Formation interne DBA42

Les modules avancés

23.09

Table des matières

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

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.

[1] mailto:formation@dalibo.com

Licence Creative Commons CC-BY-NC-SA

Marques déposées

Sur ce document

Formation	Formation interne DBA42
Titre	Les modules avancés
Révision	23.09
ISBN	N/A
PDF	https://dali.bo/dba42_pdf

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode
[3] https://www.postgresql.org/about/policies/trademarks/

EPUB	https://dali.bo/dba42_epub
HTML	https://dali.bo/dba42_html
Slides	https://dali.bo/dba42_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

0.1 TRAVAUX PRATIQUES

0.1.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Nous travaillons sur la base **cave**. La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Nous allons partitionner la table `stock` sur l'année.

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions) :

```
-- Création de lignes en 2001-2005
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
-- purge des lignes prédédentes
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

0.1.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

0.1.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

NB : Pour le TP, la base sera d'échelle 10 (environ 168 Mo). Des échelles 100 ou 1000 seraient plus réalistes.

Dans une fenêtre en arrière-plan, laisser tourner un processus pgbench avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

0.1.2.2 Partitionnement par *hash*

Pour partitionner la table pgbench_accounts par *hash* sur la colonne aid sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis pgbench_accounts ;
- la substitution de la table partitionnée à la table originale.

Tester et exécuter.

Supprimer l'ancienne table pgbench_accounts_old.

0.1.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

La table pgbench_history se remplit avec le temps. Elle doit être partitionnée par date (champ mtime). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.

- Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.

NB : Pour éviter de coder des dates en dur, il est possible, avec psql, d'utiliser une variable :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;
```

Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

0.1.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

0.1.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre `pgbench_accounts` et `pgbench_history`. Voir les contraintes créées.

Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

0.1.2.6 Index global

On veut créer un index sur `pgbench_history (aid)`.

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

0.2 TRAVAUX PRATIQUES (SOLUTIONS)

0.2.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock
SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock
GROUP BY vin_id, contenant_id, 2001 + annee % 5 ;

DELETE FROM stock WHERE annee<2001 ;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock(LIKE stock_old) PARTITION BY LIST (annee);
```

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

```
CREATE TABLE stock_2001 PARTITION of stock FOR VALUES IN (2001) ;
CREATE TABLE stock_2002 PARTITION of stock FOR VALUES IN (2002) ;
CREATE TABLE stock_2003 PARTITION of stock FOR VALUES IN (2003) ;
CREATE TABLE stock_2004 PARTITION of stock FOR VALUES IN (2004) ;
CREATE TABLE stock_2005 PARTITION of stock FOR VALUES IN (2005) ;
```

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

```
INSERT INTO stock SELECT * FROM stock_old;
```

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

```
ANALYZE;
```

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Append  (cost=0.00..417.36 rows=18192 width=16) (...)
   ->  Seq Scan on stock_2002  (cost=0.00..326.40 rows=18192 width=16) (...)
         Filter: (annee = 2002)
 Planning Time: 0.912 ms
 Execution Time: 21.518 ms
```

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX ON stock (vin_id,contenant_id,annee);
```

Les autres index ne servent à rien sur les partitions : `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index unique que nous venons de créer.

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003 ;
```

```
 Append  (cost=0.29..4.36 rows=3 width=16) (...)
   ->  Index Scan using stock_2003_vin_id_contenant_id_annee_idx on stock_2003 (...)
         Index Cond: ((vin_id = 1725) AND (annee = 2003))
 Planning Time: 1.634 ms
 Execution Time: 0.166 ms
```

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

```
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

```
ERROR:  duplicate key value violates unique constraint
↪  "stock_2004_vin_id_contenant_id_annee_idx"
DETAIL:  Key (vin_id, contenant_id, annee)=(1725, 1, 2004) already exists.
```

C'est une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

```
DELETE FROM stock WHERE annee=2004 and vin_id=1725;

UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

```
DROP TABLE stock_2001 ;
```

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;

ERROR:  no partition of relation "stock" found for row
DETAIL:  Partition key of the failing row contains (annee) = (2006).
```

Il n'existe pas de partition définie pour l'année 2006, cela échoue donc.

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

