Module B

Installation de PostgreSQL

24.04

Table des matières

Sur ce document

Formation	Module B
Titre	Installation de PostgreSQL
Révision	24.04
PDF	https://dali.bo/b_pdf
EPUB	https://dali.bo/b_epub
HTML	https://dali.bo/b_html
Slides	https://dali.bo/b_slides
TP	https://dali.bo/b_tp
TP (solutions)	https://dali.bo/b_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 :

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

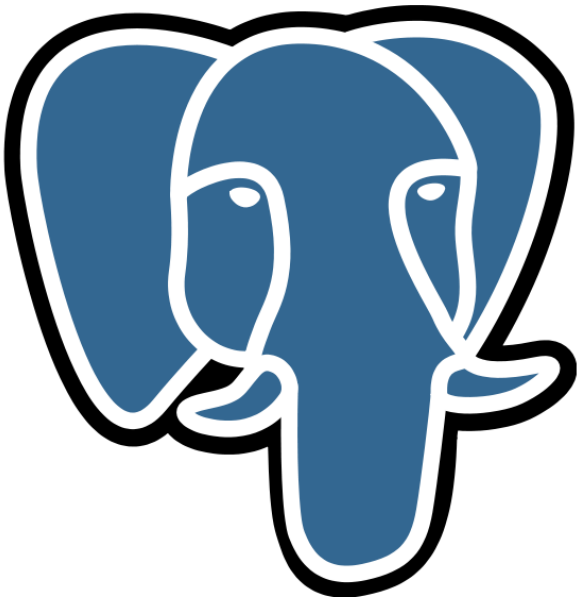
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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

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

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

1/ Installation de PostgreSQL



1.1 INTRODUCTION

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

Il existe trois façons d'installer PostgreSQL :

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

Nous allons maintenant détailler chaque façon d'installer PostgreSQL.

1.2 PRÉ-REQUIS MINIMAUX POUR UNE INSTANCE POSTGRESQL

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

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

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

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

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

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

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

1.3 INSTALLATION À PARTIR DES SOURCES

Étapes :

- Téléchargement
- Vérification des prérequis
- Compilation
- Installation

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

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

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

1.3.1 Téléchargement

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

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

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

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

1.3.2 Phases de compilation/installation

- Processus standard :

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

La compilation de PostgreSQL suit un processus classique.

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

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

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

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

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

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

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

```
make
```

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

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

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

```
sudo make install
```

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

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

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

1.3.3 Options pour ./configure

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

```
$ pg_config --configure
```

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

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

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

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

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

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

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

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

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

1.3.4 Tests de non régression

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

```
$ make check
```

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

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

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

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

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

1.3.5 Création de l'utilisateur

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

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

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

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

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

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

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

1.3.6 Création du répertoire de données de l'instance

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

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

Répertoire :

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

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

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

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

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

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

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

À ce propos, voir :

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

Lancement de initdb :

Voici ce qu'affiche cette commande :

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

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

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

Data page checksums are enabled.

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

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

Success. You can now start the database server using:

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

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

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

Authentification par défaut :

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

Sommes de contrôle :

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

Emplacement des journaux :

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

Taille des segments :

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

1.3.7 Lancement et arrêt

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

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

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

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

(Re)démarrage et arrêt :

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

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

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

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

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

Rechargement de la configuration :

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

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

est équivalente à cet ordre SQL :

```
SELECT pg_reload_conf() ;
```

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

1.4 INSTALLATION À PARTIR DES PAQUETS LINUX

- Packages Debian
- Packages RPM

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

1.4.1 Paquets Debian officiels

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

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

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

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

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

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

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

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

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

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

Les exécutables sont installés dans :

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

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

- les données sont dans :

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

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

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

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

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

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

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

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

1.4.2 Paquets Debian : spécificités

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

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

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

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

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

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

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

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

1.4.3 Paquets Debian communautaires

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

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

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

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

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

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

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

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

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

et la version choisie de PostgreSQL sera immédiatement installable :

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

1.4.4 Paquets Red Hat communautaires : yum.postgresql.org

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

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

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

1.4.5 Paquets Red Hat communautaires : installation

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

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

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

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

1.4.6 Paquets Red Hat communautaires : spécificités

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5 UTILISER POSTGRESQL DANS UN CONTENEUR

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

docker exec -it pg16 -U postgres
```

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

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

Exemple :

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

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

```
docker pull postgres
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exemple avec docker compose :

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

Fichier `docker-compose.yml`

```
version: '3.3'
```

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

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

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

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

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

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

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

Limites de l'utilisation de PostgreSQL sous docker :

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

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

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

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

1.6 INSTALLATION SOUS WINDOWS

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

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

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

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

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

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

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

1.6.1 Installeur graphique

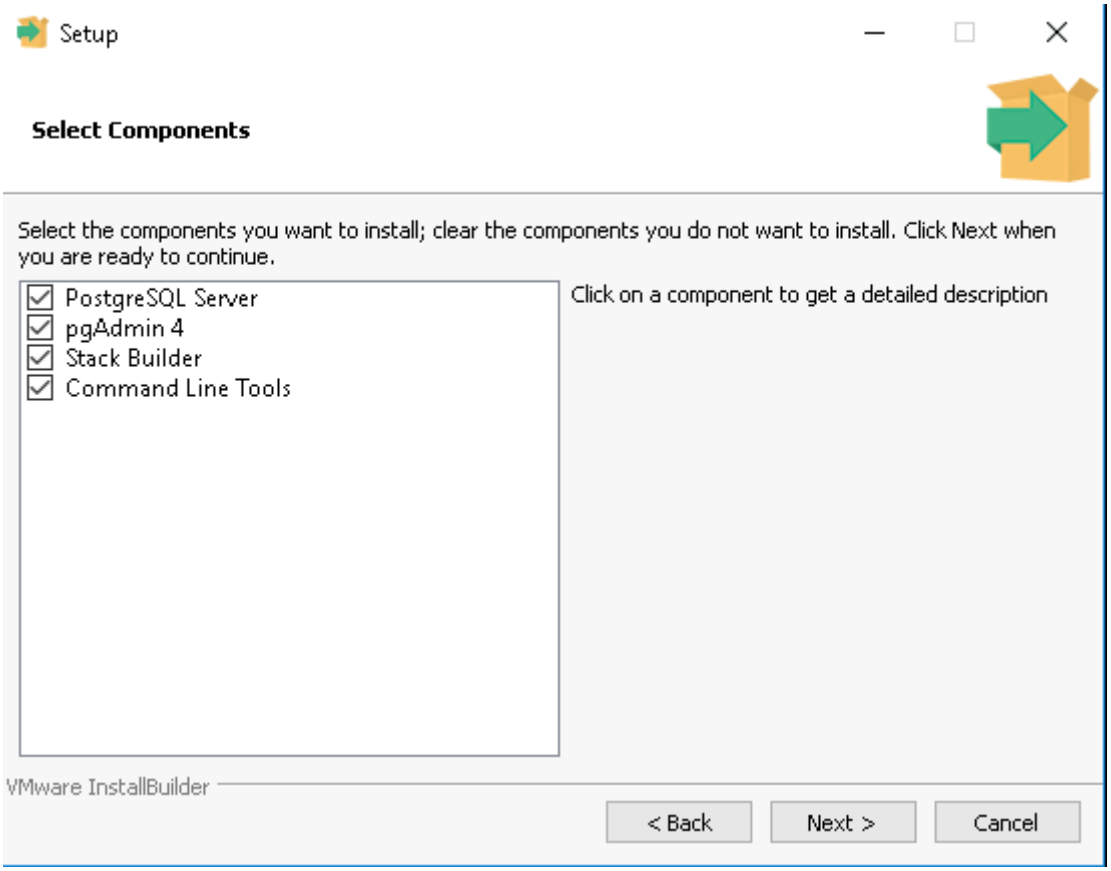


Figure 1/ .1: Installeur graphique - bienvenue

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

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

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

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

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

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

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

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

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

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

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

1.7 INDUSTRIALISATION AVEC PGLIFT

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

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

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

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

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

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

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

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

Tous les composants satellites supportés par pglift sont des logiciels libres. Le projet pglift est lui aussi nativement open source, sous licence GPLv3. Son développement se passe en public sur https://gitl

[16] https://pglift.readthedocs.io/en/latest/
[17] https://pgbackrest.org/
[18] https://github.com/prometheus-community/postgres_exporter
[19] https://prometheus.io/
[20] https://temboard.readthedocs.io/en/latest/
[21] https://powa.readthedocs.io/en/latest/#
[22] https://patroni.readthedocs.io/en/latest

ab.com/dalibo/pglift/ pour l'API Python et l'interface en ligne de commande et sur https://gitlab.com/dalibo/pglift-ansible/ pour la collection Ansible `dalibo.pglift`.

Références :

- Présentation sur le blog Dalibo[23] (Denis Laxalde)
- Documentation[24]

1.7.1 pglift : fichier de configuration

```yaml
prefix: /srv # fichier /etc/pglift/settings.yaml
postgresql:
  auth:
    host: scram-sha-256
prometheus:
  execpath: /usr/bin/prometheus-postgres-exporter
pgbackrest:
  repository:
    mode: path
    path: /srv/pgsql-backups
powa: {}
systemd: {}
rsyslog: {}
```

À coté de PostgreSQL, l'instance inclut un ensemble d'outils nécessaires à son utilisation. L'intégration de ces outils satellites est configurée localement via un fichier YAML `settings.yaml`.

Ce fichier définit comment les différents composants de l'instance sont configurés, installés et exécutés. Il permet de aussi de définir quels composants satellites facultatifs supportés par pgLift sont inclus dans l'instance. Si un élément est listé dans ce fichier sans paramètre associé, il sera exploité dans sa configuration par défaut.

Dans l'exemple ci-dessus, temBoard et Patroni ne sont pas installés, et PoWA est laissé à sa configuration par défaut.

