v variable=valeur`).

Noter la position des guillemets quand la variable est une chaîne de caractères !

Exemple :

Une fois connecté à la base **pgbench**, on déclare deux variables propres au client :

```
pgbench=# \set nomtable pgbench_accounts
pgbench=# \set taillemini 1000000
```

Elles apparaissent bien dans la liste des variables :

```
pgbench=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'pgbench'
…
nomtable = 'pgbench_accounts'
taillemini = '1000000'
```

Elles s'utilisent ainsi :

```
SELECT pg_relation_size (:'nomtable') ;

 pg_relation_size
------------------
        134299648

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class
WHERE relkind = 'r' AND pg_relation_size (oid) > :taillemini ;
```

```
     relname      | pg_size_pretty
------------------+----------------
 pgbench_accounts | 128 MB
```

La substitution s'effectue bien au niveau du client. Si l'on trace tout au niveau du serveur, ces requêtes apparaissent :

```
SELECT pg_relation_size ('pgbench_accounts') ;

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class WHERE relkind = 'r'
AND pg_relation_size (oid) > 1000000 ;
```

1.5.6 Gestion des erreurs

- Ignorer les erreurs dans une transaction
  - `ON_ERROR_ROLLBACK`
- Gérer des erreurs SQL en shell
  - `ON_ERROR_STOP`

La variable interne `ON_ERROR_ROLLBACK` n'a de sens que si elle est utilisée dans une transaction. Elle peut prendre trois valeurs :

- `off` (défaut) ;
- `on` ;
- `interactive`.

Lorsque `ON_ERROR_ROLLBACK` est à `on`, psql crée un `SAVEPOINT` systématiquement avant d'exécuter une requête SQL. Ainsi, si la requête SQL échoue, psql effectue un `ROLLBACK TO SAVEPOINT` pour annuler cette requête. Sinon il relâche le `SAVEPOINT`.

Lorsque `ON_ERROR_ROLLBACK` est à `interactive`, le comportement de psql est le même seulement si il est utilisé en interactif. Si psql exécute un script, ce comportement est désactivé. Cette valeur permet de se protéger d'éventuelles fautes de frappe.

Utiliser cette option n'est donc pas neutre, non seulement en terme de performances, mais également en terme d'intégrité des données. Il ne faut donc pas utiliser cette option à la légère.

Enfin, la variable interne `ON_ERROR_STOP` a deux objectifs : arrêter l'exécution d'un script lorsque psql rencontre une erreur et retourner un code retour shell différent de 0. Si cette variable reste à `off`, psql retournera toujours la valeur 0 même s'il a rencontré une erreur dans l'exécution d'une requête.

Une fois activée, psql retournera un code d'erreur 3 pour signifier qu'il a rencontré une erreur dans l'exécution du script.

L'exécution d'un script qui comporte une erreur retourne le code 0, signifiant que psql a pu se connecter à la base de données et exécuté le script :

```
$ psql -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
0
```

Lorsque la variable `ON_ERROR_STOP` est activée, psql retourne un code erreur 3, signifiant qu'il a rencontré une erreur :

```
$ psql -v ON_ERROR_STOP=on -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
3
```

psql retourne les codes d'erreurs suivant au shell :

- 0 au shell s'il se termine normalement ;
- 1 s'il y a eu une erreur fatale de son fait (pas assez de mémoire, fichier introuvable) ;
- 2 si la connexion au serveur s'est interrompue ou arrêtée ;
- 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` a été initialisée.

1.5.7 Formatage des résultats

- Sortie simplifiée pour exploitation automatisée : `-XAt`
  - `-t` (`--tuples-only`)
  - `-A` (`--no-align`)
  - `-X` (`--no-psqlrc`)
  - séparateurs : `-F` (`--field-separator`) et `-R` (`--record-separator`)
- Formats HTML ou CSV
  - `-H | --html`
  - `--csv` (à partir de la version 12)

`psql` peut servir à afficher des rapports basiques et possède quelques options de formatage.

L'option `--csv` suffit à répondre à ce besoin, à partir d'un client en version 12 au moins.

S'il faut définir plus finement le format, il existe des options. `-A` impose une sortie non alignée des données. En ajoutant `-t`, qui supprime l'entête, et `-X`, qui demande à ignorer le `.psqlrc`, la sortie peut être facilement exploitée par des outils classiques comme `awk` ou `sed` :

```
$ psql -XAt -c 'select datname, pg_database_size(datname) from pg_database'
postgres|87311139
powa|765977379
template1|9028387
template0|8593923
pgbench|166134563
```

Le séparateur `|` peut être remplacé par un autre avec `-F`, ou un octet nul avec `-z`, et le retour chariot de fin de ligne par une chaîne définie avec `-R`, ou un octet nul avec `-0`.

`-H` permet une sortie en HTML pour une meilleure lisibilité par un humain.

1.5.8 Résultats en pivot (tableau croisé)

- `\crosstabview [colV [colH [colD [colonnedetriH]]]]`
- Exécute la requête en tampon
  - au moins 3 colonnes

Par exemple, pour voir les différents types de clients connectés aux bases (clients système inclus), le résultat n'est pas très lisible :

```
=# \pset null ø

=# SELECT datname, backend_type, COUNT(*) as nb  FROM pg_stat_activity
   GROUP BY 1,2
   ORDER BY datname NULLS LAST, backend_type ;

 datname  |         backend_type          | nb
----------+-------------------------------+----
 pgbench  | client backend                |  2
 postgres | client backend                |  1
 powa     | powa                          |  1
 ø        | archiver                      |  1
 ø        | autovacuum launcher           |  1
 ø        | background writer             |  1
 ø        | checkpointer                  |  1
 ø        | logical replication launcher  |  1
 ø        | pg_wait_sampling collector    |  1
```

```
 ø        | walwriter                    |  1
(10 lignes)
```

On peut le reformater ainsi :

```
=# \crosstabview backend_type datname

         backend_type          | postgres | ø | powa | pgbench
-------------------------------+----------+---+------+---------
 client backend                |        2 |   |      |       1
 walwriter                     |          | 1 |      |
 autovacuum launcher           |          | 1 |      |
 logical replication launcher  |          | 1 |      |
 powa                          |          |   |    1 |
 background writer             |          | 1 |      |
 archiver                      |          | 1 |      |
 checkpointer                  |          | 1 |      |
(9 lignes)
```

1.5.9 Formatage dans les scripts SQL

- Donner un titre au résultat de la requête
  - `\pset title 'Résultat de la requête'`
- Formater le résultat
  - `\pset format html` (ou `csv`...)
- Diverses options peu utilisées

Il est possible de réaliser des modifications sur le format de sortie des résultats de requête directement dans le script SQL ou en mode interactif dans psql.

Afficher `\pset` permet de voir ces différentes options. La complétion automatique après `\pset` affiche les paramètres et valeurs possibles.

Par exemple, l'option `format` est par défaut à `aligned` mais possède d'autres valeurs :

```
=# \pset format <TAB>

aligned          csv              latex            troff-ms         wrapped
asciidoc         html             latex-longtable  unaligned
```

D'autres options existent, peu utilisées. La liste complète des options de formatage et leur description est disponible dans la documentation de la commande psql[20].

[20] https://docs.postgresql.fr/current/app-psql.html

1.5.10 Scripts & Crontab

- `cron`
  - Attention aux variables d'environnement !
- Ou tout ordonnanceur

La planification d'un script périodique s'effectue de préférence avec les outils du système, donc sous Unix avec `cron` ou une de ses variantes, même si n'importe quel ordonnanceur peut convenir.

Avec `cron`, il faut se rappeler qu'à l'exécution d'un script, l'environnement de l'utilisateur n'est pas initialisé, ou plus simplement, les fichiers de personnalisation (par ex. `.bashrc`) de l'environnement ne sont pas lus. Seule la valeur `$HOME` est initialisée. Un script fonctionnant parfaitement dans une session peut échouer une fois planifié. Il faut donc prévoir ce cas, initialiser les variables d'environnement requises de façon adéquate, et bien tester.

