Module W6

Pooling

24.04

Table des matières

Sur ce document

Formation	Module W6
Titre	Pooling
Révision	24.04
PDF	https://dali.bo/w6_pdf
EPUB	https://dali.bo/w6_epub
HTML	https://dali.bo/w6_html
Slides	https://dali.bo/w6_slides
TP	https://dali.bo/w6_tp
TP (solutions)	https://dali.bo/w6_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

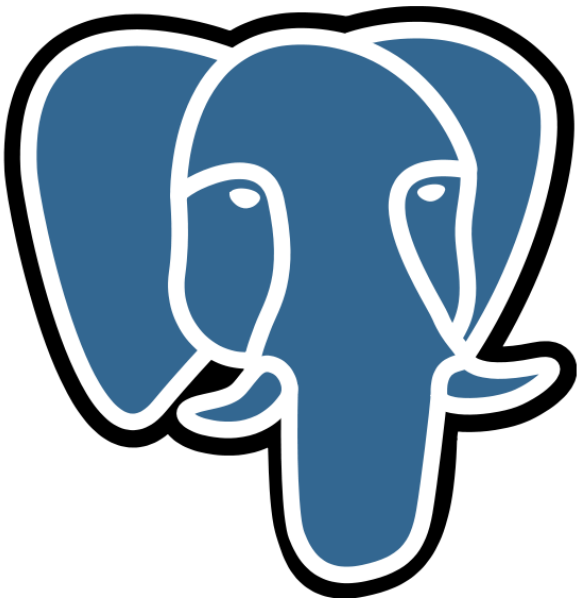
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/



1.1 AU MENU

- Concepts
- Pool de connexion avec PgBouncer

Ce module permet d'aborder le *pooling*.

Ce qui suit ne portera que sur un unique serveur, et n'aborde pas le sujet de la répartition de charge.

Nous étudierons principalement un logiciel : PgBouncer.

1.1.1 Objectifs

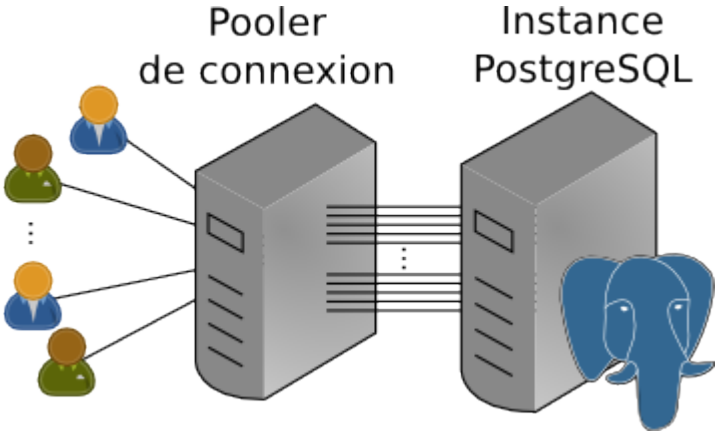
- Savoir ce qu'est un pool de connexion ?
- Avantage, inconvénients & limites
- Savoir mettre en place un pooler de connexion avec PgBouncer

1.2 POOL DE CONNEXION

- Qu'est-ce qu'un pool de connexion ?
- Présentation
- Avantages et inconvénients
- Mise en œuvre avec PgBouncer

Dans cette partie, nous allons étudier la théorie des poolers de connexion. La partie suivante sera la mise en pratique avec l'outil PgBouncer.

1.2.1 Serveur de pool de connexions



1.2.2 Serveur de pool de connexions

- S'intercale entre le SGBD et les clients
- Maintient des connexions ouvertes avec le SGBD
- Distribue aux clients ses connexions au SGBD
- Attribue une connexion existante au SGBD dans ces conditions
  - même rôle
  - même base de données
- Différents poolers :
  - intégrés aux applicatifs
  - service séparé (où ?)

Un serveur de pool de connexions s'intercale entre les clients et le système de gestion de bases de données. Les clients ne se connectent plus directement sur le SGBD pour accéder aux bases. Ils passent par le pooler qui se fait passer pour le serveur de bases de données. Le pooler maintient alors des connexions vers le SGBD et en gère lui-même l'attribution aux utilisateurs.

Chaque connexion au SGBD est définie par deux paramètres : le rôle de connexion et la base de données. Ainsi, une connexion maintenue par le pooler ne sera attribuée à un utilisateur que si ce couple rôle/base de données est le même.

Les conditions de création de connexions au SGBD sont donc définies dans la configuration du pooler.

Un pooler peut se présenter sous différentes formes :

- comme **brique logicielle** incorporée dans le code applicatif sur les serveurs d'applications (fourni par Hibernate ou Apache Tomcat, par exemple) ;
- comme **service** séparé, démarré sur un serveur et écoutant sur un port donné, où les clients se connecteront pour accéder à la base de données voulue (exemples : PgBouncer, pgPool)

Nous nous consacrons dans ce module aux pools de connexions accessibles à travers un service.

Noter qu'il ne faut pas confondre un pooler avec un outil de répartition de charge (même si un pooler peut également permettre la répartition de charge, comme PgPool).

L'emplacement d'un pooler se décide au cas par cas selon l'architecture. Il peut se trouver intégré à l'application, et lui être dédié, ce qui garantit une latence faible entre pooler et application. Il peut être centralisé sur le serveur de bases de données et servir plusieurs applications, voire se trouver sur une troisième machine. Il faut aussi réfléchir à ce qui se passera en cas de bascule entre deux instances.

1.2.3 Intérêts du pool de connexions

- Évite le coût de connexion
  - …et de déconnexion
- Optimise l'utilisation des ressources du SGBD
- Contrôle les connexions, peut les rediriger
- Évite des déconnexions
  - redémarrage (mise à jour, bascule)
  - saturations temporaires des connexions sur l'instance

Le maintien des connexions entre le pooler et le SGBD apporte un gain non négligeable lors de l'établissement des connexions. Effectivement, pour chaque nouvelle connexion à PostgreSQL, nous avons :

- la création d'un nouveau processus ;
- l'allocation des ressources mémoires utiles à la session ;
- le positionnement des paramètres de session de l'utilisateur.

Tout ceci engendre une consommation du processeur.

Ce travail peut durer plusieurs dizaines, voire centaines de millisecondes. Cette latence induite peut alors devenir un réel goulot d'étranglement dans certains contextes. Or, une connexion déjà active maintenue dans un pool peut être attribuée à une nouvelle session immédiatement : cette latence est donc *de facto* fortement limitée par le pooler.

En fonction du mode de fonctionnement, de la configuration et du type de pooler choisi, sa transparence vis-à-vis de l'application et son impact sur les performances seront différents.

De plus, cette position privilégiée entre les utilisateurs et le SGBD permet au pooler de contrôler et centraliser les connexions vers le ou les SGBD. Effectivement, les applications pointant sur le serveur de pool de connexions, le SGBD peut être situé n'importe où, voire sur plusieurs serveurs différents. Le pooler peut aiguiller les connexions vers un serveur différent en fonction de la base de données demandée. Certains poolers peuvent détecter une panne d'un serveur et aiguiller vers un autre. En cas de *switchover*, *failover*, évolution ou déplacement du SGBD, il peut suffire de reconfigurer le pooler.

Enfin, les sessions entrantes peuvent être mises en attente si plus aucune connexion n'est disponible et qu'elles ne peuvent pas en créer de nouvelle. On évite donc de lever immédiatement une erreur, ce qui est le comportement par défaut de PostgreSQL.

Pour la base de données, le pooler est une application comme une autre.

Si la configuration le permet (`pg_hba.conf`), il est possible de se connecter à une instance aussi bien via le pooler que directement selon l'utilisation (application, batch, administration…)

1.2.4 Inconvénients du pool de connexions

- Transparence suivant le mode :
  - par sessions
  - par transactions
  - par requêtes
- Performances, si mal configuré (latence)
- Point délicat : l'authentification !
- Complexité
- SPOF potentiel
- Impact sur les fonctionnalités, selon le mode

Les fonctionnalités de PostgreSQL utilisables au travers d'un pooler varient suivant son mode de fonctionnement du pooler (par requêtes, transactions ou sessions). Nous verrons que plus la mutualisation est importante, plus les restrictions apparaissent.