[23] https://blog.dalibo.com/2023/10/17/pglift-intro.html
[24] https://pglift.readthedocs.io/

1.7.2 pglift : exemples de commandes

- Initialisation

```
pglift instance create main --pgbackrest-stanza=main
```

- Modification de configuration

```
pglift pgconf -i main set log_connections=on
```

- Sauvegarde physique

```
pglift instance backup main
```

- Utilisation des outils de l'instance

```
pglift instance exec main -- psql
pglift instance exec main -- pgbackrest info
```

Interface impérative en ligne de commande :

La commande suivante permet la création d'une instance pglift :

```
$ pglift instance create main --pgbackrest-stanza=main
INFO     initializing PostgreSQL
INFO     configuring PostgreSQL authentication
INFO     configuring PostgreSQL
INFO     starting PostgreSQL 16-main
INFO     creating role 'powa'
INFO     creating role 'prometheus'
INFO     creating role 'backup'
INFO     altering role 'backup'
INFO     creating 'powa' database in 16/main
INFO     creating extension 'btree_gist' in database powa
INFO     creating extension 'pg_qualstats' in database powa
INFO     creating extension 'pg_stat_statements' in database powa
INFO     creating extension 'pg_stat_kcache' in database powa
INFO     creating extension 'powa' in database powa
INFO     configuring Prometheus postgres_exporter 16-main
INFO     configuring pgBackRest stanza 'main' for
         pg1-path=/srv/pgsql/16/main/data
INFO     creating pgBackRest stanza main
INFO     starting Prometheus postgres_exporter 16-main
```

L'instance pglift inclut l'instance PostgreSQL ainsi que l'ensemble des modules définis dans le fichier de configuration `settings.yaml`. pglift gère aussi l'intégration au système avec `systemd` ou `rsyslog` comme dans notre exemple. Tout ceci fonctionne sans privilège **root** pour une meilleure séparation des responsabilités et une meilleure sécurité.

pglift permet de récupérer l'état d'une instance à un moment donné :

```
$ pglift instance get main -o json
{
  "name": "main",
  "version": "16",
  "port": 5432,
  "settings": {
    "unix_socket_directories": "/run/user/1000/pglift/postgresql",
    "shared_buffers": "1 GB",
    "wal_level": "replica",
    "archive_mode": true,
    "archive_command": "/usr/bin/pgbackrest --config-path=/etc/pgbackrest \
    --stanza=main --pg1-path=/srv/pgsql/16/main/data archive-push %p",
    "effective_cache_size": "4 GB",
    "log_destination": "syslog",
    "logging_collector": true,
    "log_directory": "/var/log/postgresql",
    "log_filename": "16-main-%Y-%m-%d_%H%M%S.log",
    "syslog_ident": "postgresql-16-main",
    "cluster_name": "main",
    "lc_messages": "C",
    "lc_monetary": "C",
    "lc_numeric": "C",
    "lc_time": "C",
    "shared_preload_libraries": "pg_qualstats, pg_stat_statements, pg_stat_kcache"
  },
  "data_checksums": false,
  "locale": "C",
  "encoding": "UTF8",
  "standby": null,
  "state": "started",
  "pending_restart": false,
  "wal_directory": "/srv/pgsql/16/main/wal",
  "prometheus": {
    "port": 9187
  },
  "data_directory": "/srv/pgsql/16/main/data",
  "powa": {},
  "pgbackrest": {
    "stanza": "main"
  }
}
```

ou de modifier l'instance :

```
# activation du paramètres log_connections
$ pglift pgconf -i main set log_connections=on
INFO     configuring PostgreSQL
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
log_connections: None -> True
# changement du port prometheus
$ pglift instance alter main --prometheus-port 8188
INFO     configuring PostgreSQL
INFO     reconfiguring Prometheus postgres_exporter 16-main
```

```
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
INFO     starting Prometheus postgres_exporter 16-main
$ pglift instance get main # vérification
 name  version  port  data_checksums  locale  encoding  pending_restart  prometheus
  ↪  pgbackrest
 main  16       5432  False           C       UTF8      False            port: 8188
  ↪  stanza: main
```

Les instances et objets PostgreSQL peuvent être manipulés à l'aide des outils *natifs* de PostgreSQL depuis la ligne de commande :

```
$ pglift instance exec main -- pgbench -i bench
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.18 s (drop tables 0.00 s, create tables 0.01 s, ... vacuum 0.04 s, primary
 ↪  keys 0.05 s).
$ pglift instance exec main -- pgbench bench
pgbench (16.0 (Debian 16.0-1.pgdg120+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
number of failed transactions: 0 (0.000%)
latency average = 1.669 ms
initial connection time = 4.544 ms
tps = 599.125277 (without initial connection time)
```

Ceci s'applique aussi à des outils tiers, par exemple avec pgBackRest :

```
$ pglift instance exec main -- pgbackrest info
stanza: main
    status: ok
    cipher: none

    db (current)
        wal archive min/max (16): 000000010000000000000001/000000010000000000000007

        full backup: 20231016-092726F
            timestamp start/stop: 2023-10-16 09:27:26+02 / 2023-10-16 09:27:31+02
            wal start/stop: 000000010000000000000004 / 000000010000000000000004
            database size: 32.0MB, database backup size: 32.0MB
            repo1: backup set size: 4.2MB, backup size: 4.2MB

        diff backup: 20231016-092726F_20231016-092821D
            timestamp start/stop: 2023-10-16 09:28:21+02 / 2023-10-16 09:28:24+02
            wal start/stop: 000000010000000000000007 / 000000010000000000000007
            database size: 54.5MB, database backup size: 22.6MB
```

```
            repo1: backup set size: 6MB, backup size: 1.8MB
            backup reference list: 20231016-092726F
```

Le tutoriel de la ligne de commande[25] de la documentation recense tous les exemples d'utilisation des commandes usuelles de *pglift*.

Interface déclarative avec Ansible :

pglift comporte une collection de modules Ansible, sous l'espace de noms `dalibo.pglift`.Voici un exemple de playbook illustrant ses capacités :

```
- name: Set up database instances
hosts: dbserver
tasks:
  - name: main instance
    dalibo.pglift.instance:
      name: main
      state: started
      port: 5444
      settings:
        max_connections: 100
        shared_buffers: 1GB
        shared_preload_libraries: 'pg_stat_statements, passwordcheck'
      surole_password: ''
      pgbackrest:
        stanza: main
        password: ''
      prometheus:
        password: ''
        port: 9186
      roles:
        - name: admin
          login: true
          password: ''
          connection_limit: 10
          validity: '2025-01-01T00:00'
          in_roles:
            - pg_read_all_stats
            - pg_signal_backend
      databases:
        - name: main
          owner: admin
          settings:
            work_mem: 3MB
          extensions:
            - name: unaccent
              schema: public
```

Le module `dalibo.pglift.instance` permet de gérer l'instance, et les objets reliés comme des rôles ou des bases de données. Les données sensibles peuvent être prises en charge par Ansible vault[26]. Les modules Ansible permettent un contrôle plus important que la ligne de commandes, grâce aux

[25] https://pglift.readthedocs.io/en/latest/tutorials/cli.html
[26] https://docs.ansible.com/ansible/latest/vault_guide/index.html

champs imbriqués, pouvant inclure la configuration de l'instance, des bases de données, des extensions, etc.

Comme tout module Ansible en général, cette interface est complètement déclarative, idempotente et sans état. Ces modules fonctionnent avec d'autres modules Ansible, tels que `community.postgresql`.
Le tutoriel Ansible[27] de la documentation recense davantage d'exemples d'utilisation.

[27] https://pglift.readthedocs.io/en/latest/tutorials/ansible.html

1.8 PREMIERS RÉGLAGES

- Sécurité
- Configuration minimale
- Démarrage
- Test de connexion

1.8.1 Sécurité

- Politique d'accès :
  - pour l'utilisateur **postgres** système
  - pour le rôle **postgres**
- Règles d'accès à l'instance dans `pg_hba.conf`

Selon l'environnement et la politique de sécurité interne à l'entreprise, il faut potentiellement initialiser un mot de passe pour l'utilisateur système **postgres** :

```
$ passwd postgres
```

Sans mot de passe, il faudra passer par un système comme `sudo` pour pouvoir exécuter des commandes en tant qu'utilisateur **postgres**, ce qui sera nécessaire au moins au début.

Le fait de savoir qu'un utilisateur existe sur un serveur permet à un utilisateur hostile de tenter de forcer une connexion par force brute. Par exemple, ce billet de blog[28], montre que l'utilisateur **postgres** est dans le top 10 des logins attaqués.

La meilleure solution pour éviter ce type d'attaque est de ne pas définir de mot de passe pour l'utilisateur OS **postgres** et de se connecter uniquement par des échanges de clés SSH.

Il est conseillé de ne fixer aucun mot de passe pour l'utilisateur système. Il en va de même pour le rôle **postgres** dans l'instance. Une fois connecté au système, nous pourrons utiliser le mode d'authentification local `peer` pour nous connecter au rôle **postgres**. Ce mode permet de limiter la surface d'attaque sur son instance.

En cas de besoin d'accès distant en mode superutilisateur, il sera possible de créer des rôles supplémentaires avec des droits superutilisateur. Ces noms ne doivent pas être facile à deviner par de potentiels attaquants. Il faut donc éviter les rôles **admin** ou **root**.

[28] https://blog.sucuri.net/2013/07/ssh-brute-force-the-10-year-old-attack-that-still-persists.html

Si vous avez besoin de créer des mots de passe, ils doivent bien sûr être longs et complexes (par exemple en les générant avec les utilitaires `pwgen` ou `apg`).

Si vous avez utilisé l'installeur proposé par EnterpriseDB, l'utilisateur système et le rôle PostgreSQL ont déjà un mot de passe, celui demandé par l'installeur. Il n'est donc pas nécessaire de leur en configurer un autre.

Enfin, il est important de vérifier les règles d'accès au serveur contenues dans le fichier `pg_hba.conf`. Ces règles définissent les accès à l'instance en se basant sur plusieurs paramètres : utilisation du réseau ou du socket fichier, en SSL ou non, depuis quel réseau, en utilisant quel rôle, pour quelle base de données et avec quelle méthode d'authentification.