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

```
#!/bin/bash

. ${HOME}/.bashrc

…
```

Rappelons que chaque utilisateur du système peut avoir ses propres crontab. L'utilisateur peut les visualiser avec la commande `crontab -l` et les éditer avec la commande `crontab -e`.

1.5.11 Exemple de script de sauvegarde

- Sauvegarder une base et classer l'archive (squelette) :

```
#!/bin/bash
# Paramètre : la base
t=$(mktemp)                       # fichier temporaire
pg_dump -Fc "$1" > $t             # sauvegarde
d=$(eval date +%d%m%y-%H%M%S)     # date
mv $t /backup/"${1}_${d}.dump" # déplacement
exit 0
```

- …et ajouter la gestion des erreurs !
- …et les surveiller

Il est délicat d'écrire un script fiable. Ce script d'exemple possède plusieurs problèmes potentiels si le paramètre (la base) manque, si la sauvegarde échoue, si l'espace disque manque dans `/tmp`, si le déplacement échoue, si la partition cible n'est pas montée…

Parmi les outils existants, nous évoquerons notamment pg_back lors des sauvegardes[21].

Par convention, un script doit renvoyer 0 s'il s'est déroulé correctement.

[21] https://dali.bo/i1_html

1.6 OUTILS GRAPHIQUES

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

Il existe de nombreux outils graphiques permettant d'administrer des bases de données PostgreSQL. Certains sont libres, d'autres propriétaires. Certains sont payants, d'autres gratuits. Ils ont généralement les mêmes fonctionnalités de base, mais vont se distinguer sur certaines fonctionnalités un peu plus avancées comme l'import et l'export de données.

Nous allons étudier ici plusieurs outils proposés par la communauté, temBoard, pgAdmin.

1.6.1 temBoard

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

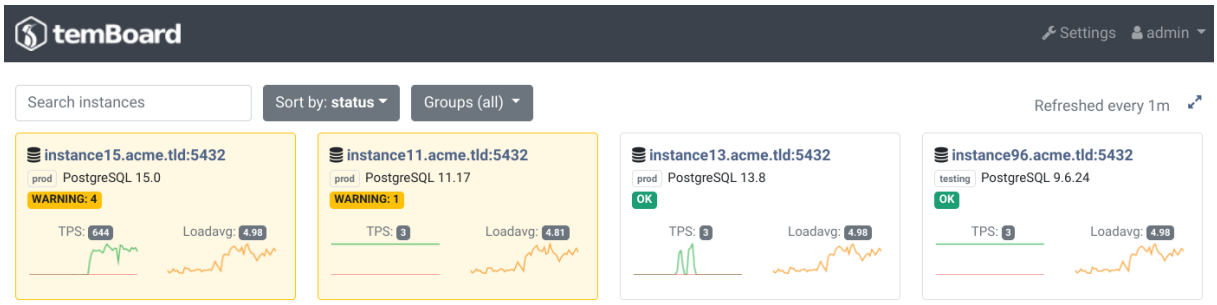
1.6.2 temBoard - PostgreSQL Remote Control

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

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

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

1.6.3 temBoard - Vue parc



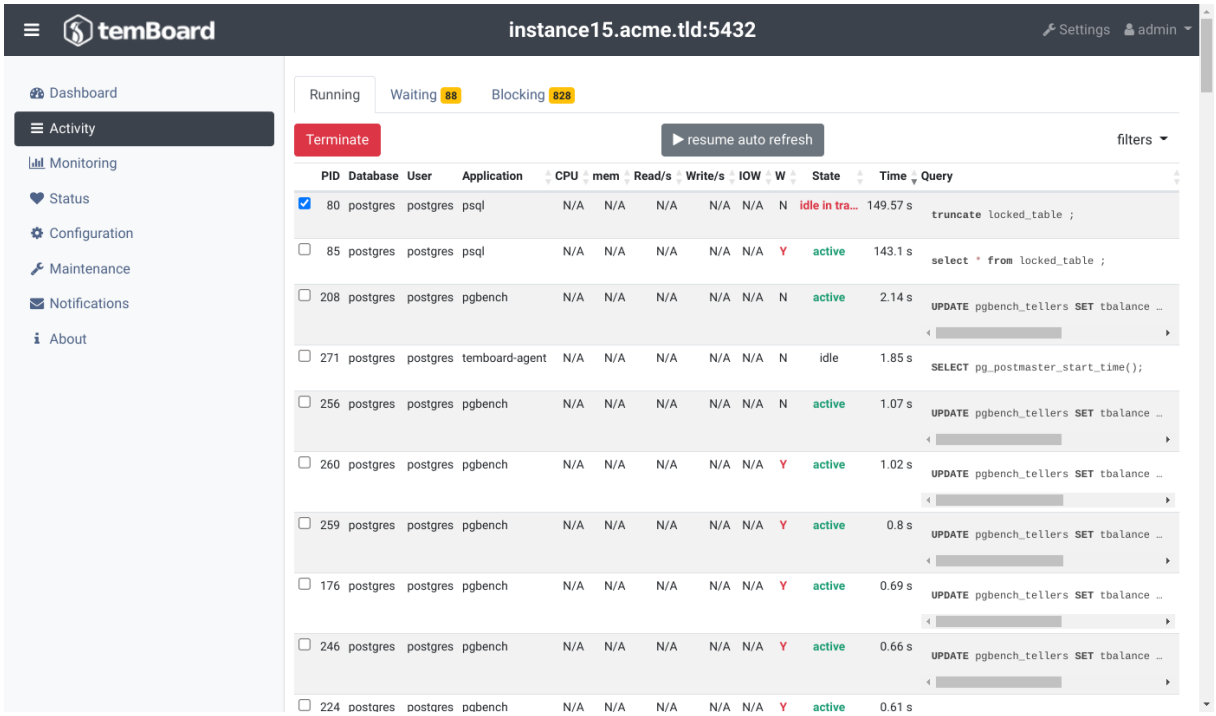
temboard fournit un tableau de bord global.

temBoard peut administrer plusieurs centaines d'instances.

1.6.4 temBoard - Tableau de bord

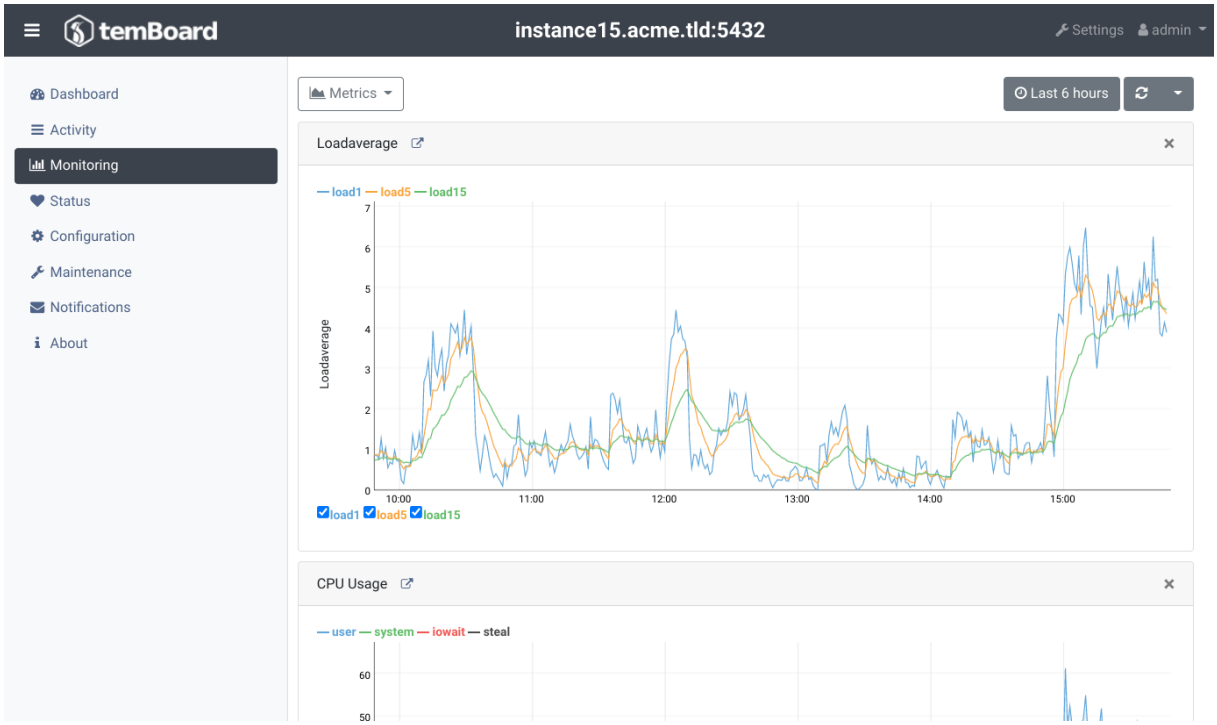
temboard présente un tableau de bord par instance.