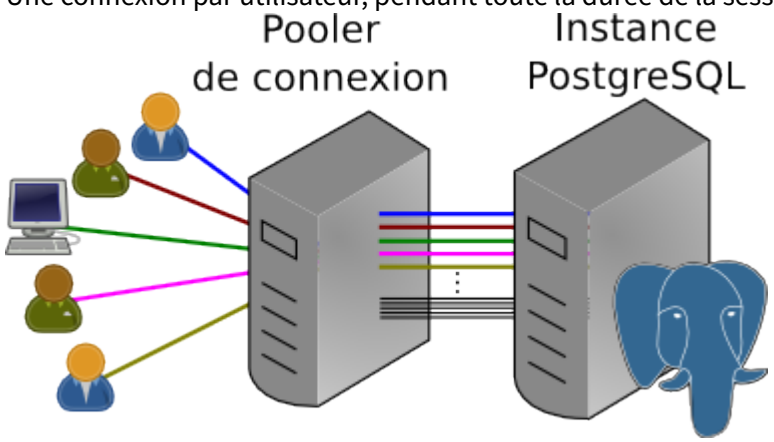
Un pooler est un élément en plus entre l'application et vos données, donc il aura un coût en performances. Il ajoute notamment une certaine latence. On n'introduit donc pas un pooler sans avoir identifié un problème. Si la configuration est bien faite, cet impact est normalement négligeable, ou en tout cas sera compensé par des gains au niveau de la base de données, ou en administration.

Comme dans tout système de proxy, un des points délicats de la configuration est l'authentification, avec certaines restrictions.

Un pooler est un élément en plus dans votre architecture. Il la rend donc plus complexe et y ajoute ses propres besoins en administration, en supervision et ses propres modes de défaillance. Si vous faites passer toutes vos connexions par un pooler, celui-ci devient un nouveau point de défaillance possible (SPOF). Une redondance est bien sûr possible, mais complique à nouveau les choses.

1.3 DE SESSIONS

Une connexion par utilisateur, pendant toute la durée de la session.



Un pool de connexion par session attribue une connexion au SGBD à un unique utilisateur pendant toute la durée de sa session. Si aucune connexion à PostgreSQL n'est disponible, une nouvelle connexion est alors créée, dans la limite exprimée dans la configuration du pooler. Si cette limite est atteinte, la session est mise en attente ou une erreur est levée.

1.3.1 Intérêts du pooling de sessions

- Avantages :
  - limite le temps d'établissement des connexions
  - mise en attente si trop de sessions
  - simple
  - transparent pour les applications
- Inconvénients :
  - périodes de non-activité des sessions conservées
  - nombre de sessions actives au pooler égal au nombre de connexions actives au SGBD

L'intérêt d'un pool de connexion en mode session est principalement de conserver les connexions ouvertes vers le SGBD. On économise ainsi le temps d'établissement de la connexion pour les nou-

velles sessions entrantes si une connexion est déjà disponible. Dans ce cas, le pooler permet d'avoir un comportement de type *pre-fork* côté SGBD.

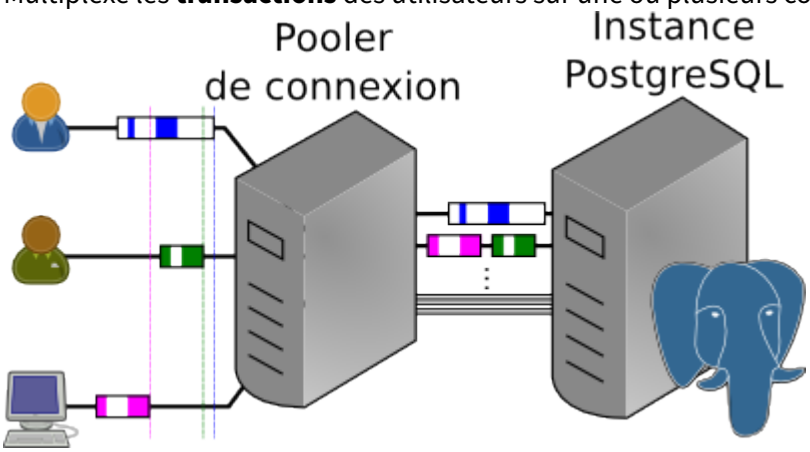
L'autre intérêt est de ne pas rejeter une connexion, même s'il n'y a plus de connexions possibles au SGBD. Contrairement au comportement de PostgreSQL, les connexions sont placées en attente si elles ne peuvent pas être satisfaites immédiatement.

Ce mode de fonctionnement est très simple et robuste, c'est le plus transparent vis-à-vis des sessions clientes, avec un impact quasi nul sur le code applicatif.

Aucune optimisation du temps de travail côté SGBD n'est donc possible. S'il peut être intéressant de limiter le nombre de sessions ouvertes sur le pooler, il sera en revanche impossible d'avoir plus de sessions ouvertes sur le pooler que de connexions disponibles sur le SGDB.

1.4 DE TRANSACTIONS

Multiplexe les **transactions** des utilisateurs sur une ou plusieurs connexions.



Dans le schéma présenté ici, chaque bloc représente une transaction délimitée par une instruction `BEGIN`, suivie plus tard d'un `COMMIT` ou d'un `ROLLBACK`. Chaque zone colorée représente une requête au sein de la transaction.

Un pool de connexions par transactions multiplexe les transactions des utilisateurs entre une ou plusieurs connexions au SGBD. Une transaction est débutée sur la première connexion à la base qui soit inactive (`idle`). Toutes les requêtes d'une transaction sont envoyées sur la même connexion.

Ce schéma suppose que le pool accorde la première connexion disponible en partant du haut dans l'ordre où les transactions se présentent.

1.4.1 Avantages & inconvénients du pooling de transactions

- Avantages
  - mêmes avantages que le pooling de sessions
  - meilleure utilisation du temps de travail des connexions
    - les connexions sont utilisées par une ou plusieurs sessions
  - plus de sessions possibles côté pooler pour moins de connexions au SGBD
- Inconvénients
  - prise en charge partielle des instructions préparées
  - période de non-activité des sessions toujours possible

Les intérêts d'un pool de connexion en mode transaction sont multiples en plus de cumuler ceux d'un pool de connexion par session.

Il est désormais possible de partager une même connexion au SGBD entre plusieurs sessions utilisateurs. En effet, il existe de nombreux contextes où une session a un taux d'occupation relativement faible : requêtes très simples et exécutées très rapidement, génération des requêtes globalement plus lente que la base de données, couche applicative avec des temps de traitement des données reçues plus importants que l'exécution côté SGBD, etc.

Avoir la capacité de multiplexer les transactions de plusieurs sessions entre plusieurs connexions permet ainsi de limiter le nombre de connexions à la base en optimisant leur taux d'occupation. Cette économie de connexions côté PostgreSQL a plusieurs avantages :

- moins de connexions à gérer par le serveur, qui est donc plus disponible pour les connexions actives ;
- moins de connexions, donc économie de mémoire, devenue disponible pour les requêtes ;
- possibilité d'avoir un plus grand nombre de clients connectés côté pooler sans pour autant atteindre un nombre critique de connexions côté SGBD.

En revanche, avec ce mode de fonctionnement, le pool de connexions n'assure pas aux clients connectés que leurs requêtes et transactions iront toujours vers la même connexion, bien au contraire ! Ainsi, si l'application utilise des requêtes préparées (c'est-à-dire en trois phases `PREPARE`, `BIND`, `EXECUTE`), la commande `PREPARE` pourrait être envoyée sur une connexion alors que les commandes `EXECUTE` pourraient être dirigées vers d'autres connexions, menant leur exécution tout droit à une erreur.

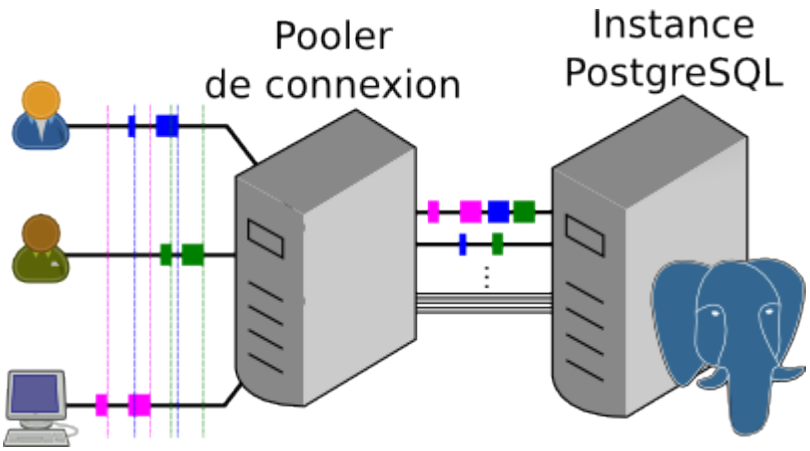
Seules les requêtes au sein d'une même transaction sont assurées d'être exécutées sur la même connexion. Ainsi, au début de cette transaction, la connexion est alors réservée exclusivement à l'utilisateur propriétaire de la transaction. Donc si le client prend son temps entre les différentes

étapes d'une transaction (statut `idle in transaction` pour PostgreSQL), il monopolisera la connexion sans que les autres clients puissent en profiter.

Ce type de pool de connexion a donc un impact non négligeable à prendre en compte lors du développement.

1.5 DE REQUÊTES

- Un pool de connexions en mode requêtes multiplexe toutes les **requêtes** sur une ou plusieurs connexions



Un pool de connexions par requêtes multiplexe les requêtes des utilisateurs entre une ou plusieurs connexions au SGBD.