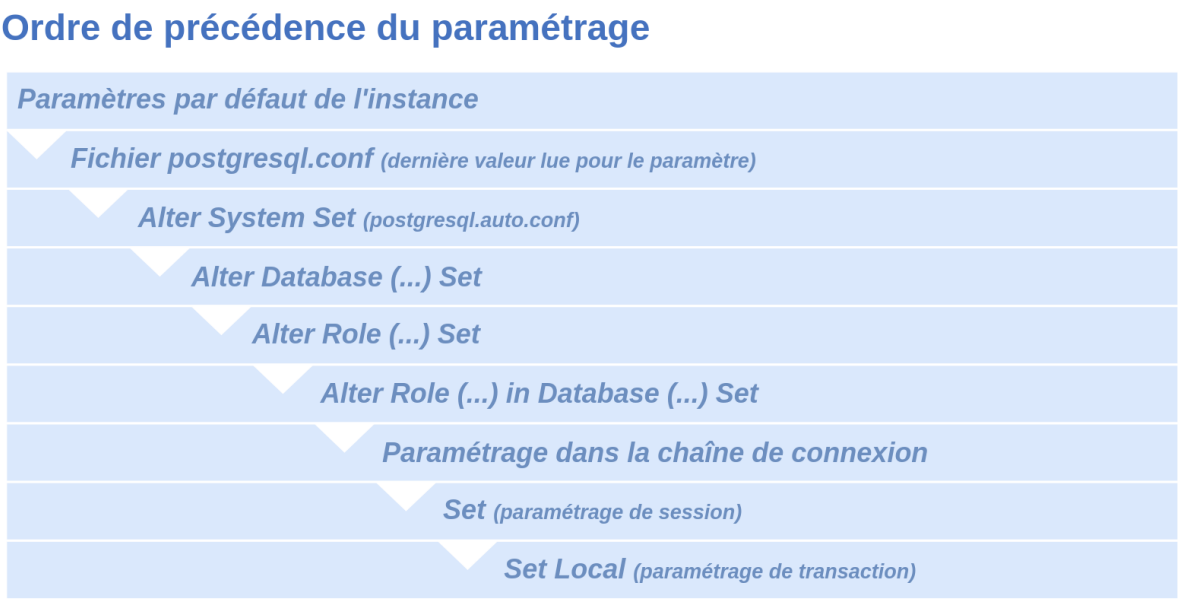
1.8.2 Configuration minimale

- Fichier `postgresql.conf`
- Configuration du moteur
- Plus de 300 paramètres
- Quelques paramètres essentiels

La configuration du moteur se fait via un seul fichier, le fichier `postgresql.conf`. Il se trouve généralement dans le répertoire des données du serveur PostgreSQL. Sous certaines distributions (Debian et affiliés principalement), il est déplacé dans `/etc/postgresql/`.

Ce fichier contient beaucoup de paramètres, plus de 300, mais seuls quelques-uns sont essentiels à connaître pour avoir une instance fiable et performante.

1.8.3 Précédence des paramètres



PostgreSQL offre une certaine granularité dans sa configuration, ainsi certains paramètres peuvent être surchargés par rapport au fichier `postgresql.conf`. Il est utile de connaître l'ordre de précédence. Par exemple, un utilisateur peut spécifier un paramètre dans sa session avec l'ordre `SET`, celui-ci sera prioritaire par rapport à la configuration présente dans le fichier `postgresql.conf`.

1.8.4 Configuration des connexions : accès au serveur

- `listen_addresses = '*'` (systématique)
- `port = 5432`
- `password_encryption` = `scram-sha-256` (v10+)

Ouvrir les accès :

Par défaut, une instance PostgreSQL n'écoute que sur l'interface de boucle locale (`localhost`) et pas sur les autres interfaces réseaux. Pour autoriser les connexions externes à PostgreSQL, il faut modifier le paramètre `listen_addresses`, en général ainsi :

```
listen_addresses = '*'
```

La valeur `*` est un joker indiquant que PostgreSQL doit écouter sur toutes les interfaces réseaux disponibles au moment où il est lancé. Il est aussi possible d'indiquer les interfaces, une à une, en les sé-

parant avec des virgules. Cette méthode est intéressante lorsqu'on veut éviter que l'instance écoute sur une interface donnée. Par prudence il est possible de se limiter aux interfaces destinées à être utilisées :

```
listen_addresses = 'localhost, 10.1.123.123'
```

La restriction par `listen_addresses` est un premier niveau de sécurité. Elle est complémentaire de la méthode plus fine par `pg_hba.conf`, par les IP clientes, utilisateur et base, qu'il faudra de toute façon déployer. De plus, modifier `listen_addresses` impose de redémarrer l'instance.

Port :

Le port par défaut des connexions TCP/IP est le `5432`. C'est la valeur traditionnelle et connue de tous les outils courants.

La modifier n'a d'intérêt que si vous voulez exécuter plusieurs instances PostgreSQL sur le même serveur (physique ou virtuel). En effet, plusieurs instances sur une même machine ne peuvent pas écouter sur le même couple adresse IP et port.

Une instance PostgreSQL n'écoute jamais que sur ce seul port, et tous les clients se connectent dessus. Il n'existe pas de notion de *listener* ou d'outil de redirection comme sur d'autres bases de données concurrentes, du moins sans outil supplémentaire (par exemple le pooler pgBouncer).

S'il y a plusieurs instances dans une même machine, elles devront posséder chacune un couple adresse IP/port unique. En pratique, il vaut mieux attribuer un port par instance. Bien sûr, PostgreSQL refusera de démarrer s'il voit que le port est déjà occupé.

Ne confondez pas la connexion à `localhost` (soit `::1` en IPv6 ou `127.0.0.1` en IPv4), qui utilise les ports TCP/IP, et la connexion dite `local`, passant par les *sockets* de l'OS (par défaut `/var/run/postgresql/.s.PGSQL.5432` sur les distributions les plus courantes). La distinction est importante dans `pg_hba.conf` notamment.

Chiffrage des mots de passe :

À partir de la version 10 et avant la version 14, le paramètre `password_encryption` est à modifier dès l'installation. Il définit l'algorithme de chiffrement utilisé pour le stockage des mots de passe. La valeur `scram-sha-256` permettra d'utiliser la nouvelle norme, plus sécurisée que l'ancien `md5`. Ce n'est plus nécessaire à partir de la version 14 car c'est la valeur par défaut. Avant toute modification, vérifiez quand même que vos outils clients sont compatibles. Au besoin, vous pouvez revenir à `md5` pour un utilisateur donné.

1.8.5 Configuration du nombre de connexions

- `max_connections = 100`
- 1 connexion = 1 processus serveur
- Compromis entre
  - CPU / nombre requêtes actives / RAM / complexité
- Danger si trop haut !
  - performances (même avec des connexions inactives)
  - risque de saturation
- Possibilité de réserver quelques connexions pour l'administration

Le nombre de connexions simultanées est limité par le paramètre `max_connections`. Dès que ce nombre est atteint, les connexions suivantes sont refusées avec un message d'erreur, et ce jusqu'à ce qu'un utilisateur connecté se déconnecte.

`max_connections` vaut par défaut 100, et c'est généralement suffisant en première intention.

Noter qu'il existe un paramètre `superuser_reserved_connections` (à 3 par défaut) qui réserve quelques connexions au superutilisateur pour qu'il puisse se connecter malgré une saturation. Depuis PostgreSQL 16, il existe un autre paramètre nommé `reserved_connections`, (à 0 par défaut) pour réserver quelques connexions aux utilisateurs à qui l'on aura attribué un rôle spécifique, nommé `pg_use_reserved_connections`. Ce peut être utile pour des utilisateurs non applicatifs (supervision et sauvegarde notamment) à qui l'on ne veut ou peut pas donner le rôle `SUPERUSER`.

Il peut être intéressant de diminuer `max_connections` pour interdire d'avoir trop de connexions actives. Cela permet de soulager les entrées-sorties, ou de monter `work_mem` (la mémoire de tri). À l'inverse, il est possible d'augmenter `max_connections` pour qu'un plus grand nombre d'utilisateurs ou d'applications puisse se connecter en même temps.

Au niveau mémoire, un processus consomme par défaut 2 Mo de mémoire vive. Cette consommation peut augmenter suivant son activité.

Il faut surtout savoir qu'à chaque connexion se voit associée un processus sur le serveur, processus qui n'est vraiment actif qu'à l'exécution d'une requête. Il s'agit donc d'arbitrer entre :

- le nombre de requêtes à exécuter à un instant T ;

- le nombre de CPU disponibles ;
- la complexité et la longueur des requêtes ;
- et même le nombre de processus que peut gérer l'OS.

L'établissement a un certain coût également. Il faut éviter qu'une application se connecte et se déconnecte sans cesse.

Il ne sert à rien d'autoriser des milliers de connexions s'il n'y a que quelques processeurs, ou si les requêtes sont lourdes. Si le nombre de requêtes réellement actives augmente fortement, le serveur peut s'effondrer. Restreindre les connexions permet de préserver le serveur, même si certaines connexions sont refusées.

Le paramétrage est compliqué par le fait qu'une même requête peut mobiliser plusieurs processeurs si elle est parallélisée. Certaines requêtes seront limitées par le CPU, d'autres par la bande passante des disques.

Enfin, même si une connexion inactive ne consomme pas de CPU et peu de RAM, elle a tout de même un impact. En effet, une connexion active va générer assez fréquemment ce qu'on appelle un snapshot (ou une image) de l'état des transactions de la base. La durée de création de ce snapshot dépend principalement du nombre de connexions, actives ou non, sur le serveur. Donc une connexion active consommera plus de CPU s'il y a 399 autres connexions, actives ou non, que s'il y a 9 connexions, actives ou non. Ce comportement est partiellement corrigé avec la version 14. Mais il vaut mieux éviter d'avoir des milliers de connexions ouvertes « au cas où ».

Intercaler un « pooler » comme pgBouncer entre les clients et l'instance peut se justifier dans certains cas :

- connexions/déconnexions très fréquentes ;
- centaines, voire milliers, de connexions généralement inactives ;
- limitation du nombre de connexions actives avec mise en attente au niveau du pooler (sans erreur).

1.8.6 Configuration de la mémoire partagée

- `shared_buffers = (?)GB`
  - 25 % de la RAM en première intention
  - max 40 %
  - complémentaire du cache OS

Shared buffers :

Chaque fois que PostgreSQL a besoin de lire ou d'écrire des données, il les charge d'abord dans son cache interne. Ce cache ne sert qu'à ça : stocker des blocs disques qui sont accessibles à tous les

processus PostgreSQL, ce qui permet d'éviter de trop fréquents accès disques car ces accès sont lents. La taille de ce cache dépend d'un paramètre appelé `shared_buffers`.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle[29] donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine dédiée de 32 Go de RAM, cela donne donc :

```
shared_buffers = 8GB
```

Le défaut de 128 Mo n'est donc pas adapté à un serveur sur une machine récente.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Modifier `shared_buffers` impose de redémarrer l'instance.

Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres comme `max_wal_size`, et de s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…).

1.8.7 Configuration : mémoire des processus

- `work_mem`
  - par processus, voire nœud
  - valeur très dépendante de la charge et des requêtes
  - fichiers temporaires vs saturation RAM
- × `hash_mem_multiplier`
- `maintenance_work_mem`
- Pas de limite stricte à la consommation mémoire des sessions
  - Augmenter prudemment & superviser

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Le paramètre le plus important est `work_mem`, qui définit la taille de la mémoire de travail d'un processus lors d'une requête, principalement lors d'opérations de tri : `ORDER BY`, certaines jointures, déduplication… Autre paramètre capital, `maintenance_work_mem` est la mémoire pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, ajouts de clé étrangère…

Cette mémoire est rendue immédiatement après la fin de l'ordre concerné.