1.6.5 temBoard - Activity



La section **Activity** permet de lister toutes les requêtes courantes (**Running**), les requêtes bloquées (**Waiting**) ou bloquantes (**Blocking**). Il est possible à partir de cette vue de terminer une session.

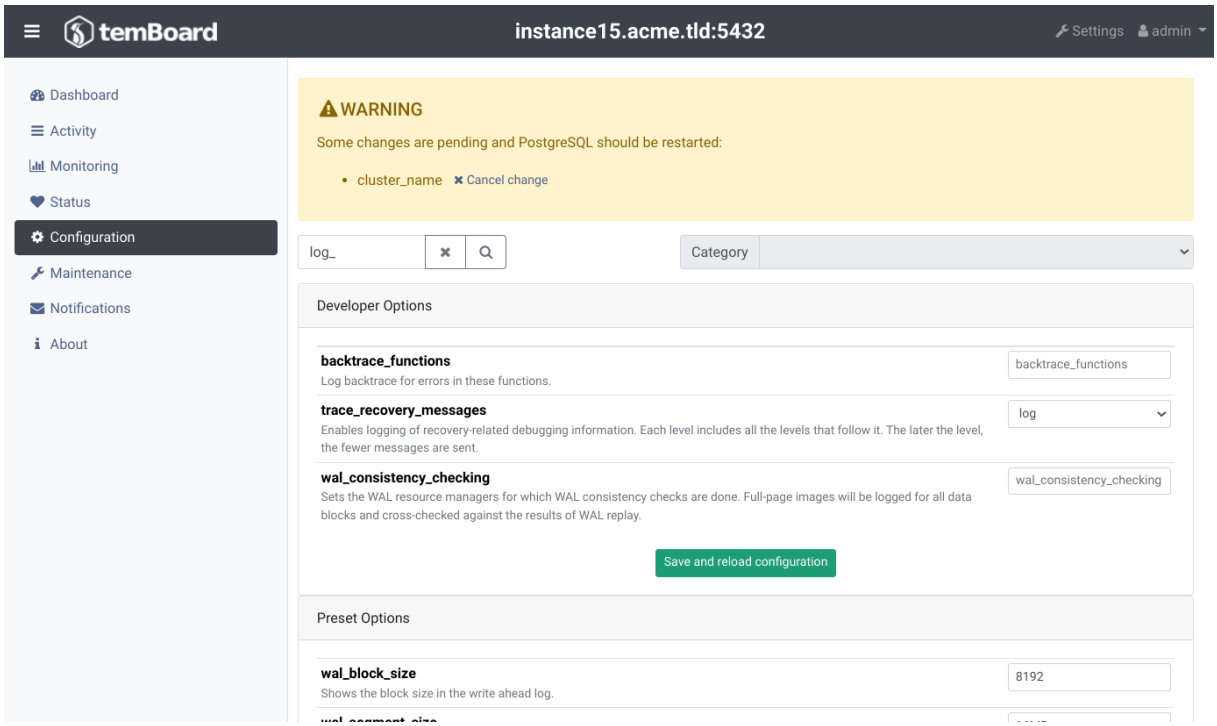
1.6.6 temBoard - Supervision



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

temBoard inclut un système d'alerte par courriel et SMS lorsqu'une métrique dépasse un seuil.

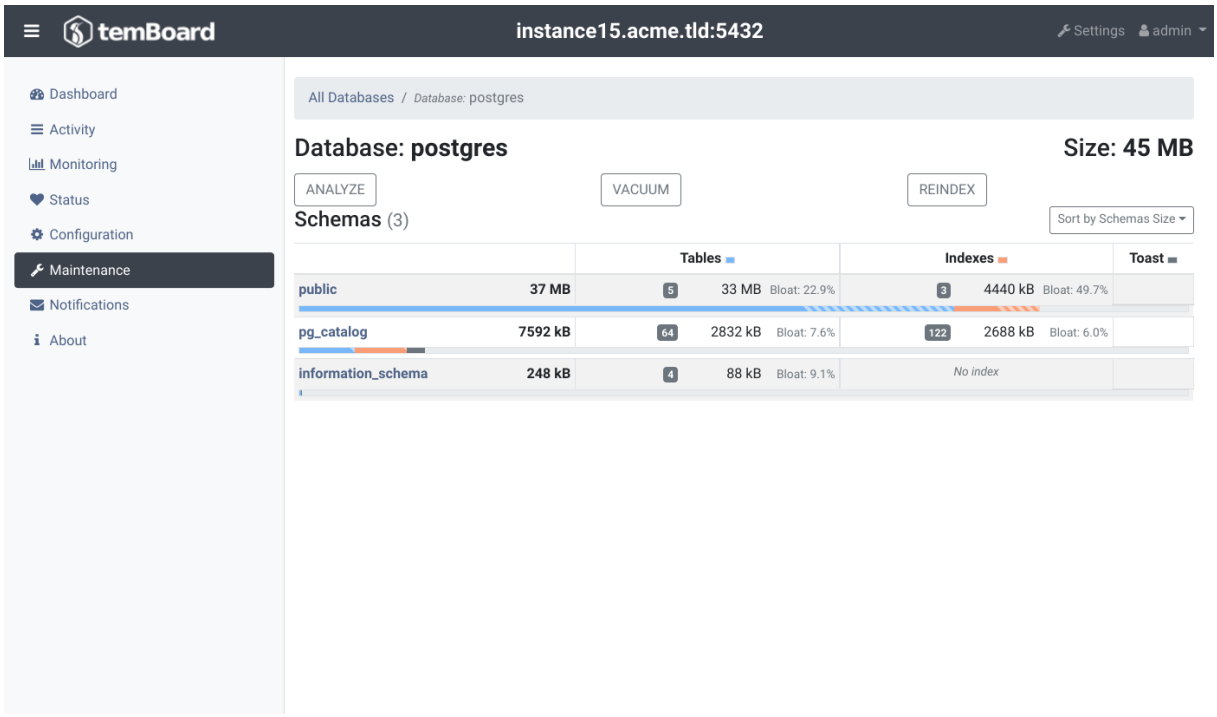
1.6.7 temBoard - Configuration



La section *Configuration* permet naviguer dans les paramètres de PostgreSQL de modifier ces paramètres

Suivant les cas, il sera proposé de recharger la configuration ou de redémarrer l'instance pour appliquer ces changements.

1.6.8 temBoard - Maintenance



temBoard estime la fragmentation du stockage des tables et des index.

Une interface permet de visualiser la fragmentation et d'agir dessus avec une granularité fine.