Dans le schéma présenté ici, chaque bloc coloré représente une requête. Elles sont placées exactement aux mêmes instants que dans le schéma présentant le pool de connexion en mode transactions.

1.5.1 Avantages & inconvénients du pooling de requêtes

- Avantages
  - les mêmes que pour le pooling de sessions et de transactions.
  - utilisation optimale du temps de travail des connexions
  - encore plus de sessions possibles côté pooler pour moins de connexions au SGBD
- Inconvénients
  - les mêmes que pour le pooling de transactions
  - interdiction des transactions !

Les intérêts d'un pool de connexions en mode requêtes sont les mêmes que pour un pool de connexion en mode de transactions. Cependant, dans ce mode, toutes les requêtes des clients sont multiplexées à travers les différentes connexions disponibles et inactives.

Ainsi, il est désormais possible d'optimiser encore plus le temps de travail des connexions au SGBD, supprimant la possibilité de bloquer une connexion dans un état `idle in transaction`. Nous sommes alors capables de partager une même connexion avec encore plus de clients, augmentant ainsi le nombre de sessions disponibles sur le pool de connexions tout en conservant un nombre limité de connexions côté SGBD.

En revanche, si les avantages sont les mêmes que ceux d'un pooler de connexion en mode transactions, les limitations sont elles aussi plus importantes. Il n'est effectivement plus possible d'utiliser des transactions, en plus des requêtes préparées !

En pratique, le pooling par requête sert à interdire totalement les transactions. En effet, un pooling par transaction n'utilisant que des transactions implicites (d'un seul ordre) parviendra au même résultat.

1.6 AVEC PGBOUNCER

- Deux projets existent : PgBouncer et PgPool-II
- Les deux sont sous licence BSD
- PgBouncer
  - le plus évolué et éprouvé pour le *pooling*

Deux projets sous licence BSD coexistent dans l'écosystème de PostgreSQL pour mettre en œuvre un pool de connexion : PgBouncer et PgPool-II.

PgPool-II[1] est le projet le plus ancien, développé et maintenu principalement par SRA OSS[2]. Ce projet est un véritable couteau suisse capable d'effectuer bien plus que du pooling (répartition de charge, bascules…). Malheureusement, cette polyvalence a un coût important en termes de fonctionnalités et complexités. PgPool n'est effectivement capable de travailler qu'en tant que pool de connexion par session.

PgBouncer[3] est un projet créé par Skype. Il a pour objectifs :

- de n'agir qu'en tant que pool de connexion ;
- d'être le plus léger possible ;
- d'avoir les meilleures performances possibles ;
- d'avoir le plus de fonctionnalités possible sur son cœur de métier.

PgBouncer étant le plus évolué des deux, nous allons le mettre en œuvre dans les pages suivantes.

[1] https://www.pgpool.net/
[2] https://www.sraoss.co.jp/index_en.php
[3] https://www.pgbouncer.org/

1.6.1 PgBouncer : Fonctionnalités

- Techniquement : un démon
- Disponible sous Unix & Windows
- Modes sessions / transactions / requêtes
- Prise en charge partielle des requêtes préparées
- Redirection vers des serveurs et/ou bases différents
- Mise en attente si plus de connexions disponibles
- Mise en pause des connexions
- Paramétrage avancé des sessions clientes et des connexions aux bases
- Mise à jour sans couper les sessions existantes
- Supervision depuis une base virtuelle de maintenance
- Pas de répartition de charge

PgBouncer est techniquement assez simple : il s'agit d'un simple démon, auxquelles les applicatifs se connectent (en croyant avoir affaire à PostgreSQL), et qui retransmet requêtes et données.

PgBouncer dispose de nombreuses fonctionnalités, toutes liées au pooling de connexions. La majorité de ces fonctionnalités ne sont pas disponibles avec PgPool.

À l'inverse de ce dernier, PgBouncer n'offre pas de répartition de charge. Ses créateurs renvoient vers des outils au niveau TCP comme HAProxy. De même, pour les bascules d'un serveur à l'autre, ils conseillent plutôt de s'appuyer sur le niveau DNS.

Ce qui suit n'est qu'un extrait de la documentation de référence, assez courte : https://www.pgbouncer.org/config.html. La FAQ[4] est également à lire.

1.6.2 PgBouncer : Installation

- Par les paquets fournis par le PGDG :
  - `yum|dnf install pgbouncer`
  - `apt install pgbouncer`
- Installation par les sources
  - Dépôt pgbouncer[5]

[4] https://www.pgbouncer.org/faq.html

PgBouncer est disponible sous la forme d'un paquet binaire sur les principales distributions Linux et les dépôts du PGDG.

Il y a quelques différences mineures d'empaquetage : sous Red Hat/CentOS/Rocky Linux, le processus tourne avec un utilisateur système **pgbouncer** dédié, alors que sur Debian et dérivées, il fonctionne sous l'utilisateur **postgres**.

Il est bien sûr possible de recompiler depuis les sources.

Sous Windows, le projet fournit une archive[6] à décompresser.

1.6.3 PgBouncer : Fichier de configuration

- Format `ini`
- Un paramètre par ligne
- Aucune unité dans les valeurs
- Tous les temps sont exprimés en seconde
- Sections : `[databases]`, `[users]`, `[pgbouncer]`

Les paquets binaires créent un fichier de configuration `/etc/pgbouncer/pgbouncer.ini`.

Une ligne de configuration concerne un seul paramètre, avec le format suivant :

```
parametre = valeur
```

PgBouncer n'accepte pas que l'utilisateur spécifie une unité pour les valeurs. L'unité prise en compte par défaut est la seconde.

Il y a plusieurs sections :

- les bases de données (`[databases]`), où on spécifie pour chaque base la chaîne de connexion à utiliser ;
- les utilisateurs (`[users]`), pour des propriétés liées aux utilisateurs ;
- le moteur (`[pgbouncer]`), où se fait tout le reste de la configuration de PgBouncer.

[6] https://github.com/pgbouncer/pgbouncer/releases/

1.6.4 PgBouncer : Connexions

- TCP/IP
  - `listen_addr` : adresses
  - `listen_port` (6432)
- Socket Unix (`unix_socket_dir` , `unix_socket_mode` , `unix_socket_group`)
- Chiffrement TLS

PgBouncer accepte les connexions en mode socket Unix et via TCP/IP. Les paramètres disponibles ressemblent beaucoup à ce que PostgreSQL propose.

`listen_addr` correspond aux interfaces réseau sur lesquels PgBouncer va écouter. Il est par défaut configuré à la boucle locale, mais vous pouvez ajouter les autres interfaces disponibles, ou tout simplement une étoile pour écouter sur toutes les interfaces. `listen_port` précise le port de connexion : traditionnellement, c'est 6432, mais on peut le changer, par exemple à 5432 pour que la configuration de connexion des clients reste identique.

Si PostgreSQL se trouve sur le même serveur et que vous voulez utiliser le port 5432 pour PgBouncer, il faudra bien sûr changer le port de connexion de PostgreSQL.

Pour une connexion uniquement en local par la socket Unix, il est possible d'indiquer où le fichier socket doit être créé (paramètre `unix_socket_dir` : `/tmp` sur Red Hat/CentOS, `/var/run/postgresql` sur Debian et dérivés), quel groupe doit lui être affecté (`unix_socket_group`) et les droits du fichier (`unix_socket_mode`). Si un groupe est indiqué, il est nécessaire que l'utilisateur détenteur du processus `pgbouncer` soit membre de ce groupe.

Cela est pris en compte par les paquets binaires d'installation.

PgBouncer supporte également le chiffrement TLS.

1.6.5 PgBouncer : Définition des accès aux bases

- Section `[databases]`
- Une ligne par base sous la forme libpq :

```
data1 = host=localhost port=5433 dbname=data1 pool_size=50
```

- Paramètres de connexion :
  - `host`, `port`, `dbname` ; `user`, `password`
  - `pool_size`, `pool_mode`, `connect_query`
  - `client_encoding`, `datestyle`, `timezone`…
- Base par défaut :

```
 + = host=ip1 port=5432 dbname=data0
```

- `auth_hba_file` : équivalent à `pg_hba.conf`

Lorsque l'utilisateur cherche à se connecter à PostgreSQL, il va indiquer l'adresse IP du serveur où est installé PgBouncer et le numéro de port où écoute PgBouncer. Il va aussi indiquer d'autres informations comme la base qu'il veut utiliser, le nom d'utilisateur pour la connexion, son mot de passe, etc.

Lorsque PgBouncer reçoit cette requête de connexion, il extrait le nom de la base et va chercher dans la section `[databases]` si cette base de données est indiquée. Si oui, il remplacera tous les paramètres de connexion qu'il trouve dans son fichier de configuration et établira la connexion entre ce client et cette base. Si jamais la base n'est pas indiquée, il cherchera s'il existe une base de connexion par défaut (nom indiqué par une étoile) et l'utilisera dans ce cas.