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go sur les machines récentes, car il concerne des opérations lourdes rarement exécutées plusieurs fois simultanément. Monter au-delà est rare, mais peut avoir un intérêt dans les créations de très gros index.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit[30], Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

[30] https://dali.bo/j1_html#configuration-du-oom

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[31] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

À partir de PostgreSQL 13, un paramètre multiplicateur peut s'appliquer à certaines opérations particulières (le hachage, lors de jointures ou agrégations). Nommé `hash_mem_multiplier`, il vaut 1 par

[31] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

défaut en versions 13 et 14, et 2 à partir de la 15. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`.

Il existe d'autres paramètres influant sur les besoins en mémoires, moins importants pour une première approche.

1.8.8 Configuration des journaux de transactions 1/2

`fsync` = `on` (si vous tenez à vos données)

À chaque fois qu'une transaction est validée (`COMMIT`), PostgreSQL écrit les modifications qu'elle a générées dans les journaux de transactions.

Afin de garantir la durabilité, PostgreSQL effectue des écritures synchrones des journaux de transaction, donc une écriture physique des données sur le disque. Cela a un coût important sur les performances en écritures s'il y a de nombreuses transactions mais c'est le prix de la sécurité.

Le paramètre `fsync` permet de désactiver l'envoi de l'ordre de synchronisation au système d'exploitation. Ce paramètre **doit** rester à `on` en production. Dans le cas contraire, un arrêt brutal de la machine peut mener à la perte des journaux non encore enregistrés et à la corruption de l'instance. D'autres paramètres et techniques existent pour gagner en performance (et notamment si certaines données peuvent être perdues) sans pour autant risquer de corrompre l'instance.

1.8.9 Configuration des journaux de transactions 2/2

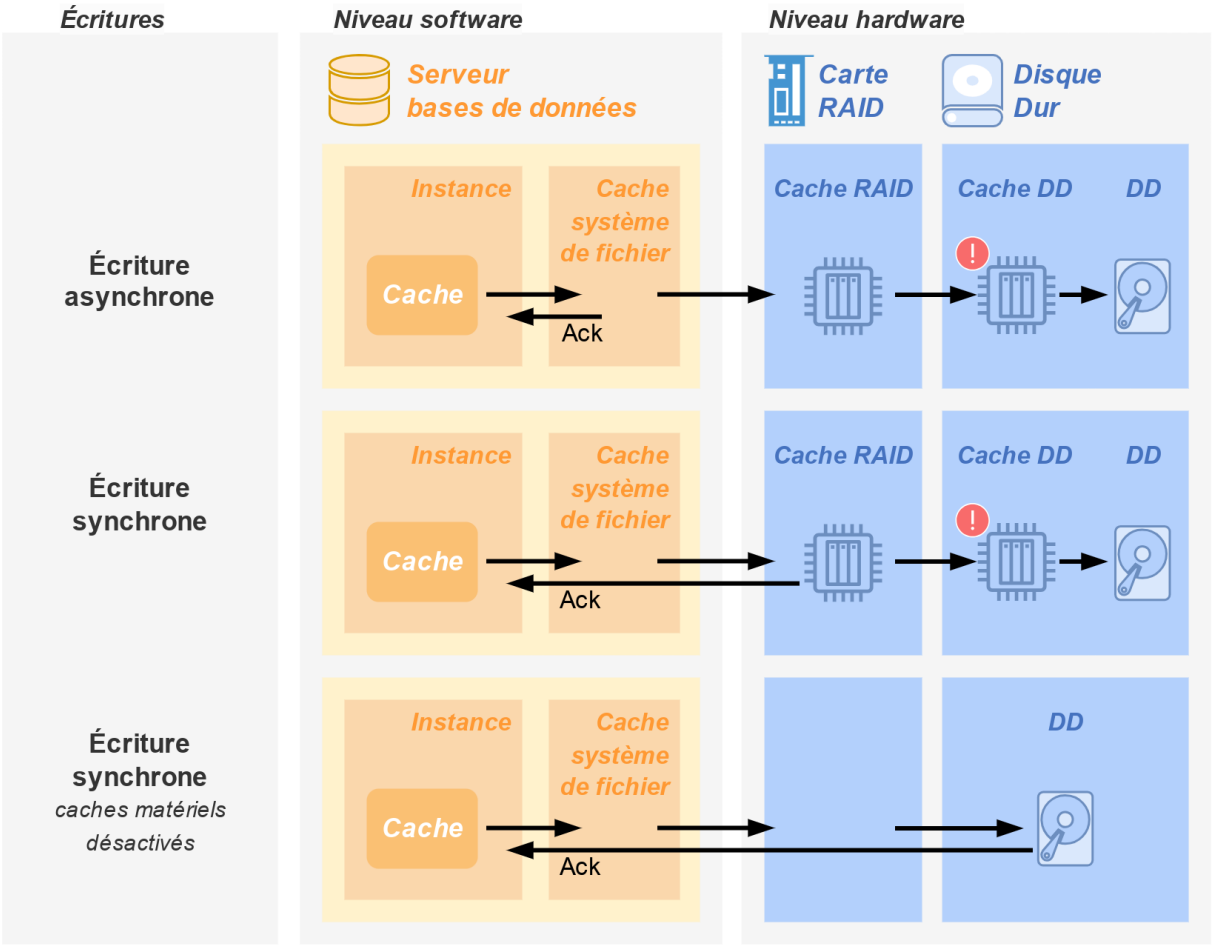


Figure 1/ .2: Niveaux de cache et fsync

Une écriture peut être soit synchrone soit asynchrone. Pour comprendre ce mécanisme, nous allons simplifier le cheminement de l'écriture d'un bloc :

- Dans le cas d'une écriture **asynchrone** : Un processus qui modifie un fichier écrit en fait d'abord dans le cache du système de fichiers du système d'exploitation (OS), cache situé en RAM (mémoire volatile). L'OS confirme tout de suite au processus que l'écriture a été réalisée pour lui rendre la main au plus vite : il y a donc un gain en performance important. Cependant, le bloc ne sera écrit sur disque que plus tard afin notamment de grouper les demandes d'écritures des autres processus, et de réduire les déplacements des têtes de lecture/écriture des disques, qui sont des opérations coûteuses en temps. Entre la confirmation de l'écriture et l'écriture réelle sur les disques, il peut se passer un certain délai : si une panne survient durant celui-ci, les données soi-disant écrites seront perdues, car pas encore physiquement sur le disque.
- Dans le cas d'une écriture **synchrone** : Un processus écrit dans le cache du système

d'exploitation, puis demande explicitement à l'OS d'effectuer la synchronisation (écriture physique) sur disque. Les blocs sont donc écrits sur les disques immédiatement et le processus n'a la confirmation de l'écriture qu'une fois cela fait. Il attendra donc pendant la durée de cette opération, mais il aura la garantie que la donnée est bien présente physiquement sur les disques. Cette synchronisation est très coûteuse et lente (encore plus avec un disque dur classique et ses têtes de disques à déplacer).

Un phénomène équivalent peut se produire à nouveau au niveau matériel (hors du contrôle de l'OS) : pour gagner en performance, les constructeurs ont rajouté un système de cache au sein des cartes RAID. L'OS (et donc le processus qui écrit) a donc confirmation de l'écriture dès que la donnée est présente dans ce cache, alors qu'elle n'est pas encore écrite sur disque. Afin d'éviter la perte de donnée en cas de panne électrique, ce cache est secouru par une batterie qui laissera le temps d'écrire le contenu du cache. Vérifiez qu'elle est bien présente sur vos disques et vos cartes contrôleur RAID.

1.8.10 Configuration des traces

- Selon système/distribution :
  - `log_destination`
  - `logging_collector`
  - emplacement et nom différent pour `postgresql-????.log`
- `log_line_prefix` à compléter :
  - `log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '`
- `lc_messages` = `C` (anglais)

PostgreSQL dispose de plusieurs moyens pour enregistrer les traces : soit il les envoie sur la sortie des erreurs (`stderr`, `csvlog` et `jsonlog`), soit il les envoie à syslog (`syslog`, seulement sous Unix), soit il les envoie au journal des événements (`eventlog`, sous Windows uniquement). Dans le cas où les traces sont envoyées sur la sortie des erreurs, il peut récupérer les messages via un démon appelé *logger process* qui va enregistrer les messages dans des fichiers. Ce démon s'active en configurant le paramètre `logging_collector` à `on`.

Tout cela est configuré par défaut différemment selon le système et la distribution. Red Hat active `logging_collector` et PostgreSQL dépose ses traces dans des fichiers journaliers `$PGDATA/log/postgresql-<jour de la semaine>.log`. Debian utilise `stderr` sans autre paramétrage et c'est le système qui dépose les traces dans `/var/log/postgresql/postgresql-VERSION-nominstance.log`. Les deux variantes fonctionnent. En fonction des habitudes et contraintes locales, il est possible de préférer et d'activer l'une ou l'autre.

L'entête de chaque ligne des traces doit contenir au moins la date et l'heure exacte (`%t` ou `%m` suivant la précision désirée) : des traces sans date et heure ne servent à rien. Des entêtes complets sont suggérés par la documentation de l'analyseur de log pgBadger :

```
log_line_prefix = '%t [%p]: [%l-1] db=%d,user=%u,app=%a,client=%h '
```

Beaucoup d'utilisateurs français récupèrent les traces de PostgreSQL en français. Bien que cela semble une bonne idée au départ, cela se révèle être souvent un problème. Non pas à cause de la qualité de la traduction, mais plutôt parce que les outils de traitement des traces fonctionnent uniquement avec des traces en anglais. Même un outil comme pgBadger, pourtant écrit par un Français, ne sait pas interpréter des traces en français. De plus, la moindre recherche sur Internet ramènera plus de liens si le message est en anglais. Positionnez donc `lc_messages` à `C`.

1.8.11 Configuration des tâches de fond

Laisser ces deux paramètres à `on` :

- `autovacuum`
- `track_counts`

En dehors du *logger process*, PostgreSQL dispose d'autres tâches de fond.

Les processus autovacuum jouent un rôle important pour disposer de bonnes performances : ils empêchent une fragmentation excessive des tables et index, et mettent à jour les statistiques sur les données (statistiques servant à l'optimiseur de requêtes).

La récupération des statistiques sur l'activité permet le bon fonctionnement de l'autovacuum et donne de nombreuses informations importantes à l'administrateur de bases de données.

Ces deux tâches de fond devraient toujours être activés.