1.6.9 pgAdmin 4

- https://www.pgadmin.org
- Application web
- Licence : PostgreSQL
- Multiplateforme, multilangue

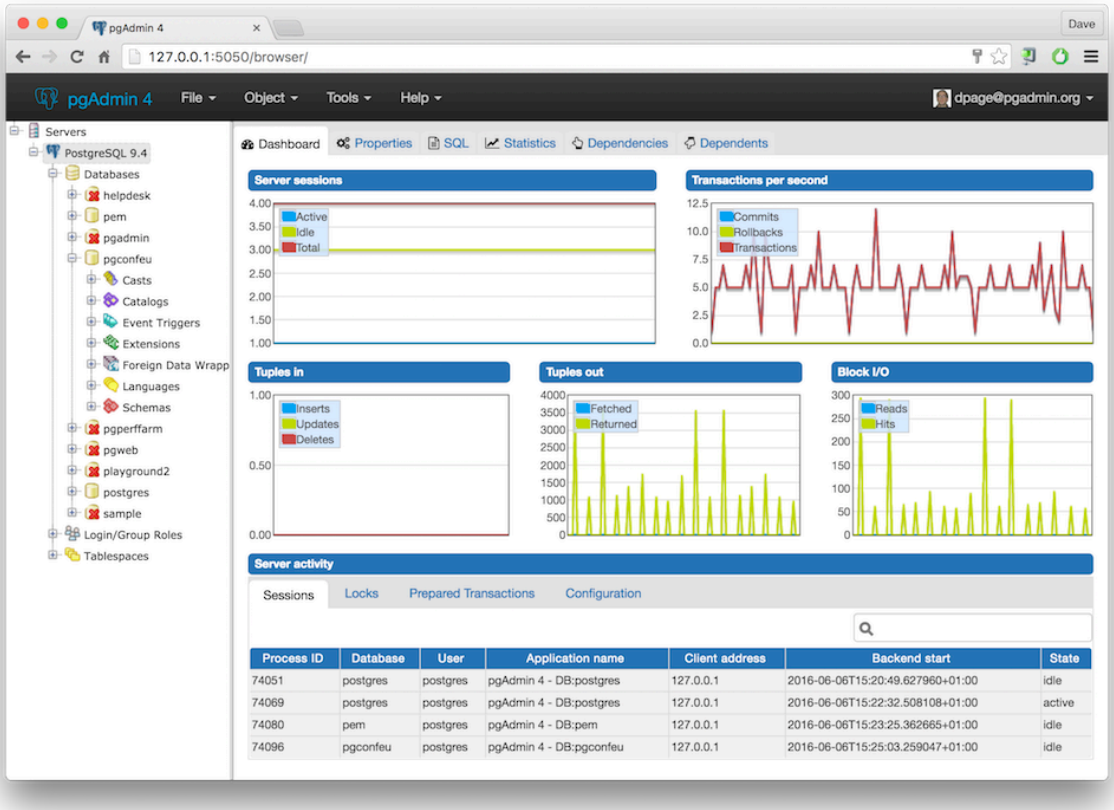
pgAdmin 4 est une application web, même si une version émulant un client lourd existe. Après un début difficile, le produit est à présent mature. Il reprend l'essentiel des fonctionnalités de pgAdmin III. Il est bien entendu compatible avec les dernières versions de PostgreSQL.

Il peut être déployé sur Windows et macOS X et bien sûr Linux, où il faudra utiliser les dépôts fournis par le projet[22], ou l'image docker.

Il est disponible sous licence PostgreSQL.

[22] https://www.pgadmin.org/download/

1.6.10 pgAdmin 4 : tableau de bord



Une des nouvelles fonctionnalités de pgAdmin 4 est l'apparition d'un tableau de bord remontant quelques métriques intéressantes. Il permet aussi la visualisation des géométries PostGIS, ce qui est parfois très utile.

1.6.11 DBeaver

- https://dbeaver.io
- Version Community sous Licence Apache 2.0
- Application Java Multiplateforme
- Version web *CloudBeaver* aussi disponible
- Supporte PostgreSQL (et ~ 80 autres SBGD)
- Version Pro payante et plus complète

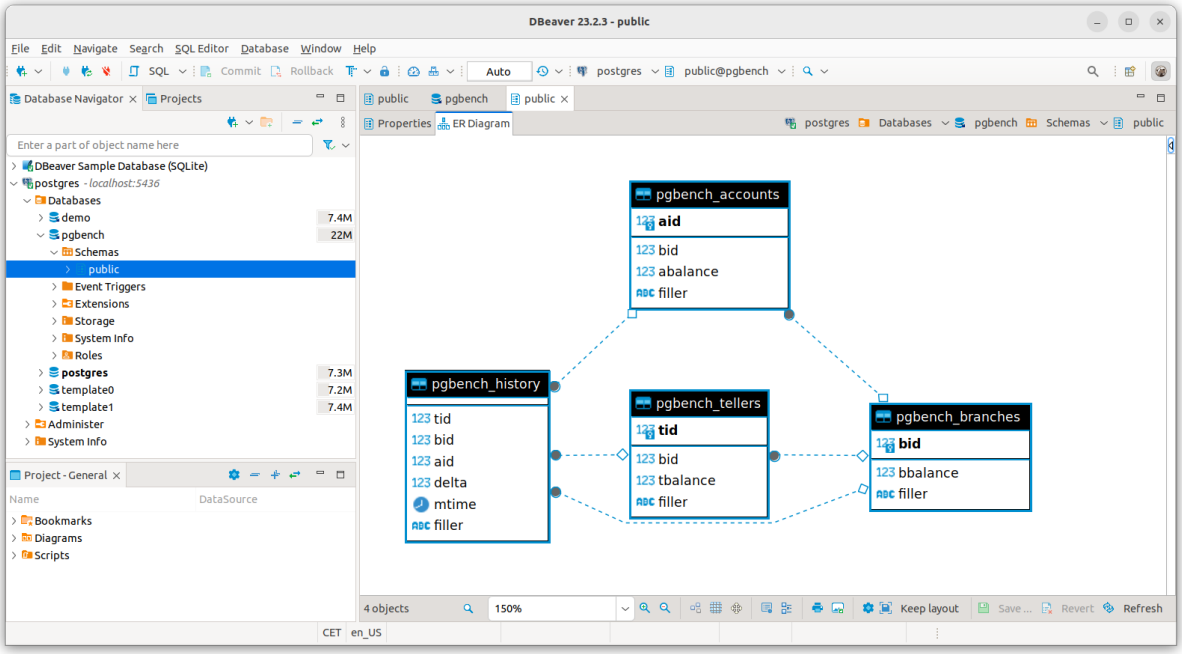
DBeaver est un outil graphique basé sur Eclipse. Il permet l'administration de plus de 80 SGBDs différents, y compris PostgreSQL. Le projet a sa propre entreprise depuis 2017 ainsi qu'une version payante plus complète. La version open-source *Community Edition* est disponible sous licence Apache 2.0. Une version web de DBeaver est disponible sous le nom de *CloudBeaver* (avec une page de démonstration publique[23]).

La version Pro, disponible sous différentes déclinaisons, propose des connecteurs propriétaires supplémentaires vers des systèmes de bases de données non relationnels (MongoDB, Redis, Cassandra, etc.), une gestion native des bases de données Cloud (AWS, Azure, etc.), des fonctionnalités dédiées aux entreprises ainsi qu'un support de la part de l'éditeur.

Il existe des paquets d'installation pour les systèmes d'exploitation les plus courants[24].