Exemples de chaîne de connexion :

```
prod = host=p1 port=5432 dbname=erp pool_size=40 pool_mode=transaction
prod = host=p1 port=5432 dbname=erp pool_size=10 pool_mode=session
```

Il est donc possible de faire beaucoup de chose :

- n'accéder qu'à un serveur dont les bases sont décrites ;
- accéder à différents serveurs PostgreSQL depuis un même serveur de pooling, suivant le nom de la base ou de l'utilisateur ;
- remplacer l'utilisateur de connexion par celui défini par `user` ;
- etc.

Néanmoins, les variables `user` et `password` sont très peu utilisées.

La chaîne de connexion est du type libpq mais tout ce qu'accepte la libpq n'est pas forcément accepté par PgBouncer (notamment pas de variable service, pas de possibilité d'utiliser directement le fichier standard `.pgpass`).

Le paramètre `auth_hba_file` peut pointer vers un fichier de même format que `pg_hba.conf` pour filtrer les accès au niveau du pooler (en plus des bases).

1.6.6 PgBouncer : Authentification par fichier de mots de passe

- Liste des utilisateurs contenue dans `userlist.txt`
- Contenu de ce fichier
  - `"utilisateur" "mot de passe"`
- Paramètres dans le fichier de configuration
  - `auth_type` : type d'authentification (`trust`, `md5`, `scram-sha-256`…)
  - `auth_file` : emplacement de la liste des utilisateurs et mots de passe
  - `admin_users` : liste des administrateurs
  - `stats_users` : liste des utilisateurs de supervision

PgBouncer n'a pas accès à l'authentification de PostgreSQL. De plus, son rôle est de donner accès à des connexions déjà ouvertes à des clients. PgBouncer doit donc s'authentifier auprès de PostgreSQL à la place des clients, et vérifier lui-même les mots de passe de ces clients. (Ce mécanisme ne dispense évidemment pas les clients de fournir les mots de passe.)

La première méthode, et la plus simple, est de déclarer les utilisateurs dans le fichier pointé par le paramètre `auth_file`, par défaut `userlist.txt`. Les utilisateurs et mots de passe y sont stockés comme ci-dessous selon le type d'authentification, obligatoirement encadrés avec des guillemets doubles.

```
"guillaume" "supersecret"
"marc" "md59fa7827a30a483125ca3b7218bad6fee"
"pgbench" "SCRAM-SHA-256$4096:Rqk+MWaDN9rKXOLuoj8eCw==$ry5DD2Ptk…+6do76FN/ys="
```

Le type d'authentification est plus limité que ce que PostgreSQL propose. Le type `trust` indique que l'utilisateur sera accepté par PgBouncer quel que soit le mot de passe qu'il fournit ; il faut que le serveur PostgreSQL soit configuré de la même façon. Cela est bien sûr déconseillé. `auth_type` peut prendre les valeurs `md5` ou `scram-sha-256` pour autoriser des mots de passe chiffrés. Pour des raisons de compatibilité descendante, `md5` permet aussi d'utiliser `scram-sha-256`.

Les paramètres de configuration `admin_users` et `stats_users` permettent d'indiquer la liste d'utilisateurs pouvant se connecter à PgBouncer directement pour obtenir des commandes de contrôle sur PgBouncer ainsi que des statistiques d'activité. Ils peuvent être déclarés dans le fichier des mots de passe avec un mot de passe arbitraire en clair.

`userlist.txt` est évidemment un fichier dont les accès doivent être les plus restreints possibles.

1.6.7 PgBouncer : Authentification par délégation

- Créer un rôle dédié
- Copier son hash de mot de passe dans `userlist.txt`
- Déclaration dans le pool avec `auth_user` :

```
prod = host=p1 port=5432 dbname=erp auth_user=frontend
```

- `auth_query` : requête pour vérifier le mot de passe via ce rôle
- => Plus la peine de déclarer les autres rôles

La maintenance du fichier de mots de passe peut vite devenir fastidieuse. Il est possible de déléguer un rôle à la recherche des mots de passe avec le paramètre `auth_user`, à poser globalement

```
auth_user = frontend
```

ou au niveau de la base :

```
prod = host=p1 port=5432 dbname=erp pool_mode=transaction auth_user=frontend
```

Ce rôle se connectera et ira valider dans l'instance le *hash* du mot de passe du client. Il sera donc inutile de déclarer d'autres rôles dans `userlist.txt`.

Le rôle d'authentification et son mot de passe se déclarent par exemple ainsi :

```
SET password_encryption = 'scram-sha-256' ;
CREATE ROLE frontend PASSWORD 'motdepassecompliqué' LOGIN ;
SELECT rolpassword FROM pg_authid WHERE rolname = 'frontend' \gx
```

Le hachage du mot de passe obtenu est recopié dans `userlist.txt` :

```
"frontend" "SCRAM-SHA-256$4096:LaN76vw5sMU/0kvs9joNpA==$/ … ="
```

L'utilisateur **frontend** va utiliser le paramètre `auth_query` pour savoir comment récupérer les identifiants de connexion de l'utilisateur applicatif qui veut se connecter. Par défaut, il s'agit simplement de requêter la vue `pg_shadow` :

```
auth_query = SELECT usename, passwd FROM pg_shadow WHERE usename=$1
```

D'autres variantes sont possibles, comme une requête plus élaborée sur `pg_authid`, ou une fonction avec les bons droits de consultation avec une clause `SECURITY DEFINER`. La documentation donne un exemple[7] :

```
CREATE OR REPLACE FUNCTION pgbouncer.user_lookup
                    (IN i_username text, OUT uname text, OUT phash text)
RETURNS record AS $$
BEGIN
    SELECT usename, passwd FROM pg_catalog.pg_shadow
    WHERE usename = i_username INTO uname, phash;
    RETURN;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
REVOKE ALL ON FUNCTION pgbouncer.user_lookup(text) FROM public, pgbouncer;
GRANT EXECUTE ON FUNCTION pgbouncer.user_lookup(text) TO pgbouncer;
```

et cette fonction s'utilise ainsi :

```
auth_query = SELECT * FROM pgbouncer.user_lookup($1);
```

Il faut évidemment que l'utilisateur d'authentification (et seulement lui) ait les droits nécessaires, et cela dans toutes les bases impliquées.

La mise en place de cette configuration est facilement source d'erreur, il faut bien surveiller les traces de PostgreSQL *et* PgBouncer.

1.6.8 PgBouncer : Nombre de connexions

- Côté client :
  - `max_client_conn` (100)
  - attention à `ulimit` !
  - `max_db_connections`
- Par utilisateur/base :
  - `default_pool_size` (20)
  - `min_pool_size` (0)
  - `reserve_pool_size` (0)

PostgreSQL dispose d'un nombre de connexions maximum (`max_connections` dans `postgresql.conf`, 100 par défaut). Il est un compromis entre le nombre de requêtes simultanément actives, leur complexité, le nombre de CPU, le nombre de processus gérables par l'OS… L'utilisation d'un pooler

[7] http://www.pgbouncer.org/config.html#example

en multiplexage se justifie notamment quand des centaines, voire des milliers, de connexions simultanées sont nécessaires, celles-ci étant inactives la plus grande partie du temps. Même avec un nombre modeste de connexions, une application se connectant et se déconnectant très souvent peut profiter d'un pooler.

Les paramètres suivants de `pgbouncer.ini` permettent de paramétrer tout cela et de poser différentes limites. Les valeurs dépendent beaucoup de l'utilisation : *pooler* unique pour une seule base, *poolers* multiples pour plusieurs bases, utilisateur applicatif unique ou pas…

Nombre de connexions côté client :

Le paramètre de configuration `max_client_conn` permet d'indiquer le nombre total maximum de connexions clientes à PgBouncer. Sa valeur par défaut est de seulement 100, comme l'équivalent sous PostgreSQL.

Un `max_client_conn` élevé permet d'accepter plus de connexions depuis les applications que n'en offrirait PostgreSQL. Si ce nombre de clients est dépassé, les applications se verront refuser les connexions. En dessous, PgBouncer accepte les connexions, et, au pire, les met en attente. Cela peut arriver si la base PostgreSQL, saturée en connexions, refuse la connexion ; ou si PgBouncer ne peut ouvrir plus de connexions à la base à cause d'une des autres limites ci-dessous. L'application subira donc une latence supplémentaire, mais évitera un refus de connexion qu'elle ne saura pas forcément bien gérer.

`max_db_connections` représente le maximum de connexions, tous utilisateurs confondus, à une base donnée, déclarée dans PgBouncer, donc du point de vue d'un client. Cela peut être modifié dans les chaînes de connexions pour arbitrer entre les différentes bases.

S'il n'y a qu'une base utile, côté serveur comme côté PgBouncer, et que tout l'applicatif passe par ce dernier, `max_db_connections` peut être proche du `max_connections`. Mais il faut laisser un peu de place aux connexions administratives, de supervision, etc.