1.8.12 Se faciliter la vie

- Création automatique de configuration
  - pgtune et https://pgtune.leopard.in.ua/
  - http://pgconfigurator.cybertec.at/
- Documentation et analyse de configuration
  - https://postgresqlco.nf

pgtune existe en plusieurs versions. La version en ligne de commande va détecter automatiquement le nombre de CPU et la quantité de RAM, alors que la version web nécessitera que ces informations soient saisies. Suivant le type d'utilisation, pgtune proposera une configuration adaptée. Cette configuration n'est évidemment pas forcément optimale par rapport à vos applications, tout simplement parce qu'il ne connaît que les ressources et le type d'utilisation, mais c'est généralement un bon point de départ.

pgconfigurator est un outil plus récent, un peu plus graphique, mais il remplit exactement le même but que pgtune.

Enfin, le site postgresql.co.nf[32] est un peu particulier. C'est en quelque sorte une encyclopédie sur les paramètres de PostgreSQL, mais il est aussi possible de lui faire analyser une configuration. Après analyse, des informations supplémentaires seront affichées pour améliorer cette configuration, que ce soit pour la stabilité du serveur comme pour ses performances.

[32] https://postgresqlco.nf

1.9 MISE À JOUR

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

1.9.1 Recommandations

- Les *Release Notes*
- Intégrité des données
- Bien redémarrer le serveur !

Chaque nouvelle version de PostgreSQL est accompagnée d'une note expliquant les améliorations, les corrections et les innovations apportées par cette version, qu'elle soit majeure ou mineure. Ces notes contiennent toujours une section dédiée aux mises à jour dans laquelle se trouvent des conseils essentiels.

Les *Releases Notes* sont présentes dans l'annexe E de la documentation officielle[33].

Les données de votre instance PostgreSQL sont toujours compatibles d'une version mineure à l'autre. Ainsi, les mises à jour vers une version mineure supérieure peuvent se faire sans migration de données, sauf cas exceptionnel qui serait alors précisé dans les notes de version. Par exemple, de la 15.3 à la 15.4, il a été recommandé de reconstruire les index de type BRIN[34] pour prendre en compte une correction de bug les concernant. Autre exemple : à partir de la 10.3[35], `pg_dump` a imposé des noms d'objets qualifiés pour des raisons de sécurité, ce qui a posé problème pour certains réimports.

Pensez éventuellement à faire une sauvegarde préalable par sécurité.

À contrario, si la mise à jour consiste en un changement de version majeure (par exemple, de la 9.6 à la 16), il est nécessaire de s'assurer que les données seront transférées correctement sans incompatibilité. Là encore, il est important de lire les *Releases Notes* **avant** la mise à jour.

Le site https://why-upgrade.depesz.com/, basé sur les release notes, permet de compiler les différences entre plusieurs versions de PostgreSQL.

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

[33] https://docs.postgresql.fr/current/release.html
[34] https://www.postgresql.org/docs/release/15.4/
[35] https://www.postgresql.org/docs/release/10.3/

1.9.2 Mise à jour mineure

- Méthode
  - arrêter PostgreSQL
  - mettre à jour les binaires
  - redémarrer PostgreSQL
- Pas besoin de s'occuper des données, sauf cas exceptionnel
  - bien lire les *Release Notes* pour s'en assurer

Faire une mise à jour mineure est simple et rapide.

La première action est de lire les *Release Notes* pour s'assurer qu'il n'y a pas à se préoccuper des données. C'est généralement le cas mais il est préférable de s'en assurer avant qu'il ne soit trop tard.

La deuxième action est de faire la mise à jour. Tout dépend de la façon dont PostgreSQL a été installé :

- par compilation, il suffit de remplacer les anciens binaires par les nouveaux ;
- par paquets précompilés, il suffit d'utiliser le système de paquets (`apt` sur Debian et affiliés, `yum` ou `dnf` sur Red Hat et affiliés) ;
- par l'installeur graphique, en le ré-exécutant.

Ceci fait, un redémarrage du serveur est nécessaire. Il est intéressant de noter que les paquets Debian s'occupent directement de cette opération. Il n'est donc pas nécessaire de le refaire.

1.9.3 Mise à jour majeure

- Bien lire les *Release Notes*
- Bien tester l'application avec la nouvelle version
  - rechercher les régressions en terme de fonctionnalités et de performances
  - penser aux extensions et aux outils
- Pour mettre à jour
  - mise à jour des binaires
  - et mise à jour/traitement des fichiers de données
- 3 méthodes
  - dump/restore
  - réplication logique, externe (Slony) ou interne
  - `pg_upgrade`

Faire une mise à jour majeure est une opération complexe à préparer prudemment.

La première action là-aussi est de lire les *Release Notes* pour bien prendre en compte les régressions potentielles en terme de fonctionnalités et/ou de performances. Cela n'arrive presque jamais mais c'est possible malgré toutes les précautions mises en place.

La deuxième action est de mettre en place un serveur de tests où se trouve la nouvelle version de PostgreSQL avec les données de production. Ce serveur sert à tester PostgreSQL mais aussi, et même surtout, l'application. Le but est de vérifier encore une fois les régressions possibles.

N'oubliez pas de tester les extensions non officielles, voire développées en interne, que vous avez installées. Elles sont souvent moins bien testées.

N'oubliez pas non plus de tester les outils d'administration, de monitoring, de modélisation. Ils nécessitent souvent une mise à jour pour être compatibles avec la nouvelle version installée.

Une fois que les tests sont concluants, arrive le moment de la mise en production. C'est une étape qui peut être longue car les fichiers de données doivent être traités. Il existe plusieurs méthodes que nous détaillerons après.

1.9.4 Mise à jour majeure par dump/restore

- Méthode historique
- Simple et sans risque
  - mais d'autant plus longue que le volume de données est important
- Outils :
  - `pg_dumpall -g` puis `pg_dump`
  - `psql` puis `pg_restore`

Il s'agit de la méthode la plus ancienne et la plus sûre. L'idée est de sauvegarder l'ancienne version avec l'outil de sauvegarde de la nouvelle version. `pg_dumpall` peut suffire, mais `pg_dump` est malgré tout recommandé. Le problème de lenteur vient surtout de la restauration. `pg_restore` est un outil assez lent pour des volumétries importantes. Il convient donc de sélectionner cette solution si le volume de données n'est pas conséquent (pas plus d'une centaine de Go) ou si les autres méthodes ne sont pas possibles. Cependant, il est possible d'accélérer la restauration en utilisant la parallélisation (option `--jobs`). Ceci n'est possible que si la sauvegarde a été faite avec `pg_dump -Fd` ou `-Fc`. Il est à noter que cette sauvegarde peut elle aussi être parallélisée (option `--jobs` là encore).

1.9.5 Mise à jour majeure par Slony

- Nécessite d'utiliser l'outil de réplication Slony
- Permet un retour en arrière immédiat sans perte de données

La méthode Slony est certainement la méthode la plus compliquée. C'est aussi une méthode qui permet un retour arrière vers l'ancienne version sans perte de données.

L'idée est d'installer la nouvelle version de PostgreSQL normalement, sur le même serveur ou sur un autre serveur. Il faut installer Slony sur l'ancienne et la nouvelle instance, et déclarer la réplication de l'ancienne instance vers la nouvelle. Les utilisateurs peuvent continuer à travailler pendant le transfert initial des données. Ils n'auront pas de blocages, tout au plus une perte de performances dues à la lecture et à l'envoi des données vers le nouveau serveur. Une fois le transfert initial réalisé, les données modifiées entre temps sont transférées vers le nouveau serveur.

Une fois arrivé à la synchronisation des deux serveurs, il ne reste plus qu'à déclencher un *switchover*.

La réplication aura lieu ensuite entre le nouveau serveur et l'ancien serveur, ce qui permet un retour en arrière sans perte de données. Une fois acté que le nouveau serveur donne pleine satisfaction, il suffit de désinstaller Slony des deux côtés.

1.9.6 Mise à jour majeure par réplication logique

- Possible entre versions 10 et supérieures
- Remplace Slony, Bucardo…
- Bascule très rapide
- Et retour possible

La réplication logique rend possible une migration entre deux instances de version majeure différente avec une indisponibilité très courte.

La réplication logique n'est disponible en natif qu'à partir de la version 10, la base à migrer doit donc être en version 10 ou supérieure.

Le même principe que les outils de réplication par trigger comme Slony ou Bucardo est utilisé, mais plus simplement et avec les outils du moteur. Le principe est de répliquer une base à l'identique alors que la production tourne. Des clés primaires sur chaque table sont souhaitables mais pas forcément obligatoires.

Lors de la bascule, il suffit d'attendre que les dernières données soient répliquées, ce qui peut être très rapide, et de connecter les applications au nouveau serveur. La réplication peut alors être inversée pour garder l'ancienne production synchrone, permettant de rebasculer dessus en cas de problème sans perdre les données modifiées depuis la bascule.

1.9.7 Mise à jour majeure par pg_upgrade

- `pg_upgrade` : fourni avec PostgreSQL
- Prérequis : pas de changement de format des fichiers entre versions
- Nécessite les deux versions sur le même serveur
- Support des serveurs PostgreSQL à migrer :
  - version minimale 9.2 pour pg_upgrade v15
  - version minimale 8.4 sinon

`pg_upgrade` est certainement l'outil le plus rapide pour une mise à jour majeure.

Il profite du fait que les formats des fichiers de données n'évolue pas, ou de manière rétrocompatible, entre deux versions majeures. Il n'est donc pas nécessaire de tout réécrire.

Grossièrement, son fonctionnement est le suivant. Il récupère la déclaration des objets sur l'ancienne instance avec un `pg_dump` du schéma de chaque base et de chaque objet global. Il intègre la déclaration des objets dans la nouvelle instance. Il fait un ensemble de traitement sur les identifiants d'objets et de transactions. Puis, il copie les fichiers de données de l'ancienne instance vers la nouvelle instance. La copie est l'opération la plus longue, mais comme il n'est pas nécessaire de reconstruire les index et de vérifier les contraintes, cette opération est bien plus rapide que la restauration d'une sauvegarde style `pg_dump`. Pour aller encore plus rapidement, il est possible de créer des liens physiques à la place de la copie des fichiers. Ceci fait, la migration est terminée.

En 2010, Stefan Kaltenbrunner et Bruce Momjian avaient mesuré qu'une base de 150 Go mettait 5 heures à être mise à jour avec la méthode historique (sauvegarde/restauration). Elle mettait 44 minutes en mode copie et 42 secondes en mode lien lors de l'utilisation de `pg_upgrade`.

Vu ses performances, ce serait certainement l'outil à privilégier. Cependant, c'est un outil très complexe et quelques bugs particulièrement méchants ont terni sa réputation. Notre recommandation est de bien tester la mise à jour avant de le faire en production, et uniquement sur des bases suffisamment volumineuses permettant de justifier l'utilisation de cet outil.