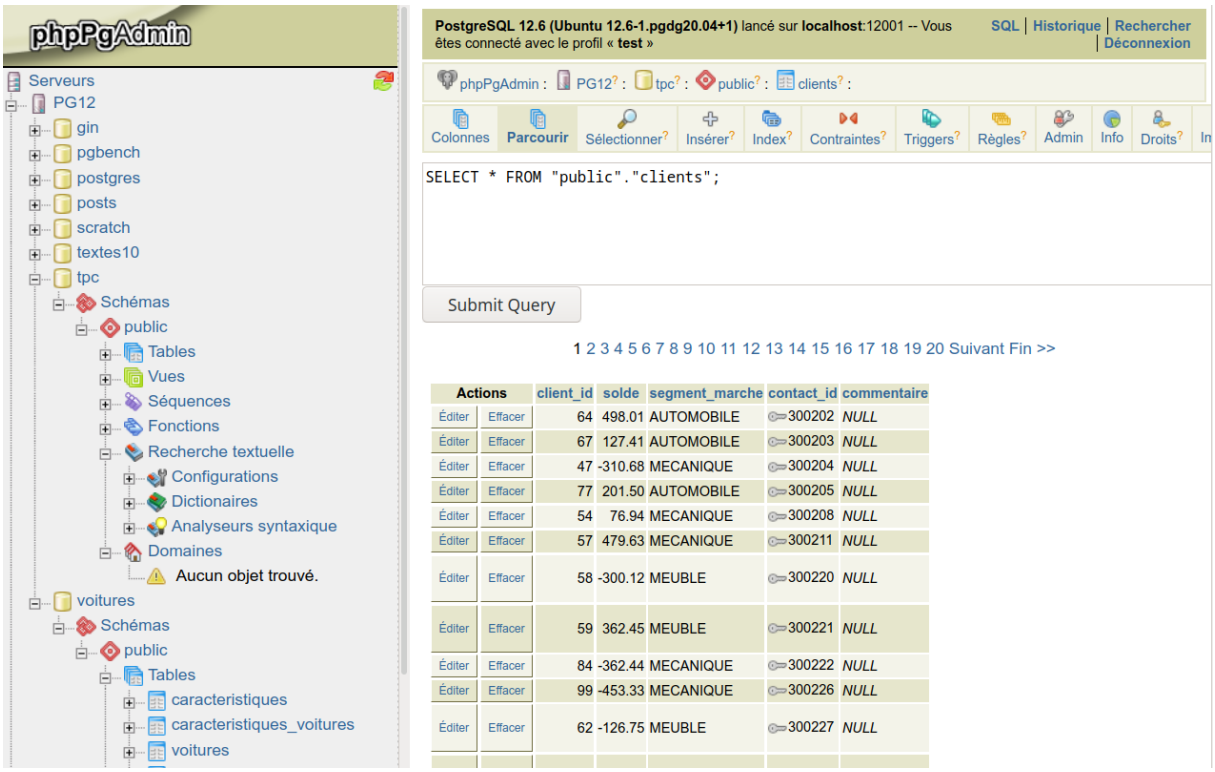
[23] https://demo.cloudbeaver.io/
[24] https://dbeaver.io/download/

1.6.12 DBeaver : fenêtre principale



La vue principale est divisée en deux sections, le côté gauche référence les différentes instances et le côté droit permet de consulter les différentes informations d'une instance en particulier. Ici, nous avons une vue entité-association des tables crées avec l'outil `pgbench` quand l'option `--foreign-keys` est spécifiée.

1.6.13 phpPgAdmin



1.6.14 phpPgAdmin : fonctionnalités

- Licence: GNU Public License
- Application web, simple
  - consultation, édition
  - sauvegarde, export
- Pérennité ?

PhpPgAdmin est une application web en PHP, légère et simple d'emploi, que l'on peut ouvrir à un utilisateur non informaticien pour y modifier des données.

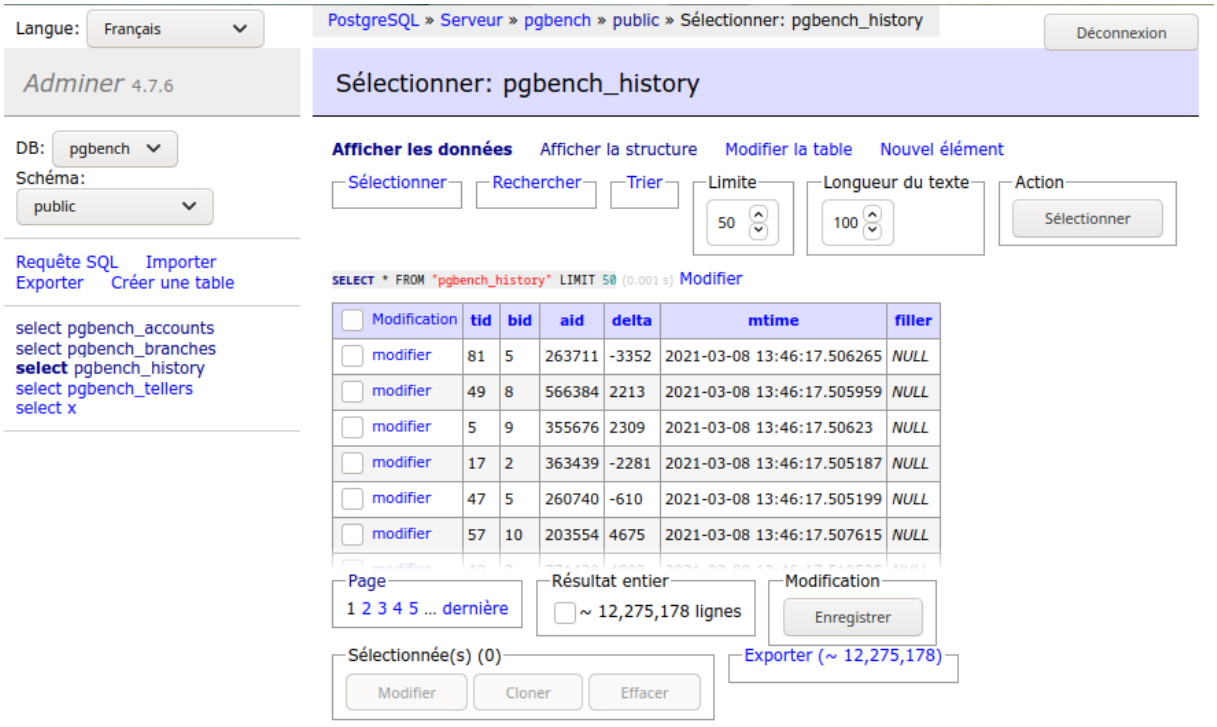
Le projet PhpPgAdmin a malheureusement des problèmes de maintenance récurrents. Le « nouveau » dépôt officiel[25] semble abandonné. Ce fork[26] semble le plus actif, et il sert de base entre autres aux

[25] https://github.com/phppgadmin/phppgadmin
[26] https://github.com/ReimuHakurei/phpPgAdmin

paquets Debian et OpenSUSE. Il est compatible avec PostgreSQL 16 et PHP 8.2.

Notre conseil est de préférer la version web de pgAdmin 4, qui est beaucoup plus lourd, mais plus puissant et plus pérenne. Du moins s'il convient au public voulu.

1.6.15 AdminerEvo



1.6.16 AdminerEvo : fonctionnalités

- https://docs.adminerevo.org/
- Application web pour utilisateurs
- Basique mais simple & efficace
- Et simple : 1 fichier PHP
- Multibases, multilangues
- Licence : Apache License ou GPL 2

AdminerEvo prend la suite de Adminer[27] qui n'est plus maintenu et ne doit plus être utilisé. C'est une