Connexions côté serveur :

`default_pool_size` est le nombre maximum de connexions PgBouncer/PostgreSQL d'un *pool*. Un *pool* est un couple utilisateur/base de données côté PgBouncer. Il est possible de personnaliser cette valeur base par base, en ajoutant `pool_size=…` dans la chaîne de connexion. Si dans cette même chaîne il y a un paramètre `user` qui impose le nom, il n'y a plus qu'un *pool*.

S'il y a trop de demandes de connexion pour le pool, les transactions sont mises en attente. Cela peut être nécessaire pour équilibrer les ressources entre les différents utilisateurs, ou pour ne pas trop charger le serveur ; mais l'attente peut devenir intolérable pour l'application. Une « réserve » de connexions peut alors être définie avec `reserve_pool_size` : ces connexions sont utilisables dans une situation grave, c'est-à-dire si des connexions se retrouvent à attendre plus d'un certain délai, défini par `reserve_pool_timeout` secondes.

À l'inverse, pour faciliter les montées en charge rapides, `min_pool_size` définit un nombre de connexions qui seront immédiatement ouvertes dès que le pool voit sa première connexion, puis maintenues ouvertes.

Ces deux derniers paramètres peuvent aussi être globaux ou personnalisés dans les chaînes de connexion.

Descripteurs de fichiers :

PgBouncer utilise des descripteurs de fichiers pour les connexions. Le nombre de descripteurs peut être bien plus important que ce que n'autorise par défaut le système d'exploitation. Le maximum théorique est de :

```
max_client_conn + (max_pool_size * nombre de bases * nombre d'utilisateurs)
```

Le cas échéant (en pratique, au-delà de 1000 connexions au pooler), il faudra augmenter le nombre de descripteurs disponibles, sous peine d'erreurs de connexion :

```
ERROR accept() failed: Too many open files
```

Sur Debian et dérivés, un moyen simple est de rajouter cette commande dans `/etc/default/pgbouncer` :

```
ulimit -n 8192
```

Mais plus généralement, il est possible de modifier le service systemd ainsi :

```
sudo systemctl edit pgbouncer
```

ce qui revient à créer un fichier `/etc/systemd/system/pgbouncer.service.d/override.conf` contenant la nouvelle valeur :

```
[Service]
LimitNOFILE=8192
```

Puis il faut redémarrer le pooler :

```
sudo systemctl restart pgbouncer
```

et vérifier la prise en compte dans le fichier de traces de PgBouncer, nommé `pgbouncer.log` (dans `/var/log/postgresql/` sous Debian, `/var/log/pgbouncer/` sur CentOS/Red Hat) :

```
LOG kernel file descriptor limit: 8192 (hard: 8192);
    max_client_conn: 4000, max expected fd use: 6712
```

1.6.9 PgBouncer : Types de connexions

- Mode de multiplexage
  - `pool_mode` (session)
- À la connexion
  - `ignore_startup_parameter = options`
  - attention à `PGOPTIONS` !
- À la déconnexion
  - `server_reset_query`
  - défaut : `DISCARD ALL`

Grâce au paramètre `pool_mode` (dans la chaîne de connexion à la base par exemple), PgBouncer accepte les différents modes de pooling :

- par **session**, pour économiser les temps de (dé)connexion : c'est le défaut ;
- par **transaction**, pour optimiser les connexions en place ;
- par **requête**, notamment si l'on peut se passer des transactions explicites (courant sur plusieurs ordres).

Les restrictions de chaque mode sont listées sur le site[8].

Lorsqu'un client se connecte, il peut utiliser des paramètres de connexion que PgBouncer ne connaît pas ou ne sait pas gérer. Si PgBouncer détecte un paramètre de connexion qu'il ne connaît pas, il rejette purement et simplement la connexion. Le paramètre `ignore_startup_parameters` permet de changer ce comportement, d'ignorer le paramètre et de procéder à la connexion. Par exemple, une variable d'environnement `PGOPTIONS` interdit la connexion depuis psql, il faudra donc définir :

```
ignore_startup_parameters = options
```

ce qui malheureusement réduit à néant l'intérêt de cette variable pour modifier le comportement de PostgreSQL.

À la déconnexion du client, comme la connexion côté PostgreSQL peut être réutilisée par un autre client, il est nécessaire de réinitialiser la session : enlever la configuration de session, supprimer les tables temporaires, supprimer les curseurs, etc. Pour cela, PgBouncer exécute une liste de requêtes configurables ainsi :

```
server_reset_query = DISCARD ALL
```

[8] https://www.pgbouncer.org/features.html

Ce défaut suffira généralement. Il n'est en principe utile qu'en pooling de session, mais peut être forcé en pooling par transaction ou par requête :

```
server_reset_query_always = 1
```

1.6.10 PgBouncer : Instructions préparées

- En mode Transaction
  - prise en charge partielle des instructions préparées
  - depuis la 1.21 : `max_prepared_statements`
  - nécessite un connecteur PG compatible

En mode transactionnel, PgBouncer réutilise les mêmes connexions pour des transactions différentes et simultanées.

Chaque fois qu'une transaction commence (avec un `BEGIN`), se termine (avec un `COMMIT`), ou même lorsqu'une requête ordinaire est exécutée, PgBouncer maintient la même connexion pour un client donné. Mais dès la fin de la transaction, PgBouncer libère la connexion pour réutilisation par la prochaine transaction ou requête. (C'est la différence par rapport au mode session, où PgBouncer attend la fin de la session.)

Le mode transactionnel réduit notablement le nombre de connexions nécessaires.

Cependant, jusqu'à la version 1.21 de PgBouncer, l'utilisation d'instructions préparées n'était pas possible en mode transactionnel. On ne pouvait donc pas bénéficier de leurs avantages tels que la mise en cache des plans de requête.

La version 1.21 de PgBouncer introduit le support des instructions préparées en mode transactionnel, et ceci est transparent pour les clients compatibles.

Le nouveau paramètre `max_prepared_statements` de PgBouncer permet de gérer le nombre d'instructions préparées pour chaque connexion. Cette valeur est par défaut à 0 (désactivé). La valeur 10 est recommandée comme point de départ avant tests. Il faudra ensuite contrôler l'utilisation de la mémoire et du CPU côté PgBouncer. La documentation[9] de PgBouncer fournit des informations permettant d'estimer le besoin en mémoire.

Le mode transactionnel de PgBouncer n'est compatible qu'avec les instructions préparées au niveau du protocole, ce qui peut ne pas fonctionner correctement avec certaines bibliothèques clientes (ou « connecteur »), et en particulier pas pour les commandes d'instructions préparées au niveau SQL (c'est-à-dire `PREPARE`, `EXECUTE` et `DEALLOCATE`, directement transmis à PostgreSQL et non gérés par PgBouncer, à l'exception de `DEALLOCATE ALL` et `DISCARD ALL` qui, eux, sont pris en charge).

[9] https://www.pgbouncer.org/config.html#max_prepared_statements

Ensuite il n'est pas pour le moment possible d'utiliser `DEALLOCATE`, ce qui présente une limitation de la gestion des instructions préparées. (Cette limitation pourrait être levée par l'ajout d'une fonctionnalité à partir de la version 17 de PostgreSQL.) Des erreurs peuvent donc être observées au niveau des clients qui réalise des `DEALLOCATE`.

Vérifiez donc dans la documentation de votre connecteur PostgreSQL la compatibilité avec ce mode. Par exemple, PHP/PDO n'est pour le moment pas compatible[10].

1.6.11 PgBouncer : Durée de vie

- D'une tentative de connexion
  - `client_login_timeout`
  - `server_connect_timeout`
- D'une connexion
  - `server_lifetime`
  - `server_idle_timeout`
  - `client_idle_timeout`
- Pour recommencer une demande de connexion
  - `server_login_retry`
- D'une requête
  - `query_timeout = 0`

PgBouncer dispose d'un grand nombre de paramètres de durée de vie. Ils permettent d'éviter de conserver des connexions trop longues, notamment si elles sont inactives. C'est un avantage sur PostgreSQL qui ne dispose pas de ce type de paramétrage.

Les paramètres en `client_*` concernent les connexions entre le client et PgBouncer, ceux en `server_*` concernent les connexions entre PgBouncer et PostgreSQL.

Il est ainsi possible de libérer plus ou moins rapidement des connexions inutilisées, notamment s'il y a plusieurs *pools* concurrents, ou plusieurs sources de connexions à la base, ou si les pics de connexions sont irréguliers.

Il faut cependant faire attention. Par exemple, interrompre les connexions inactives avec `client_idle_timeout` peut couper brutalement la connexion à une application cliente qui ne s'y attend pas.

[10] https://www.pgbouncer.org/faq.html#how-to-use-prepared-statements-with-transaction-pooling

1.6.12 PgBouncer : Traces

- Fichier
  - `logfile`
- Évènements tracés
  - `log_connections`
  - `log_disconnections`
  - `log_pooler_errors`
- Statistiques
  - `log_stats` (tous les `stats_period` s)

PgBouncer dispose de quelques options de configuration pour les traces.

Le paramètre `logfile` indique l'emplacement (par défaut `/var/log/pgbouncer` sur Red Hat/CentOS, `/var/log/postgres` sur Debian et dérivés). On peut rediriger vers `syslog`.