Lors du développement de la version 15, les développeurs ont supprimé certaines vieilles parties du code, ce qui le rend à présent incompatible avec des versions très anciennes (de la 8.4 à la 9.1).

1.9.8 Mise à jour de l'OS

Si vous migrez aussi l'OS ou déplacez les fichiers d'une instance :

- compatibilité architecture
- compatibilité librairies
  - réindexation parfois nécessaire
  - ex : Debian 10 et glibc 2.28

Un projet de migration PostgreSQL est souvent l'occasion de mettre à jour le système d'exploitation. Vous pouvez également en profiter pour déplacer l'instance sur un autre serveur à l'OS plus récent en copiant (à froid) le PGDATA.

Il faut bien sûr que l'architecture physique (32/64 bits, *big/little indian*) reste la même. Cependant, même entre deux versions de la même distribution, certains composants du système d'exploitation peuvent avoir une influence, à commencer par la `glibc`. Cette dernière définit l'ordre des caractères, ce qui se retrouve dans les index. Une incompatibilité entre deux versions sur ce point oblige donc à

reconstruire les index, sous peine d'incohérence avec les fonctions de comparaison sur le nouveau système et de corruption à l'écriture.

Daniel Vérité détaille sur son blog[36] le problème pour les mises à jour entre Debian 9 et 10, à cause de la mise à jour de la `glibc`. L'utilisation des collations ICU[37] dans les index contourne le problème mais elles sont encore peu répandues.

Ce problème ne touche bien sûr pas les migrations ou les restaurations avec `pg_dump` / `pg_restore` : les données sont alors transmises de manière logique, indépendamment des caractéristiques physiques des instances source et cible, et les index sont systématiquement reconstruits sur la machine cible.

[36] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html
[37] https://blog-postgresql.verite.pro/2018/07/27/icu_ext.html

1.10 CONCLUSION

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

1.10.1 Pour aller plus loin

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

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

La documentation de Dalibo pour l'installation de PostgreSQL sur Windows est disponible sur https://public.dalibo.com/archives/etudes/installer_postgresql_9.0_sous_windows.pdf.

1.10.2 Questions

N'hésitez pas, c'est le moment !

1.11 QUIZ

https://dali.bo/b_quiz

1.12 TRAVAUX PRATIQUES

1.12.1 Installation à partir des sources (optionnel)

But : Installer PostgreSQL à partir du code source

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

Outils de compilation

Installer les outils de compilation suivants, si ce n'est déjà fait.

Sous Rocky Linux 8 ou 9, il faudra utiliser `dnf` :

```
sudo dnf -y group install "Development Tools"
sudo dnf -y install readline-devel openssl-devel wget bzip2
```

Sous Debian ou Ubuntu :

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

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

Se connecter en tant que l'utilisateur **srcpostgres**.

Téléchargement

- Consulter le site officiel du projet et relever la dernière version de PostgreSQL.
- Télécharger l'archive des fichiers sources de la dernière version stable.
- Les placer dans `/opt/pgsql/src`.

Compilation et installation

L'installation des binaires compilés se fera dans `/opt/pgsql/15/`.

- Configurer en conséquence l'environnement de compilation (`./configure`).
- Compiler PostgreSQL.

Installer les fichiers obtenus.

Où se trouvent les binaires installés de PostgreSQL ?

Configurer le système

Ajouter les variables d'environnement `PATH` et `LD_LIBRARY_PATH` au `~srcpostgres/.bash_profile` de l'utilisateur **srcpostgres** pour accéder facilement à ces binaires.

Création d'une instance

Avec `initdb`, initialiser une instance dans `/opt/pgsql/15/data` en spécifiant **postgres** comme nom de super-utilisateur, et en activant les sommes de contrôle.

Démarrer l'instance.

- Tenter une première connexion avec `psql`.
- Pourquoi cela échoue-t-il ?

Se connecter en tant qu'utilisateur **postgres**. Ressortir.

Dans `.bash_profile`, configurer la variable d'environnement `PGUSER` pour se connecter toujours en tant que **postgres**.

Première base

Créer une première base de donnée nommée `test`.

Se connecter à la base `test` et créer quelques tables.

Arrêt

Arrêter cette instance.

1.12.2 Installation depuis les paquets binaires du PGDG

But : Installer PostgreSQL à partir des paquets communautaires
Cette instance servira aux TP suivants.

Pré-installation

Quelle commande permet d'installer les paquets binaires de PostgreSQL ?

Quelle version est packagée ?

Quels paquets devront également être installés ?

Installation

Installer le dépôt.

Désactiver le module d'installation pour la version PostgreSQL de la distribution.

Installer les paquets de PostgreSQL14 : serveur, client, contribs.

Quel est le chemin des binaires ?

Création de la première instance

Créer une première instance avec les outils de la famille Red Hat en activant les sommes de contrôle (*checksums*).

Vérifier ce qui a été fait dans le journal `initdb.log`.

Démarrage

Démarrer l'instance.

Activer le démarrage de l'instance au démarrage de la machine.

> Où sont les fichiers de données (`PGDATA`), et les traces de l'instance ?

Configuration

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

> Quel est l'utilisateur sous lequel tourne l'instance ?

Connexion

> En tant que **root**, tenter une connexion avec `psql` .

> En tant que **postgres**, tenter une connexion avec `psql` . Quitter.

> À quelle base se connecte-t-on par défaut ?

> Créer une première base de données et y créer des tables.

1.13 TRAVAUX PRATIQUES (SOLUTIONS)

1.13.1 Installation à partir des sources (optionnel)

Outils de compilation

> Installer les outils de compilation suivants, si ce n'est déjà fait.

Ces actions doivent être effectuées en tant qu'utilisateur privilégié (soit directement en tant que **root**, soit en utilisant la commande `sudo`).

Sous Rocky Linux 8 ou 9, il faudra utiliser `dnf` :

```
sudo dnf -y group install "Development Tools"
sudo dnf -y install readline-devel openssl-devel wget bzip2
```

Sous Debian ou Ubuntu :

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

Une fois ces outils installés, tout ce qui suit devrait fonctionner sur toute version de Linux.

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

```
sudo useradd --home-dir /opt/pgsql --system --create-home srcpostgres
sudo usermod --shell /bin/bash srcpostgres
```

> Se connecter en tant que l'utilisateur **srcpostgres**.

Se connecter en tant qu'utilisateur **srcpostgres** :

```
sudo su - srcpostgres
```

Téléchargement

> - Consulter le site officiel du projet et relever la dernière version de PostgreSQL.
> - Télécharger l'archive des fichiers sources de la dernière version stable.
> - Les placer dans `/opt/pgsql/src`.

En tant qu'utilisateur **srcpostgres**, créer un répertoire dédié aux sources :

```
mkdir ~srcpostgres/src
cd ~/src
```

Aller sur https://postgresql.org[38], cliquer *Download* et récupérer le lien vers l'archive des fichiers sources de la dernière version stable (PostgreSQL 15.2 au moment où ceci est écrit). Il est possible de le faire en ligne de commande :

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

```
wget https://ftp.postgresql.org/pub/source/v15.2/postgresql-15.2.tar.bz2
```

Il faut décompresser l'archive :

```
tar xjvf postgresql-15.2.tar.bz2
cd postgresql-15.2
```

Compilation et installation

> L'installation des binaires compilés se fera dans `/opt/pgsql/15/`.
>
> - Configurer en conséquence l'environnement de compilation (`./configure`).
> - Compiler PostgreSQL.

Configuration :

```
./configure --prefix /opt/pgsql/15
```

```
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking which template to use... linux
checking whether NLS is wanted... no
checking for default port number... 5432
...
...
config.status: linking src/include/port/linux.h to src/include/pg_config_os.h
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
```

Des fichiers sont générés, notamment le *Makefile*.

La compilation se lance de manière classique. Elle peut prendre un certain temps sur les machines un peu anciennes :

```
make
```

```
make -C ./src/backend generated-headers
make[1]: Entering directory '/opt/pgsql/postgresql-15.2/src/backend'
make -C catalog distprep generated-header-symlinks
make[2]: Entering directory '/opt/pgsql/postgresql-15.2/src/backend/catalog'
make[2]: Nothing to be done for 'distprep'.
...
...
make[2]: Entering directory '/opt/pgsql/postgresql-15.2/src/test/perl'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/opt/pgsql/postgresql-15.2/src/test/perl'
make[1]: Leaving directory '/opt/pgsql/postgresql-15.2/src'
make -C config all
make[1]: Entering directory '/opt/pgsql/postgresql-15.2/config'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/opt/pgsql/postgresql-15.2/config
```

> Installer les fichiers obtenus.

L'installation peut se faire en tant que **srcpostgres** car nous avons défini comme cible le répertoire `/opt/pgsql/15/` qui lui appartient :

```
make install
```

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

> Où se trouvent les binaires installés de PostgreSQL ?

Les binaires installés sont situés dans le répertoire `/opt/pgsql/15/bin`.

```
ls -1 /opt/pgsql/15/bin
```

```
clusterdb
createdb
createuser
...
pg_verifybackup
pg_waldump
pgbench
postgres
postmaster
psql
reindexdb
vacuumdb
```

Configurer le système

> Ajouter les variables d'environnement `PATH` et `LD_LIBRARY_PATH` au `~srcpostgres/.bash_profile` de l'utilisateur **srcpostgres** pour accéder facilement à ces binaires.

Ajouter les lignes suivantes à la fin du fichier `~srcpostgres/.bash_profile` (ce fichier peut ne pas exister préalablement, et un autre fichier peut être nécessaire selon l'environnement utilisé) :

```
export PGDATA=/opt/pgsql/15/data
export PATH=/opt/pgsql/15/bin:$PATH
export LD_LIBRARY_PATH=/opt/pgsql/15/lib:$LD_LIBRARY_PATH
```

Il faut ensuite recharger le fichier à l'aide de la commande suivante (ne pas oublier le point et l'espace au début de la commande) ; ou se déconnecter et se reconnecter.