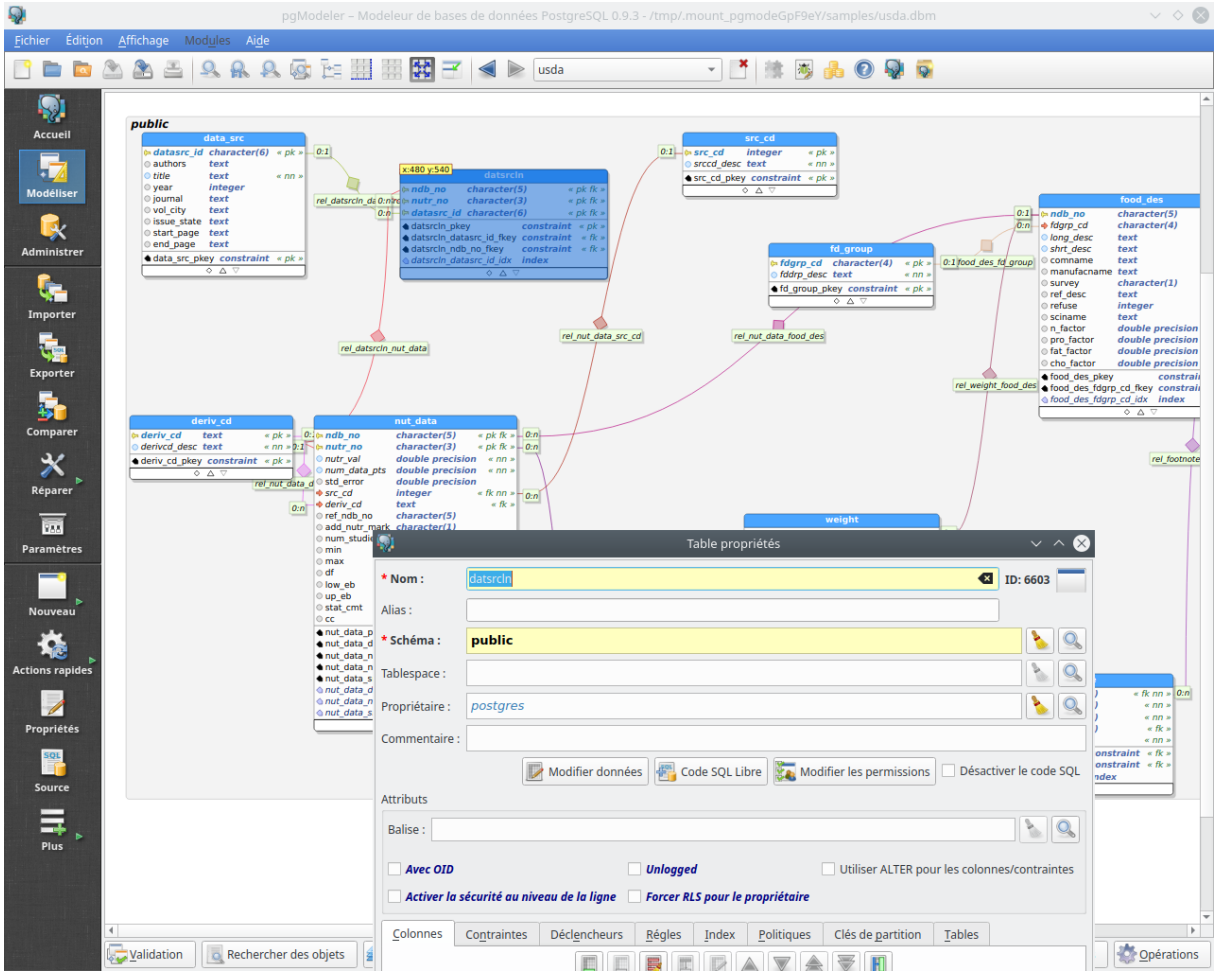
[27] https://www.adminer.org/

application web à destination des utilisateurs, pouvant gérer plusieurs types de bases, dont PostgreSQL.

Il utilise en un unique fichier PHP (éventuellement personnalisable par CSS). On récupérera simplement le dépôt Github[28].

L'interface est très basique. Elle peut sembler datée, voire primitive, mais elle est très simple et regroupe efficacement l'essentiel des fonctionnalités. C'est un candidat au remplacement de phpPgAdmin pour des utilisateurs non techniques.

1.6.17 pgModeler



[28] https://github.com/adminerevo/adminerevo

1.6.18 pgModeler

- Site officiel : https://pgmodeler.io/
- Licence : GPLv3
- Modélisation de base de données
- Fonctionnalité d'import export
- Comparaison de base

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

Il est à noter que, bien que le projet soit libre, son installation par les sources peut être laborieuse, et les paquets ne sont pas forcément disponibles. L'équipe de développement propose des paquets binaires à prix modique.

1.7 CONCLUSION

- Les outils en ligne de commande sont « rustiques » mais puissants
- Ils peuvent être remplacés par des outils graphiques
- En cas de problème, il est essentiel de les maîtriser.

1.7.1 Questions

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/de_quiz

1.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/de_solutions.

But :

- Acquérir certains automatismes dans l'utilisation de `psql`
- Créer des premières bases de données

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.

Il est important de bien discerner les différents utilisateurs impliqués au niveau système et PostgreSQL.

> Ouvrir plusieurs fenêtres ou consoles : au moins une avec l'utilisateur habituel (**dalibo** ici), une avec **root**, une avec l'utilisateur système **postgres**, une pour suivre le contenu des traces (`postgresql*.log`).

Nouvelle base bench :

> En tant qu'utilisateur système **postgres**, et avec l'utilitaire en ligne de commande `createdb`, créer une base de données nommée **bench** (elle appartiendra à **postgres**).

> Avec `psql`, se connecter à la base **bench** en tant qu'utilisateur **postgres**.

> Lister les bases de l'instance.

> Se déconnecter de PostgreSQL.

Voir les tables :

> Pour remplir quelques tables dans la base **bench**, on utilise un outil de *bench* livré avec PostgreSQL :
>
> ```
> /usr/pgsql-17/bin/pgbench -i --foreign-keys bench
> ```

> Quelle est la taille de la base après alimentation ?

Afficher la liste des tables de la base **bench** et leur taille.

Quelle est la structure de la table `pgbench_accounts` ?

Afficher l'ensemble des autres objets non système (index, séquences, vues…) de la base.

Nouvel utilisateur :

Toujours en tant qu'utilisateur système **postgres**, avec l'utilitaire `createuser`, créer un rôle **dupont** (il doit avoir l'attribut `LOGIN` !).

Sous psql, afficher la liste des rôles (utilisateurs).

Voir les objets système :

Dans la base **bench**, afficher l'ensemble des tables systèmes (schéma `pg_catalog`).

Afficher l'ensemble des vues systèmes (schéma `pg_catalog`).

Manipuler les données :

Le but est de créer une copie de la table `pgbench_tellers` de la base **bench** avec `CREATE TABLE AS`. Afficher l'aide de cette commande.

Créer une table `pgbench_tellers_svg`, copie de la table `pgbench_tellers`.

Sortir le contenu de la table `pgbench_tellers` dans un fichier `/tmp/pgbench_tellers.csv` (commande `\copy`).

Quel est le répertoire courant ? Sans quitter `psql`, se déplacer vers `/tmp/`, et en lister le contenu.

Afficher le contenu du fichier `/tmp/pgbench_tellers.csv` depuis `psql`.

> Créer un fichier `décompte.sql`, contenant 3 requêtes pour compter le nombre de lignes dans les 3 tables de **bench** :
>
> - Il devra écrire dans le fichier `/tmp/décompte.txt`.
> - Le faire exécuter par `psql`.

Détruire la base :

> Supprimer la base **bench**.

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

Les copies d'écran qui suivent proviennent d'un PostgreSQL 15 sous RockyLinux 8 paramétré en anglais. Elles peuvent différer légèrement selon la version de PostgreSQL, l'OS et la langue.

> Ouvrir plusieurs fenêtres ou consoles : au moins une avec l'utilisateur habituel (**dalibo** ici), une avec **root**, une avec l'utilisateur système **postgres**, une pour suivre le contenu des traces (`postgresql*.log`).

Pour devenir **root** :

```
$ sudo su -
```

Pour devenir **postgres** :

```
$ sudo -iu postgres
```

Pour voir le contenu des traces défiler, se connecter dans une nouvelle fenêtre à nouveau en tant que **postgres**, et aller chercher le fichier de traces. Sur Red Hat/CentOS, il est par défaut dans `$PGDATA/log` et son nom exact varie chaque jour :

```
$ sudo -iu postgres
$ ls -l /var/lib/pgsql/15/data/log
-rw-------. 1 postgres postgres 4462 Jan  6 11:12 postgresql-Fri.log
$ tail -f  /var/lib/pgsql/15/data/log/postgresql-Tue.log
```

Par défaut ne s'afficheront que peu de messages : arrêt/redémarrage, erreur de connexion… Laisser la fenêtre ouverte en arrière-plan ; elle servira à analyser les problèmes.

Nouvelle base bench :

> En tant qu'utilisateur système **postgres**, et avec l'utilitaire en ligne de commande `createdb`, créer une base de données nommée **bench** (elle appartiendra à **postgres**).

Si vous n'êtes pas déjà **postgres** :

```
$ sudo -iu postgres
```

```
$ createdb --echo bench
SELECT pg_catalog.set_config('search_path', '', false)
CREATE DATABASE bench;
```

Noter que `createdb` ne fait que générer un ordre SQL. On peut aussi aussi directement exécuter cet ordre depuis `psql` sous un compte superutilisateur.