Ensuite, il est possible de configurer les évènements tracés, notamment les connexions (avec `log_connections`) et les déconnexions (avec `log_disconnections`).

Par défaut, `log_stats` est activé : PgBouncer trace alors les statistiques sur les dernières 60 secondes (paramètres `stats_period`).

```
2020-11-30 19:10:07.839 CET [290804] LOG stats: 54 xacts/s, 380 queries/s,
   in 23993 B/s, out 10128 B/s, xact 304456 us, query 43274 us, wait 14685821 us
```

1.6.13 PgBouncer : Administration

- Pseudo-base `pgbouncer` :

```
sudo -iu postgres psql -h /var/run/postgresql -p 6432 -d pgbouncer
```

- Administration
  - `RELOAD, PAUSE, SUSPEND, RESUME, SHUTDOWN`
- Supervision
  - `SHOW CONFIG|DATABASES|POOLS|CLIENTS|…`
  - `…|SERVERS|STATS|FDS|SOCKETS|…`
  - `…|ACTIVE_SOCKETS|LISTS|MEM`

PgBouncer possède une pseudo-base nommée `pgbouncer`. Il est possible de s'y connecter avec psql ou un autre outil. Il faut pour cela se connecter avec un utilisateur autorisé (déclaration par les paramètres `admin_users` et `stats_users`). Elle permet de répondre à quelques ordres d'administration et de consulter quelques vues.

Les utilisateurs « administrateurs » ont le droit d'exécuter des instructions de contrôle, comme recharger la configuration (`RELOAD`), mettre le système en pause (`PAUSE`), supprimer la pause (`RESUME`), forcer une déconnexion/reconnexion dès que possible (`RECONNECT`, le plus propre en cas de modification de configuration), tuer toutes les sessions d'une base (`KILL`), arrêter PgBouncer (`SHUTDOWN`), etc.

Les utilisateurs statistiques peuvent récupérer des informations sur l'activité de PgBouncer : statistiques sur les bases, les pools de connexions, les clients, les serveurs, etc. avec `SHOW STATS`, `SHOW STATS_AVERAGE`, `SHOW TOTALS`, `SHOW MEM`, etc.

```
# sudo -iu postgres psql -h /var/run/postgresql -p 6432 pgbouncer
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1), serveur 1.14.0/bouncer)

pgbouncer=# SHOW help ;
NOTICE:  Console usage
DÉTAIL :
        SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|USERS|VERSION
        SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
        SHOW DNS_HOSTS|DNS_ZONES
        SHOW STATS|STATS_TOTALS|STATS_AVERAGES|TOTALS
        SET key = arg
        RELOAD
        PAUSE [<db>]
        RESUME [<db>]
```

```
        DISABLE <db>
        ENABLE <db>
        RECONNECT [<db>]
        KILL <db>
        SUSPEND
        SHUTDOWN

pgbouncer=# SHOW DATABASES \gx
-[ RECORD 1 ]-------+-------------------------------------
name                | pgbench_1000_sur_server3
host                | 192.168.74.5
port                | 13002
database            | pgbench_1000
force_user          |
pool_size           | 10
reserve_pool        | 7
pool_mode           | session
max_connections     | 0
current_connections | 17
paused              | 0
disabled            | 0
-[ RECORD 2 ]-------+-------------------------------------
…

pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]-------------------------------------
database   | pgbench_1000_sur_server3
user       | pgbench
cl_active  | 10
cl_waiting | 80
sv_active  | 10
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 835428
pool_mode  | session
-[ RECORD 2 ]-------------------------------------
database   | pgbouncer
user       | pgbouncer
cl_active  | 1
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | statement

pgbouncer=# SHOW STATS \gx
-[ RECORD 1 ]-----+------------------------
database          | pgbench_1000_sur_server3
total_xact_count  | 16444
```

```
total_query_count | 109711
total_received    | 6862181
total_sent        | 3041536
total_xact_time   | 8885633095
total_query_time  | 8873756132
total_wait_time   | 14123238083
avg_xact_count    | 103
avg_query_count   | 667
avg_recv          | 41542
avg_sent          | 17673
avg_xact_time     | 97189
avg_query_time    | 14894
avg_wait_time     | 64038262
-[ RECORD 2 ]-----+-------------------------
database          | pgbouncer
total_xact_count  | 1
total_query_count | 1
total_received    | 0
total_sent        | 0
total_xact_time   | 0
total_query_time  | 0
total_wait_time   | 0
avg_xact_count    | 0
avg_query_count   | 0
avg_recv          | 0
avg_sent          | 0
avg_xact_time     | 0
avg_query_time    | 0
avg_wait_time     | 0
```

```
pgbouncer=# SHOW MEM ;
     name     | size | used | free | memtotal
--------------+------+------+------+----------
 user_cache   |  360 |   11 |   39 |    18000
 db_cache     |  208 |    5 |   73 |    16224
 pool_cache   |  480 |    2 |   48 |    24000
 server_cache |  560 |   17 |   33 |    28000
 client_cache |  560 |   91 | 1509 |   896000
 iobuf_cache  | 4112 |   74 | 1526 |  6579200
```

Toutes ces informations sont utilisées notamment par la sonde Nagios check_postgres[11] pour permettre une supervision de cet outil.

L'outil d'audit pgCluu[12] peut intégrer cette base à ses rapports. Il faudra penser à ajouter la chaîne de connexion à PgBouncer, souvent `--pgbouncer-args='-p 6432'`, aux paramètres de `pgcluu_collectd`.

[11] https://github.com/bucardo/check_postgres
[12] https://pgcluu.darold.net

1.7 CONCLUSION

- Un outil pratique :
  - pour parer à certaines limites de PostgreSQL
  - pour faciliter l'administration
- Limitations généralement tolérables
- Ne jamais installer un pooler sans être certain de son apport :
  - SPOF
  - complexité

1.7.1 Questions

```
SELECT * FROM questions ;
```

1.8 TRAVAUX PRATIQUES

Créer un rôle PostgreSQL nommé **pooler** avec un mot de passe.

Pour mieux suivre les traces, activer `log_connections` et `log_disconnections`, et passer `log_min_duration_statement` à 0.

Installer PgBouncer. Configurer `/etc/pgbouncer/pgbouncer.ini` pour pouvoir se connecter à n'importe quelle base du serveur via PgBouncer (port 6432). Ajouter **pooler** dans `/etc/pgbouncer/userlist.txt`. L'authentification doit être `md5`. Ne pas oublier `pg_hba.conf`. Suivre le contenu de `/var/log/pgbouncer/pgbouncer.log`. Se connecter par l'intermédiaire du pooler sur une base locale.

Activer l'accès à la pseudo-base `pgbouncer` pour les utilisateurs **postgres** et **pooler**. Laisser la session ouverte pour suivre les connexions en cours.

1.8.1 par session

Ouvrir deux connexions sur le pooler. Combien de connexions sont-elles ouvertes côté serveur ?

1.8.2 par transaction

Passer PgBouncer en pooling par transaction. Bien vérifier qu'il n'y a plus de connexions ouvertes.

Rouvrir deux connexions via PgBouncer. Cette fois, combien de connexions sont ouvertes côté serveur ?

Successivement et à chaque fois dans une transaction, créer une table dans une des sessions ouvertes, puis dans l'autre insérer des données. Suivre le nombre de connexions ouvertes. Recommencer avec des transactions simultanées.

1.8.3 par requête

Passer le pooler en mode pooling par requête et tenter d'ouvrir une transaction.

Repasser PgBouncer en pooling par session.

1.8.4 pgbench

> Créer une base nommée `bench` appartenant à **pooler**. Avec pgbench, l'initialiser avec un *scale factor* de 100.

> Lancer des tests (lectures uniquement, avec `--select`) de 60 secondes avec 80 connexions : une fois sur le pooler, et une fois directement sur le serveur. Comparer les performances.

> Refaire ce test en demandant d'ouvrir et fermer les connexions (`-C`), sur le serveur puis sur le pooler. Effectuer un `SHOW POOLS` pendant ce dernier test.

1.9 TRAVAUX PRATIQUES (SOLUTIONS)

> Créer un rôle PostgreSQL nommé **pooler** avec un mot de passe.