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

Vérifier que les chemins sont bons :

```
which psql
```

```
~/15/bin/psql
```

Création d'une instance

> Avec `initdb`, initialiser une instance dans `/opt/pgsql/15/data` en spécifiant **postgres** comme nom de super-utilisateur, et en activant les sommes de contrôle.

```
$ initdb -D $PGDATA -U postgres --data-checksums

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

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

creating directory /opt/pgsql/15/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

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

Success. You can now start the database server using:

    pg_ctl -D /opt/pgsql/15/data -l logfile start
```

Démarrer l'instance.

```
pg_ctl -D $PGDATA -l $PGDATA/server.log start

waiting for server to start.... done
server started

cat $PGDATA/server.log

2023-03-21 18:04:22.445 UTC [51123] LOG:  starting PostgreSQL 15.2 on
 ↪  x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.3.1 20220421 (Red Hat 11.3.1-2),
 ↪  64-bit
2023-03-21 18:04:22.446 UTC [51123] LOG:  listening on IPv4 address "127.0.0.1",
 ↪  port 5432
2023-03-21 18:04:22.446 UTC [51123] LOG:  could not bind IPv6 address "::1": Cannot
 ↪  assign requested address
2023-03-21 18:04:22.452 UTC [51123] LOG:  listening on Unix socket
 ↪  "/tmp/.s.PGSQL.5432"
2023-03-21 18:04:22.459 UTC [51126] LOG:  database system was shut down at
 ↪  2023-03-21 18:03:31 UTC
2023-03-21 18:04:22.467 UTC [51123] LOG:  database system is ready to accept
 ↪  connections
```

- Tenter une première connexion avec `psql`.
- Pourquoi cela échoue-t-il ?

```
psql
```

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
 ↪  role "srcpostgres" does not exist
```

Par défaut, `psql` demande à se connecter avec un nom d'utilisateur identique à celui en cours, mais la base de données ne connaît pas l'utilisateur **srcpostgres**. Par défaut, elle ne connaît que **postgres**.

Se connecter en tant qu'utilisateur **postgres**. Ressortir.

```
psql -U postgres
```

```
psql (15.2)
Type "help" for help.

postgres=# exit
```

Noter que la connexion fonctionne parce que le `pg_hba.conf` livré avec les sources est par défaut très laxiste (méthode `trust` en local et *via* `localhost` !). (Il y a d'ailleurs eu un avertissement lors de la création de la base.)

Dans `.bash_profile`, configurer la variable d'environnement `PGUSER` pour se connecter toujours en tant que **postgres**. Retester la connexion directe avec `psql`.

Ajouter ceci à à la fin du fichier `~srcpostgres/.bash_profile` :

```
export PGUSER=postgres
```

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

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

La connexion doit fonctionner sur le champ :

```
psql
```

```
psql (15.2)
Type "help" for help.

postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
postgres=#
\q
```

Première base

> Créer une première base de donnée nommée `test`.

En ligne de commande shell :

```
createdb --echo test
```

```
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE test;
```

Alternativement, depuis `psql` :

```
postgres=# CREATE DATABASE test ;
```

```
CREATE DATABASE
```

> Se connecter à la base `test` et créer quelques tables.

```
psql test
```

```
test=# CREATE TABLE premieretable (x int) ;
CREATE TABLE
```

Arrêt

> Arrêter cette instance.

```
$ pg_ctl stop
```

```
waiting for server to shut down.... done
server stopped
```

```
tail $PGDATA/server.log
```

```
2023-03-21 18:06:51.316 UTC [51137] FATAL:  role "srcpostgres" does not exist
2023-03-21 18:09:22.559 UTC [51124] LOG:  checkpoint starting: time
2023-03-21 18:09:26.696 UTC [51124] LOG:  checkpoint complete: wrote 44 buffers
 ↪  (0.3%); 0 WAL file(s) added, 0 removed, 0 recycled; write=4.116 s, sync=0.009 s,
 ↪  total=4.137 s; sync files=11, longest=0.006 s, average=0.001 s; distance=299 kB,
 ↪  estimate=299 kB
2023-03-21 18:14:19.381 UTC [51123] LOG:  received fast shutdown request
2023-03-21 18:14:19.400 UTC [51123] LOG:  aborting any active transactions
2023-03-21 18:14:19.401 UTC [51123] LOG:  background worker "logical replication
 ↪  launcher" (PID 51129) exited with exit code 1
2023-03-21 18:14:19.402 UTC [51124] LOG:  shutting down
2023-03-21 18:14:19.404 UTC [51124] LOG:  checkpoint starting: shutdown immediate
2023-03-21 18:14:19.496 UTC [51124] LOG:  checkpoint complete: wrote 897 buffers
 ↪  (5.5%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.022 s, sync=0.064 s,
 ↪  total=0.095 s; sync files=249, longest=0.049 s, average=0.001 s; distance=4016
 ↪  kB, estimate=4016 kB
2023-03-21 18:14:19.502 UTC [51123] LOG:  database system is shut down
```

1.13.2 Installation depuis les paquets binaires du PGDG

Pré-installation

> Quelle commande permet d'installer les paquets binaires de PostgreSQL ?

Tout dépend de votre distribution. Les systèmes les plus représentés sont Debian et ses dérivés (notamment Ubuntu), ainsi que Red Hat et dérivés (CentOS, Rocky Linux).

Le présent TP utilise Rocky Linux 8, basé sur une version communautaire qui se veut être le successeur du projet CentOS, interrompu en 2021[39]. La version 9 fonctionne également. Une version plus complète, ainsi que l'utilisation de paquets Debian, sont traités dans l'annexe « Installation de PostgreSQL depuis les paquets communautaires ».

> Quelle version est packagée ?

La dernière version stable de PostgreSQL disponible au moment de la rédaction de ce module est la version 15.2. Par contre, la dernière version disponible dans les dépôts dépend de votre distribution. C'est la raison pour laquelle **les dépôts du PGDG sont à privilégier**.

> Quels paquets devront également être installés ?

Le paquet `libpq` devra également être installé. À partir de la version 11, il est aussi nécessaire d'installer les paquets `llvmjit` (pour la compilation à la volée), qui réclame elle-même la présence du dépôt EPEL, mais c'est une fonctionnalité optionnelle qui ne sera pas traitée ici.

Installation

> Installer le dépôt en vous inspirant des consignes sur :
> https://www.postgresql.org/download/linux/redhat
> mais en ajoutant les contribs et les sommes de contrôle.

Préciser :

- PostgreSQL 15
- Red Hat Enterprise, Rocky or Oracle version 8 (ou 9 selon le cas)
- x86_64

Nous allons reprendre ligne à ligne ce script et le compléter.

Se connecter avec l'utilisateur **root** sur la machine de formation, et recopier le script proposé par le guide. Dans la commande ci-dessous, les deux lignes **doivent être copiées et collées ensemble**.

```
# Rocky Linux 8
dnf install -y https://download.postgresql.org\
/pub/repos/yum/reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# Rocky Linux 9
dnf install -y https://download.postgresql.org\
/pub/repos/yum/reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

[39] https://blog.centos.org/2020/12/future-is-centos-stream

> Désactiver le module d'installation pour la version PostgreSQL de la distribution.

Cette opération est nécessaire pour Rocky Linux 8 ou 9.

```
dnf -qy module disable postgresql
```

> Installer les paquets de PostgreSQL15 : serveur, client, contribs.

```
dnf install -y postgresql15-server postgresql15-contrib
```

Il s'agit respectivement des binaires du serveur et des « contribs » et extensions (en principe optionnelles, mais chaudement conseillées).

Le paquet `postgresql15` (outils client) fait partie des dépendances et est installé automatiquement.

On met volontairement de côté le paquet `llvmjit`.

> Quel est le chemin des binaires ?

Ils se trouvent dans `/usr/pgsql-15/bin/` (chemin propre à ce packaging) :

```
ls -1 /usr/pgsql-15/bin/
```

```
clusterdb
createdb
...
...
postgres
postgresql-15-check-db-dir
postgresql-15-setup
postmaster
psql
reindexdb
vacuumdb
vacuumlo
```

Noter qu'il existe des liens dans `/usr/bin` pointant vers la version la plus récente des outils en cas d'installation de plusieurs versions :

```
which psql
```

```
/usr/bin/psql
```

```
file /usr/bin/psql
```

```
/usr/bin/psql: symbolic link to /etc/alternatives/pgsql-psql
```

```
file /etc/alternatives/pgsql-psql
```

```
/etc/alternatives/pgsql-psql: symbolic link to /usr/pgsql-15/bin/psql
```

Création de la première instance

> Créer une première instance avec les outils de la famille Red Hat en activant les sommes de contrôle (*checksums*).

La création d'une instance passe par un outil spécifique à ces paquets.

Cet outil doit être appelé en tant que **root** (et non **postgres**).

Optionnellement, on peut ajouter des paramètres d'initialisation à cette étape. La mise en place des sommes de contrôle est généralement conseillée pour être averti de toute corruption des fichiers.

Toujours en temps que **root** :

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

Initializing database ... OK
```

> Vérifier ce qui a été fait dans le journal `initdb.log`.

La sortie de la commande précédente est redirigée vers le fichier `initdb.log` situé dans le répertoire qui contient celui de la base (`PGDATA`). Il est possible d'y vérifier l'ensemble des étapes réalisées, notamment l'activation des sommes de contrôle.

```
$ cat /var/lib/pgsql/15/initdb.log

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

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

fixing permissions on existing directory /var/lib/pgsql/15/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. You can now start the database server using:

    /usr/pgsql-15/bin/pg_ctl -D /var/lib/pgsql/15/data/ -l logfile start
```

Ne pas tenir compte de la dernière ligne, qui est une suggestion qui ne tient pas compte des outils prévus pour cet OS.

Démarrage

> Démarrer l'instance.

Attention, si vous avez créé une instance à partir des sources dans le TP précédent, elle doit impérativement être arrêtée pour pouvoir démarrer la nouvelle instance !
En effet, comme nous n'avons pas modifié le port par défaut (5432), les deux instances ne peuvent pas être démarrées en même temps, sauf à modifier le port dans la configuration de l'une d'entre elles.

En tant que **root** :

```
systemctl start postgresql-15
```

Si aucune erreur ne s'affiche, tout va bien à priori.

Pour connaître l'état de l'instance :

```
systemctl status postgresql-15
```

```
● postgresql-15.service - PostgreSQL 15 database server
   Loaded: loaded (/usr/lib/systemd/system/postgresql-15.service; disabled; vendor
 ↪  preset: disabled)
   Active: active (running) since Tue 2023-03-21 18:36:33 UTC; 5s ago
     Docs: https://www.postgresql.org/docs/15/static/
  Process: 68744 ExecStartPre=/usr/pgsql-15/bin/postgresql-15-check-db-dir ${PGDATA}
 ↪  (code=exited, status=0/SUCCESS)
 Main PID: 68749 (postmaster)
    Tasks: 7 (limit: 14208)
   Memory: 17.5M
   CGroup: /system.slice/postgresql-15.service
           ├─68749 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
           ├─68751 postgres: logger
           ├─68752 postgres: checkpointer
           ├─68753 postgres: background writer
           ├─68755 postgres: walwriter
           ├─68756 postgres: autovacuum launcher
           └─68757 postgres: logical replication launcher

Mar 21 18:36:33 vm-formation1 systemd[1]: Starting PostgreSQL 15 database server...
Mar 21 18:36:33 vm-formation1 postmaster[68749]: 2023-03-21 18:36:33.123 UTC [68749]
 ↪  LOG:  redirecting log output to logging collector process
Mar 21 18:36:33 vm-formation1 postmaster[68749]: 2023-03-21 18:36:33.123 UTC [68749]
 ↪  HINT:  Future log output will appear in directory "log".
Mar 21 18:36:33 vm-formation1 systemd[1]: Started PostgreSQL 15 database server.
```

> Activer le démarrage de l'instance au démarrage de la machine.

Le packaging Red Hat ne prévoie pas l'activation du service au boot, il faut le demander explicitement :

```
systemctl enable postgresql-15
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/postgresql-15.service →
↪  /usr/lib/systemd/system/postgresql-15.service.
```

> Où sont les fichiers de données (`PGDATA`), et les traces de l'instance ?

Les données et fichiers de configuration sont dans `/var/lib/pgsql/15/data/`.