> Avec `psql`, se connecter à la base **bench** en tant qu'utilisateur **postgres**.

```
$ psql -d bench -U postgres
psql (15.1)
Type "help" for help.

bench=#
```

> Lister les bases de l'instance.

```
bench=# \l
```

```
bench=# \l
                                                 List of databases
  Name    |  Owner   | Encoding |   Collate   |    Ctype    | …|   Access privileges
----------+----------+----------+-------------+-------------+--+-----------------------
 ↪  ---
bench     | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |
postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |
template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  | =c/postgres         +
          |          |          |             |             |  | postgres=CTc/postgres
template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  | =c/postgres         +
          |          |          |             |             |  | postgres=CTc/postgres
(4 rows)
```

(Deux colonnes indiquant à la source des librairies pour les collations ont été masquées.)

Noter que depuis le shell, le même résultat est renvoyé par :

```
$ psql -l
```

> Se déconnecter de PostgreSQL.

```
bench=# \q
```

(`exit` et **Ctrl-D** fonctionnent également.)

Voir les tables :

> Pour remplir quelques tables dans la base **bench**, on utilise un outil de *bench* livré avec PostgreSQL :
>
> `/usr/pgsql-17/bin/pgbench -i --foreign-keys bench`

L'outil est livré avec PostgreSQL, mais n'est pas dans les chemins par défaut sur Red Hat/CentOS/Rocky Linux.

La connexion doit fonctionner depuis n'importe quel compte système, il s'agit d'une connexion cliente tout comme `psql`.

> Quelle est la taille de la base après alimentation ?

`\l+` renvoie ceci :

```
$ psql
postgres=# \l+
                                                 List of databases
  Name    |  Owner   | Encoding |   Collate   |    …    |   Access privileges   |
 ↪  Size   | …
----------+----------+----------+-------------+---------+-----------------------+---
 ↪  ------+----
```

```
bench     | postgres | UTF8     | en_US.UTF-8 |         |                       | 23 MB
 ↪  | …
postgres  | postgres | UTF8     | en_US.UTF-8 |         |                        | 6477
 ↪  kB | …
template0 | postgres | UTF8     | en_US.UTF-8 |         | =c/postgres          +| 7297
 ↪  kB | …
          |          |          |             |         | postgres=CTc/postgres |
 ↪  | …
template1 | postgres | UTF8     | en_US.UTF-8 |         | =c/postgres          +| 7377
 ↪  kB | …
         |          |          |             |         | postgres=CTc/postgres |         |
```

La base **bench** fait donc 23 Mo.

> Afficher la liste des tables de la base **bench** et leur taille.

`\dt` affiche les tables, `\dt+` ajoute quelques informations dont la taille.

```
postgres=# \c bench
You are now connected to database "bench" as user "postgres".

bench=# \dt+
 Schema |       Name       | Type  |  Owner   | Persistence | Access method |  Size
 ↪  | Description
--------+------------------+-------+----------+-------------+---------------+-------
 ↪  --+-------------
 public | pgbench_accounts | table | postgres | permanent   | heap          | 13 MB
 ↪  |
 public | pgbench_branches | table | postgres | permanent   | heap          | 40 kB
 ↪  |
 public | pgbench_history  | table | postgres | permanent   | heap          | 0 bytes
 ↪  |
 public | pgbench_tellers  | table | postgres | permanent   | heap          | 40 kB
 ↪  |
```

(Il est courant d'utiliser `\d` et non `\dt`. S'afficheront alors aussi les vues et les séquences.)

> Quelle est la structure de la table `pgbench_accounts` ?

`\d` (voire `d+`) est sans doute un des ordres les plus utiles à connaître :

```
bench=# \d pgbench_accounts
             Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Foreign-key constraints:
    "pgbench_accounts_bid_fkey" FOREIGN KEY (bid) REFERENCES pgbench_branches(bid)
```

```
Referenced by:
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                                FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
```

La table a quatre colonnes : `aid`, `bid`, `abalance`, `filler`.

La première porte la clé primaire (et ne peut donc être à `NULL`).

La seconde est une clé étrangère pointant vers `pgbench_branches`.

La table `pgbench_history` porte une clé étrangère pointant vers la clé primaire de cette table.

> Afficher l'ensemble des autres objets non système (index, séquences, vues…) de la base.

Les index :

```
bench=# \di+
                                                    List of relations
Schema |         Name          | Type  |  Owner   |      Table       | Persistence | …
 ↪  |  Size   | …
-------+-----------------------+-------+----------+------------------+-------------
 ↪  +---+---------+--
public | pgbench_accounts_pkey | index | postgres | pgbench_accounts | permanent   |
 ↪  | 2208 kB |
public | pgbench_branches_pkey | index | postgres | pgbench_branches | permanent   |
 ↪  | 16 kB   |
public | pgbench_tellers_pkey  | index | postgres | pgbench_tellers  | permanent   |
 ↪  | 16 kB   |
```

Ces index sont ceux nécessités par les clés primaires.

Il n'y a ni séquence ni vue :

```
bench=# \ds
N'a trouvé aucune relation.
bench=# \dv
N'a trouvé aucune relation.
```

Nouvel utilisateur :

> Toujours en tant qu'utilisateur système **postgres**, avec l'utilitaire `createuser`, créer un rôle **dupont** (il doit avoir l'attribut `LOGIN` !).

```
$ createuser --echo --login dupont
SELECT pg_catalog.set_config('search_path', '', false)
CREATE ROLE dupont NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

> Sous psql, afficher la liste des rôles (utilisateurs).

`\du` affiche les rôles (sauf ceux système).

Il n'y a que le superutilisateur **postgres** (par défaut), et **dupont** créé tout à l'heure mais sans droit particulier :

```
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 dupont    |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Voir les objets système :

> Dans la base **bench**, afficher l'ensemble des tables systèmes (schéma `pg_catalog`).

```
bench=# \dt pg_catalog.*
                    List of relations
   Schema   |           Name            | Type  |  Owner
------------+---------------------------+-------+----------
 pg_catalog | pg_aggregate              | table | postgres
 pg_catalog | pg_aggregate              | table | postgres
 pg_catalog | pg_am                     | table | postgres
…
 pg_catalog | pg_statistic              | table | postgres
…
(64 rows)
```

Notons que pour afficher uniquement les tables système, on préférera le raccourci `\dtS`.

> Afficher l'ensemble des vues systèmes (schéma `pg_catalog`).

Certaines des vues ci-dessous sont très utiles dans la vie de DBA :

```
bench=# \dv pg_catalog.*
                       List of relations
   Schema   |               Name                | Type |  Owner
------------+-----------------------------------+------+----------
 pg_catalog | pg_available_extension_versions   | view | postgres
 pg_catalog | pg_available_extensions           | view | postgres
 pg_catalog | pg_backend_memory_contexts        | view | postgres
 pg_catalog | pg_config                         | view | postgress
 pg_catalog | pg_cursors                        | view | postgres
 pg_catalog | pg_file_settings                  | view | postgres
 pg_catalog | pg_group                          | view | postgres
 pg_catalog | pg_hba_file_rules                 | view | postgres
 pg_catalog | pg_ident_file_mappings            | view | postgres
 pg_catalog | pg_indexes                        | view | postgres
 pg_catalog | pg_locks                          | view | postgres
 pg_catalog | pg_matviews                       | view | postgres
…
 pg_catalog | pg_roles                          | view | postgres
…
 pg_catalog | pg_settings                       | view | postgres
 pg_catalog | pg_shadow                         | view | postgres
…
 pg_catalog | pg_stat_activity                  | view | postgres
…
 pg_catalog | pg_stat_archiver                  | view | postgres
```

```
…
 pg_catalog | pg_stat_database                 | view | postgres
…
 pg_catalog | pg_stat_progress_analyze        | view | postgres
 pg_catalog | pg_stat_progress_basebackup     | view | postgres
 pg_catalog | pg_stat_progress_cluster        | view | postgres
 pg_catalog | pg_stat_progress_copy           | view | postgres
 pg_catalog | pg_stat_progress_create_index   | view | postgres
 pg_catalog | pg_stat_progress_vacuum         | view | postgres
…
 pg_catalog | pg_stat_replication             | view | postgres
 pg_catalog | pg_stat_replication_slots       | view | postgres
…
 pg_catalog | pg_statio_all_tables            | view | postgres
…
 pg_catalog | pg_statio_user_indexes          | view | postgres
 pg_catalog | pg_statio_user_sequences        | view | postgres
 pg_catalog | pg_statio_user_tables           | view | postgres
 pg_catalog | pg_stats                        | view | postgres
…
 (75 rows)
```

Là encore, `\dvS` est un équivalent pour les tables systèmes.