Les connexions se feront avec l'utilisateur **pooler** que nous allons créer avec le (trop évident) mot de passe « pooler » :

```
$ createuser --login --pwprompt --echo pooler
Saisir le mot de passe pour le nouveau rôle :
Le saisir de nouveau :
…
CREATE ROLE pooler PASSWORD 'md52a1394e4bcb2e9370746790c13ac33ac'
NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

(NB : le hash sera beaucoup plus complexe si le chiffrement SCRAM-SHA-256 est activé, mais cela ne change rien au principe.)

> Pour mieux suivre les traces, activer `log_connections` et `log_disconnections`, et passer `log_min_duration_statement` à 0.

PostgreSQL trace les rejets de connexion, mais, dans notre cas, il est intéressant de suivre aussi les connexions abouties.

Dans `postgresql.conf` :

```
log_connections = on
log_disconnections = on
log_min_duration_statement = 0
```

Puis on recharge la configuration :

```
sudo systemctl reload postgresql-14
```

En cas de problème, le suivi des connexions dans `/var/lib/pgsql/14/data/log` peut être très pratique.

> Installer PgBouncer. Configurer `/etc/pgbouncer/pgbouncer.ini` pour pouvoir se connecter à n'importe quelle base du serveur via PgBouncer (port 6432). Ajouter **pooler** dans `/etc/pgbouncer/userlist.txt`. L'authentification doit être `md5`. Ne pas oublier `pg_hba.conf`. Suivre le contenu de `/var/log/pgbouncer/pgbouncer.log`. Se connecter par l'intermédiaire du pooler sur une base locale.

L'installation est simple :

```
sudo dnf install pgbouncer
```

La configuration se fait dans `/etc/pgbouncer/pgbouncer.ini`.

Dans la section `[databases]` on spécifie la chaîne de connexion à l'instance, pour toute base :

```
* = host=127.0.0.1 port=5432
```

Il faut ajouter l'utilisateur au fichier `/etc/pgbouncer/userlist.txt`. La syntaxe est de la forme `"user" "hachage du mot de passe"`. La commande `createuser` l'a renvoyé ci-dessus, mais généralement il faudra aller interroger la vue `pg_shadow` ou la table `pg_authid` de l'instance PostgreSQL :

```
SELECT usename,passwd FROM pg_shadow WHERE usename = 'pooler';
```

```
 usename |               passwd
---------+-------------------------------------
 pooler  | md52a1394e4bcb2e9370746790c13ac33ac
```

Le fichier `/etc/pgbouncer/userlist.txt` contiendra donc :

```
"pooler" "md52a1394e4bcb2e9370746790c13ac33ac"
```

Il vaut mieux que seul l'utilisateur système dédié (**pgbouncer** sur Red Hat/CentOS/Rocky Linux) voit ce fichier :

```
sudo chown pgbouncer: userlist.txt
```

De plus il faut préciser dans `pgbouncer.ini` que nous fournissons des mots de passe hachés :

```
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
```

Si ce n'est pas déjà possible, il faut autoriser l'accès de **pooler** en local à l'instance PostgreSQL. Du point de vue de PostgreSQL, les connexions se feront depuis 127.0.0.1 (IP du pooler). Ajouter cette ligne dans le fichier `pg_hba.conf` et recharger la configuration de l'instance :

```
host    all             pooler          127.0.0.1/32            md5
```

```
sudo systemctl reload postgresql-14
```

Enfin, on peut démarrer le pooler :

```
sudo systemctl restart pgbouncer
```

Dans une autre session, on peut suivre les tentatives de connexion :

```
sudo tail -f /var/log/pgbouncer/pgbouncer.log
```

La connexion directement au pooler doit fonctionner :

```
psql -h 127.0.0.1 -p 6432 -U pooler -d postgres
Mot de passe pour l'utilisateur pooler :
psql (14.1)
Saisissez « help » pour l'aide.

postgres=>
```

Dans `pgbouncer.log` :

```
2020-12-02 08:42:35.917 UTC [2208] LOG C-0x152a490: postgres/pooler@127.0.0.1:55096
                                    login attempt: db=postgres user=pooler tls=no
```

Noter qu'en cas d'erreur de mot de passe, l'échec apparaîtra dans ce dernier fichier, et pas dans `postgresql.log`.

> Activer l'accès à la pseudo-base `pgbouncer` pour les utilisateurs **postgres** et **pooler**. Laisser la session ouverte pour suivre les connexions en cours.

```
; comma-separated list of users, who are allowed to change settings
admin_users = postgres,pooler

; comma-separated list of users who are just allowed to use SHOW command
stats_users = stats, postgres,pooler

sudo systemctl reload pgbouncer

$ psql -h 127.0.0.1 -p6432 -U pooler -d pgbouncer
Mot de passe pour l'utilisateur pooler :
psql (14.1, serveur 1.15.0/bouncer)
Saisissez « help » pour l'aide.

pgbouncer=# SHOW HELP ;
NOTICE:  Console usage
DÉTAIL :
        SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|USERS|VERSION
…
```

Si une connexion via PgBouncer est ouverte par ailleurs, on la retrouve ici :

```
pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]---------
database   | pgbouncer
user       | pgbouncer
cl_active  | 1
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | statement
-[ RECORD 2 ]---------
database   | postgres
user       | pooler
cl_active  | 1
cl_waiting | 0
sv_active  | 1
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
```

```
maxwait_us | 0
pool_mode  | session
```

1.9.1 par session

> Ouvrir deux connexions sur le pooler. Combien de connexions sont-elles ouvertes côté serveur ?

Le pooling par session est le mode par défaut de PgBouncer.

On se connecte dans 2 sessions différentes :

```
$ psql -h 127.0.0.1 -p6432 -U pooler -d postgres
psql (14.1)

postgres=>

$ psql -h 127.0.0.1 -p6432 -U pooler -d postgres
…
SELECT COUNT(*) FROM pg_stat_activity
WHERE backend_type='client backend' AND usename='pooler' ;
 count
-------
     2
```

Ici, PgBouncer a donc bien ouvert autant de connexions côté serveur que côté pooler.

1.9.2 par transaction

> Passer PgBouncer en pooling par transaction. Bien vérifier qu'il n'y a plus de connexions ouvertes.

Il faut changer le `pool_mode` dans `pgbouncer.ini`, soit globalement :

```
; When server connection is released back to pool:
;   session      - after client disconnects
;   transaction  - after transaction finishes
;   statement    - after statement finishes
pool_mode = transaction
```

soit dans la définition des connexions :

```
* = host=127.0.0.1 port=5432 pool_mode=transaction
```

En toute rigueur, il n'y a besoin que de recharger la configuration de PgBouncer, mais il y a le problème des connexions ouvertes. Dans notre cas, nous pouvons forcer une déconnexion brutale :

```
sudo systemct restart pgbouncer
```

> Rouvrir deux connexions via PgBouncer. Cette fois, combien de connexions sont ouvertes côté serveur ?

Après reconnexion de 2 sessions, la pseudo-base indique 2 connexions clientes, 1 serveur :

```
pgbouncer=# SHOW POOLS \gx
…
-[ RECORD 2 ]-----------
database   | postgres
user       | pooler
cl_active  | 2
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 1
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Ce que l'on retrouve en demandant directement au serveur :

```
postgres=> SELECT COUNT(*) FROM pg_stat_activity
           WHERE backend_type='client backend' AND usename='pooler' ;
 count
-------
     1
```

> **Successivement** et à chaque fois dans une transaction, créer une table dans une des sessions ouvertes, puis dans l'autre insérer des données. Suivre le nombre de connexions ouvertes. Recommencer avec des transactions simultanées.

Dans la première connexion ouvertes :

```
BEGIN ;
CREATE TABLE log (i timestamptz) ;
COMMIT ;
```

Dans la deuxième :

```
BEGIN ;
INSERT INTO log SELECT now() ;
END ;
```

On a bien toujours une seule connexion :

```
pgbouncer=# SHOW POOLS \gx
…
-[ RECORD 2 ]-----------
database   | postgres
user       | pooler
cl_active  | 2
cl_waiting | 0
```

```
sv_active  | 0
sv_idle    | 0
sv_used    | 1
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Du point de vue du serveur PostgreSQL, tout s'est passé dans la même session (même PID) :

```
… 10:01:45.448 UTC [2841] LOG:  duration: 0.025 ms  statement: BEGIN ;
… 10:01:45.450 UTC [2841] LOG:  duration: 0.631 ms  statement: CREATE TABLE log (i
 ↪  timestamptz) ;
… 10:01:45.454 UTC [2841] LOG:  duration: 4.037 ms  statement: COMMIT ;
… 10:01:49.128 UTC [2841] LOG:  duration: 0.053 ms  statement: BEGIN ;
… 10:01:49.129 UTC [2841] LOG:  duration: 0.338 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
… 10:01:49.763 UTC [2841] LOG:  duration: 4.393 ms  statement: END ;
```

À présent, commençons la seconde transaction avant la fin de la première.

Session 1 :

```
BEGIN ; INSERT INTO log SELECT now() ;
```

Session 2 :

```
BEGIN ; INSERT INTO log SELECT now() ;
```

De manière transparente, une deuxième connexion au serveur a été créée :

```
pgbouncer=# show pools \gx
…
-[ RECORD 2 ]-----------
database   | postgres
user       | pooler
cl_active  | 2
cl_waiting | 0
sv_active  | 2
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Ce que l'on voit dans les traces de PostgreSQL :

```
… 10:05:49.695 UTC [2841] LOG:  duration: 0.144 ms  statement: select 1
… 10:05:49.695 UTC [2841] LOG:  duration: 0.014 ms  statement: BEGIN ;
… 10:05:49.695 UTC [2841] LOG:  duration: 0.110 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
… 10:05:52.320 UTC [2943] LOG:  connection received: host=127.0.0.1 port=50554
… 10:05:52.321 UTC [2943] LOG:  connection authorized: user=pooler database=postgres
```

```
… 10:05:52.323 UTC [2943] LOG:  duration: 0.171 ms  statement: SET
 ↪  application_name='psql';
… 10:05:52.323 UTC [2943] LOG:  duration: 0.015 ms  statement: BEGIN ;
… 10:05:52.324 UTC [2943] LOG:  duration: 0.829 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
```

Du point de l'application, cela a été transparent.