```
ls -1 /var/lib/pgsql/15/data/
```

```
base
current_logfiles
global
log
pg_commit_ts
pg_dynshmem
pg_hba.conf
pg_ident.conf
pg_logical
pg_multixact
pg_notify
pg_replslot
pg_serial
pg_snapshots
pg_stat
pg_stat_tmp
pg_subtrans
pg_tblspc
pg_twophase
PG_VERSION
pg_wal
pg_xact
postgresql.auto.conf
postgresql.conf
postmaster.opts
postmaster.pid
```

Les traces sont dans le sous-répertoire `log`.

```
ls -l /var/lib/pgsql/15/data/log/
```

```
total 4
-rw-------. 1 postgres postgres 709 Mar  2 10:37 postgresql-Wed.log
```

NB : Dans les paquets pour Rocky Linux, le nom exact du fichier dépend du jour de la semaine.

Configuration

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

Il est possible de vérifier dans le fichier `postgresql.conf` que par défaut, le serveur écoute uniquement l'interface réseau `localhost` (la valeur est commentée mais c'est bien celle par défaut) :

```
$ grep listen_addresses /var/lib/pgsql/15/data/postgresql.conf
#listen_addresses = 'localhost'         # what IP address(es) to listen on;
```

Il faudra donc modifier cela pour que des utilisateurs puissent se connecter depuis d'autres machines :

```
listen_addresses = '*'          # what IP address(es) to listen on;
```

Il est aussi possible de vérifier au niveau système en utilisant la commande `netstat` (qui nécessite l'installation du paquet `net-tools`) :

```
netstat -anp | grep postmaster
```

```
tcp      0  0 127.0.0.1:5432   0.0.0.0:*         LISTEN      28028/postmaster
tcp6     0  0 ::1:5432         :::*              LISTEN      28028/postmaster
udp6     0  0 ::1:57123        ::1:57123         ESTABLISHED 28028/postmaster
unix 2 [ ACC ] STREAM LISTENING 301922 28028/postmaster /tmp/.s.PGSQL.5432
unix 2 [ ACC ] STREAM LISTENING 301920 28028/postmaster
 ↪  /var/run/postgresql/.s.PGSQL.5432
```

(La présence de lignes `tcp6` dépend de la configuration de la machine.)

> Quel est l'utilisateur sous lequel tourne l'instance ?

C'est l'utilisateur nommé **postgres** :

```
ps -U postgres -f -o pid,user,cmd
```

```
    PID USER     CMD
  52533 postgres /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
  52534 postgres  \_ postgres: logger
  52535 postgres  \_ postgres: checkpointer
  52536 postgres  \_ postgres: background writer
  52538 postgres  \_ postgres: walwriter
  52539 postgres  \_ postgres: autovacuum launcher
  52540 postgres  \_ postgres: logical replication launcher
```

Il possède aussi le `PGDATA` :

```
ls -l /var/lib/pgsql/15/
```

```
total 8
drwx------.  2 postgres postgres    6 Feb  9 08:13 backups
drwx------. 20 postgres postgres 4096 Mar  4 16:18 data
-rw-------.  1 postgres postgres  910 Mar  2 10:35 initdb.log
```

postgres est le nom traditionnel sur la plupart des distributions, mais il n'est pas obligatoire (par exemple, le TP par compilation utilise un autre utilisateur).

Connexion

> En tant que **root**, tenter une connexion avec `psql`.

```
 # psql
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432"
 ↪  failed: FATAL:  role "root" does not exist
```

Cela échoue car `psql` tente de se connecter avec l'utilisateur système en cours, soit **root**. Ça ne marchera pas mieux cependant en essayant de se connecter avec l'utilisateur **postgres** :

```
psql -U postgres
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432"
 ↪  failed: FATAL:  Peer authentication failed for user "postgres"
```

En effet, le `pg_hba.conf` est configuré de telle manière que l'utilisateur de PostgreSQL et celui du système doivent porter le même nom.

> En tant que **postgres**, tenter une connexion avec `psql`. Quitter.

```
sudo -iu postgres psql
```

```
psql (15.3)
Type "help" for help.

postgres=# exit
```

La connexion fonctionne donc indirectement depuis tout utilisateur pouvant effectuer un `sudo`.

> À quelle base se connecte-t-on par défaut ?

```
sudo -iu postgres psql
```

```
psql (15.2)
Type "help" for help.

postgres=# \conn
invalid command \conn
Try \? for help.
postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in
 ↪  "/var/run/postgresql" at port "5432".
postgres=#
```

Là encore, la présence d'une base nommée `postgres` est une tradition et non une obligation.

Première base

> Créer une première base de données et y créer des tables.

```
sudo -iu postgres psql
```

```
postgres=# CREATE DATABASE test ;
CREATE DATABASE
```

Alternativement :

```
sudo -iu postgres createdb test
```

Se connecter explicitement à la bonne base :

```
sudo -iu postgres psql -d test
```

```
test=# CREATE TABLE mapremieretable (x int);
CREATE TABLE

test=# \d+

                          Liste des relations
 Schéma |       Nom       | Type  | Propriétaire | Taille  | Description
--------+-----------------+-------+--------------+---------+-------------
 public | mapremieretable | table | postgres     | 0 bytes |
```

1.14 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.14.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 16) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 16 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql16-server postgresql16-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql16-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb
# cat /var/lib/pgsql/16/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-16/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/16/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-16
# systemctl stop postgresql-16
# systemctl status postgresql-16
# systemctl reload postgresql-16
# systemctl restart postgresql-16
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-16
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/16/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-16.service \
        /etc/systemd/system/postgresql-16-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/16/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb postgresql-16-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/16/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-16-infocentre
# systemctl [enable|disable] postgresql-16-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

1.14.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 16 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-16 postgresql-client-16
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/16/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/16/main`
Fichiers de configuration	dans `/etc/postgresql/16/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 16 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/16/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 16 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 16 infocentre \
  --port=12345 \
  --datadir=/PGDATA/16/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/16/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/16/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 16 infocentre start
```

1.14.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (16.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (16.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/16/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

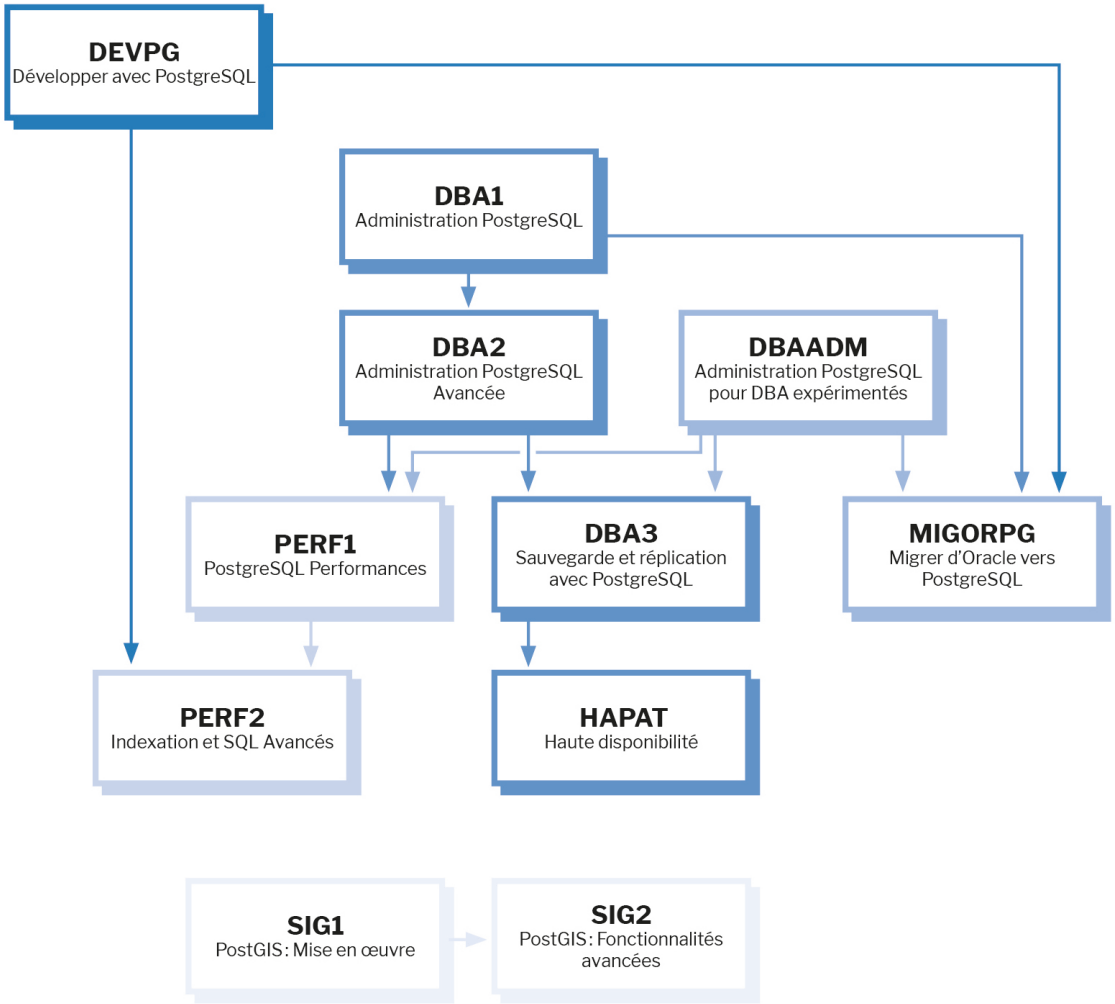
```
root:~# systemctl reload postgresql-16

root:~# pg_ctlcluster 16 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

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.