Manipuler les données :

> Le but est de créer une copie de la table `pgbench_tellers` de la base **bench** avec `CREATE TABLE AS`. Afficher l'aide de cette commande.

```
bench=# \h CREATE TABLE AS
postgres=# \c bench
You are now connected to database "bench" as user "postgres".
bench=# \h CREATE TABLE AS
Command:     CREATE TABLE AS
Description: define a new table from the results of a query
Syntax:
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
 ↪  table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]

URL: https://www.postgresql.org/docs/15/sql-createtableas.html
```

> Créer une table `pgbench_tellers_svg`, copie de la table `pgbench_tellers`.

```
bench=# CREATE TABLE pgbench_tellers_svg AS SELECT * FROM pgbench_tellers ;
SELECT 10
```

> Sortir le contenu de la table `pgbench_tellers` dans un fichier `/tmp/pgbench_tellers.csv` (commande `\copy`).

```
bench=# \copy pgbench_tellers TO '/tmp/pgbench_tellers.csv'
COPY 10
```

Rappelons que la commande `\copy` est propre à `psql` (outil client), et est exécutée sur le client, avec l'accès au système de fichiers du client. `\copy` ne doit pas être confondue avec `COPY`, commande similaire exécutée par le serveur, et n'ayant accès qu'au système de fichiers du serveur. Il est important de bien connaître la distinction même si le client est utilisé ici sur le serveur avec l'utilisateur système **postgres**.

> Quel est le répertoire courant ? Sans quitter `psql`, se déplacer vers `/tmp/`, et en lister le contenu.

Le répertoire courant est celui en cours quand `psql` a été lancé. Selon les cas, ce peut être `/home/dalibo`, `/var/lib/pgsql/`… On peut se déplacer avec `\cd`.

```
bench=# \! pwd
/home/dalibo

bench=# \cd /tmp

bench=# \! ls
pgbench_tellers.csv
systemd-private-1b08135528d846088bb892f5a82aec9e-bolt.service-1hjHUH
…
bench=#
```

> Afficher le contenu du fichier `/tmp/pgbench_tellers.csv` depuis `psql`.

Son contenu est le suivant :

```
bench=# \! cat /tmp/pgbench_tellers.csv
1       1       0       \N
2       1       0       \N
3       1       0       \N
4       1       0       \N
5       1       0       \N
6       1       0       \N
7       1       0       \N
8       1       0       \N
9       1       0       \N
10      1       0       \N
```

On aurait pu l'ouvrir avec un éditeur de texte ou n'importe quel autre programme présent sur le client :

```
bench=# \! vi /tmp/pgbench_tellers.csv
```

> Créer un fichier `décompte.sql`, contenant 3 requêtes pour compter le nombre de lignes dans les 3 tables de **bench** :
>
> - Il devra écrire dans le fichier `/tmp/décompte.txt`.
> - Le faire exécuter par `psql`.

Le fichier doit contenir ceci :

```
\o /tmp/décompte.txt
SELECT COUNT(*) FROM pgbench_accounts ;
SELECT COUNT(*) FROM pgbench_tellers ;
SELECT COUNT(*) FROM pgbench_branches ;
```

La première ligne ordonne d'écrire la sortie des ordres dans le fichier indiqué.

Il peut être appelé par :

```
$ psql -d bench -f /tmp/décompte.sql
```

ou :

```
$ psql -d bench < /tmp/décompte.sql
```

Vérifier ensuite le contenu de `/tmp/décompte.txt`.

Détruire la base :

> Supprimer la base **bench**.

Depuis la ligne de commande du système d'exploitation, en tant qu'utilisateur système **postgres** :

```
$ dropdb --echo bench
SELECT pg_catalog.set_config('search_path', '', false);
DROP DATABASE bench;
```

Alternativement, si l'on est connecté en tant que superutilisateur à l'instance (pas sous la base à supprimer !) :

```
postgres=# DROP DATABASE bench ;
DROP DATABASE
```

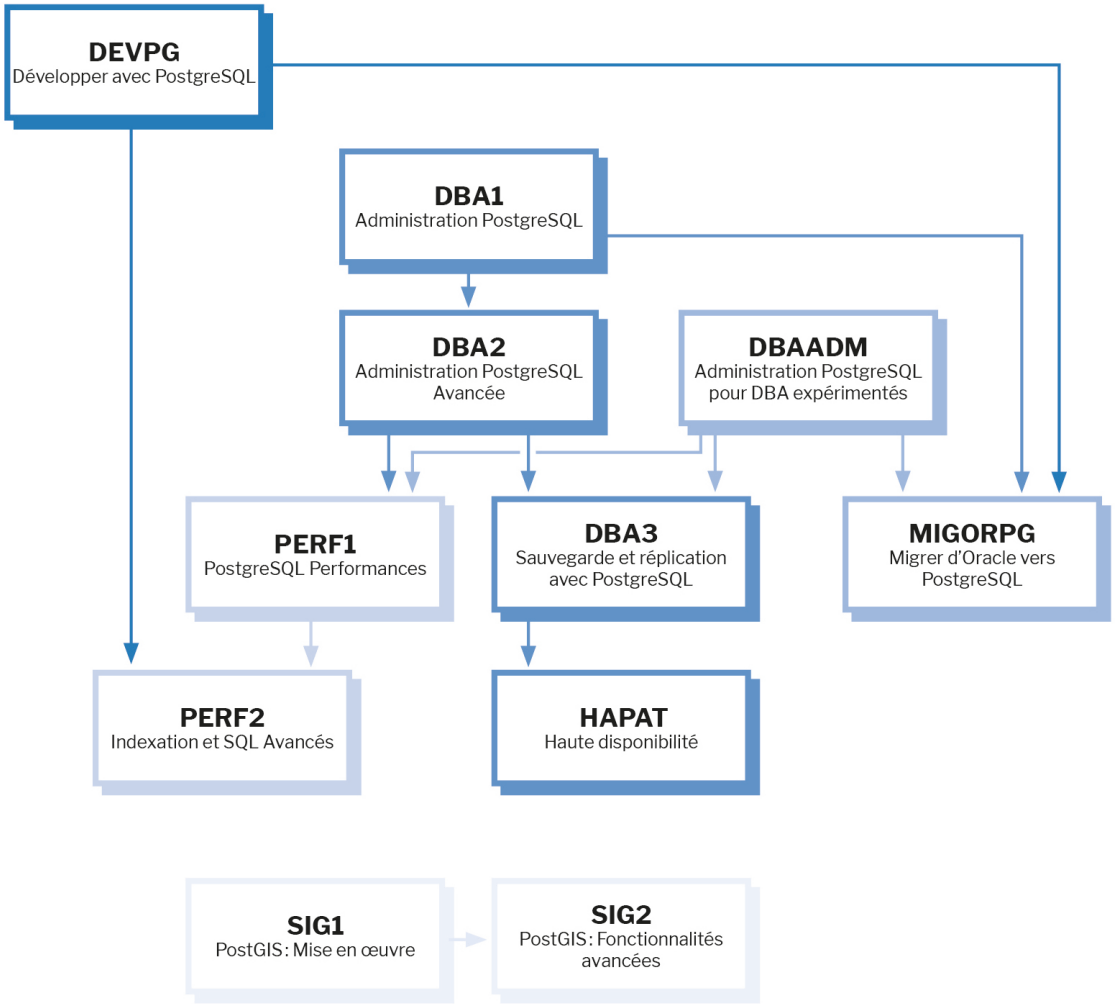
Noter l'absence de demande de confirmation !

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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