Cette deuxième connexion va rester ouverte, mais elle n'est pas forcément associée à la deuxième session. Cela peut se voir simplement ainsi en demandant le PID du *backend* sur le serveur, qui sera le même dans les deux sessions :

```
postgres=> SELECT pg_backend_pid() ;

 pg_backend_pid
----------------
           2841
```

1.9.3 par requête

> Passer le pooler en mode pooling par requête et tenter d'ouvrir une transaction.

De la même manière que ci-dessus, soit :

```
pool_mode = statement
```

soit :

```
* = host=127.0.0.1 port=5432 pool_mode=statement
```

Redémarrage du pooler :

```
 # systemctl restart pgbouncer
```

Si on essaie de démarrer une transaction :

```
BEGIN;
ERROR:  transaction blocks not allowed in statement pooling mode
la connexion au serveur a été coupée de façon inattendue
        Le serveur s'est peut-être arrêté anormalement avant ou durant le
        traitement de la requête.
La connexion au serveur a été perdue. Tentative de réinitialisation : Succès.
```

Le pooling par requête empêche l'utilisation de transactions.

> Repasser PgBouncer en pooling par session.

Cela revient à revenir au mode par défaut (`pool_mode=session`).

1.9.4 Pgbench

> Créer une base nommée `bench` appartenant à **pooler**. Avec pgbench, l'initialiser avec un *scale factor* de 100.

Le pooler n'est pas configuré pour que **postgres** puisse s'y connecter, il faut donc se connecter directement à l'instance pour créer la base :

```
postgres$ createdb -h /var/run/postgresql -p 5432 --owner pooler  bench
```

La suite peut passer par le pooler :

```
$ /usr/pgsql-14/bin/pgbench -i -s 100 -U pooler -h 127.0.0.1 -p 6432 bench
Password:
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
10000000 of 10000000 tuples (100%) done (elapsed 25.08 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 196.24 s (drop tables 0.00 s, create tables 0.06 s, client-side generate
 ↪  28.00 s,
                  vacuum 154.35 s, primary keys 13.83 s).
```

> Lancer des tests (lectures uniquement, avec `--select`) de 60 secondes avec 80 connexions : une fois sur le pooler, et une fois directement sur le serveur. Comparer les performances.

NB : Pour des résultats rigoureux, pgbench doit être utilisé sur une plus longue durée.

Sur le pooler, on lance :

```
$ /usr/pgsql-14/bin/pgbench \
  --select -T 60 -c 80 -p 6432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 209465
latency average = 22.961 ms
tps = 3484.222638 (including connections establishing)
tps = 3484.278500 (excluding connections establishing)
```

(Ces chiffres ont été obtenus sur un portable avec SSD.)

On recommence directement sur l'instance. (Si l'ordre échoue par saturation des connexions, il faudra attendre que PgBouncer relâche les 20 connexions qu'il a gardées ouvertes.)

```
$ /usr/pgsql-14/bin/pgbench \
  --select -T 60 -c 80 -p 5432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
```

```
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 241482
latency average = 19.884 ms
tps = 4023.255058 (including connections establishing)
tps = 4023.573501 (excluding connections establishing)
```

Le test n'est pas assez rigoureux (surtout sur une petite machine de test) pour dire plus que : les résultats sont voisins.

> Refaire ce test en demandant d'ouvrir et fermer les connexions (`-C`), sur le serveur puis sur le pooler. Effectuer un `SHOW POOLS` pendant ce dernier test.

Sur le serveur :

```
$ /usr/pgsql-14/bin/pgbench \
  -C --select -T 60 -c 80 -p 5432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
Password:
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 9067
latency average = 529.654 ms
tps = 151.041956 (including connections establishing)
tps = 152.922609 (excluding connections establishing)
```

On constate une division par 26 du débit de transactions : le coût des connexions/déconnexions est énorme.

Si on passe par le pooler :

```
$ /usr/pgsql-14/bin/pgbench \
  -C --select -T 60 -c 80 -p 6432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
Password:
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 49926
latency average = 96.183 ms
tps = 831.745556 (including connections establishing)
tps = 841.461561 (excluding connections establishing)
```

On ne retrouve pas les performances originales, mais le gain est tout de même d'un facteur 5, puisque les connexions existantes sur le serveur PostgreSQL sont réutilisées et n'ont pas à être recréées.

Pendant ce dernier test, on peut consulter les connexions ouvertes : il n'y en que 20, pas 80. Noter le grand nombre de celles en attente.

```
pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]---------
database   | bench
user       | pooler
cl_active  | 20
cl_waiting | 54
sv_active  | 20
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 73982
pool_mode  | session
…
```

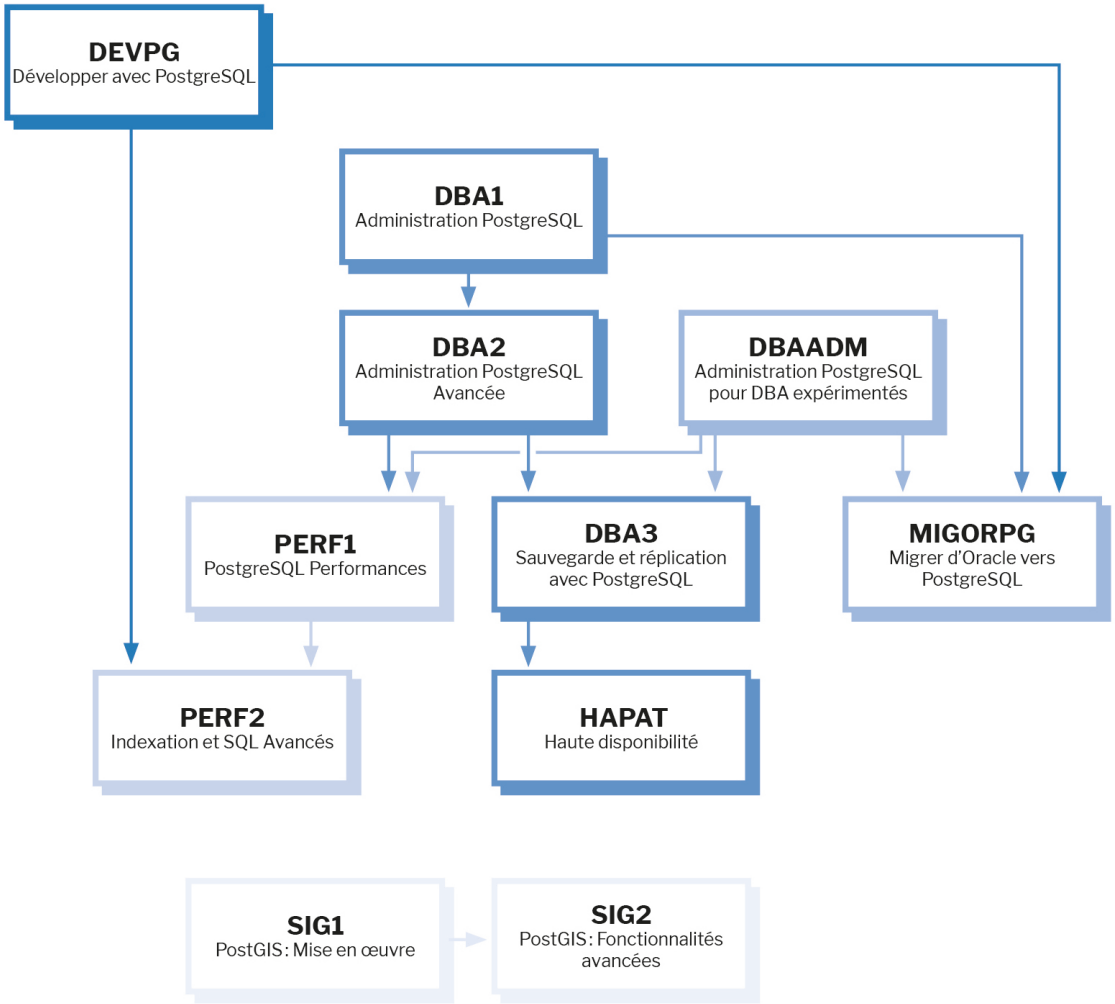
Ces tests n'ont pas pour objectif d'être représentatif mais juste de mettre en évidence le coût d'ouverture/fermeture de connexion. Dans ce cas, le pooler peut apporter un gain très significatif sur les performances.

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.