```
CREATE TABLE stock_default PARTITION OF stock DEFAULT ;

INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;
```

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

```
CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

ERROR:  updated partition constraint for default partition "stock_default"
        would be violated by some row
```

Cela échoue car des enregistrements présents dans la partition par défaut répondent à cette nouvelle contrainte de partitionnement.

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

```
BEGIN ;

ALTER TABLE stock DETACH PARTITION stock_default;

CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

INSERT INTO stock SELECT * FROM stock_default WHERE annee = 2006 ;

DELETE FROM stock_default WHERE annee = 2006 ;

ALTER TABLE stock ATTACH PARTITION stock_default DEFAULT ;

COMMIT ;
```

0.2.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

0.2.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 10 pgbench
```

Dans une fenêtre en arrière-plan, laisser tourner un processus `pgbench` avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

```
$ /usr/pgsql-14/bin/pgbench -n -T3600 -c20 -j2 --debug pgbench
```

L'activité est à ajuster en fonction de la puissance de la machine. Laisser l'affichage défiler dans une fenêtre pour bien voir les blocages.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

Un verrou en attente peut bloquer les opérations d'autres transactions venant après. On peut annuler l'opération à partir d'un certain seuil pour éviter ce phénomène :

```
pgbench=# SET lock_timeout TO '10s' ;
```

Cela ne concerne cependant pas les opérations une fois que les verrous sont acquis. On peut garantir qu'un ordre donné ne durera pas plus d'une certaine durée :

```
SET statement_timeout TO '10s' ;
```

En fonction de la rapidité de la machine et des données à déplacer, cette interruption peut être tolérable ou non.

0.2.2.2 Partitionnement par *hash*

Pour partitionner la table `pgbench_accounts` par *hash* sur la colonne `aid` sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis `pgbench_accounts` ;
- la substitution de la table partitionnée à la table originale.

> Tester et exécuter.

Le champ `aid` n'a pas de signification, un partitionnement par *hash* est adéquat.

Le script peut être le suivant :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

BEGIN ;

-- Nouvelle table partitionnée

CREATE TABLE pgbench_accounts_part (LIKE pgbench_accounts INCLUDING ALL)
PARTITION BY HASH (aid) ;

CREATE TABLE pgbench_accounts_1 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 0 ) ;

CREATE TABLE pgbench_accounts_2 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 1 ) ;

CREATE TABLE pgbench_accounts_3 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 2 ) ;

-- Transfert des données

-- Bloquer les accès à la table le temps du transfert
-- (sinon risque de perte de données !)
LOCK TABLE pgbench_accounts ;

-- Copie des données
INSERT INTO pgbench_accounts_part
SELECT * FROM pgbench_accounts ;

-- Substitution par renommage
ALTER TABLE pgbench_accounts RENAME TO pgbench_accounts_old ;
ALTER TABLE pgbench_accounts_part RENAME TO pgbench_accounts ;

-- Contrôle

\d+

-- On ne validera qu'après contrôle
-- (pendant ce temps les sessions concurrentes restent bloquées !)

COMMIT ;
```

À la moindre erreur, la transaction tombe en erreur. Il faudra demander manuellement ROLLBACK.

Si la durée fixée par `statement_timeout` est dépassée, on aura cette erreur :

```
ERROR:  canceling statement due to statement timeout
Time: 10115.506 ms (00:10.116)
```

Surtout, le traitement pgbench reprend en arrière-plan. On peut alors relancer le script corrigé plus tard.

Si tout se passe bien, un `\d+` renvoie ceci :

```
                          Liste des relations
 Schéma |         Nom          |        Type        | Propriétaire | Taille  | …
--------+----------------------+--------------------+--------------+---------+--
 public | pgbench_accounts     | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_2   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_3   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_old | table              | postgres     | 130 MB  |
 public | pgbench_branches     | table              | postgres     | 136 kB  |
 public | pgbench_history      | table              | postgres     | 5168 kB |
 public | pgbench_tellers      | table              | postgres     | 216 kB  |
```

On peut vérifier rapidement que les valeurs de `aid` sont bien réparties entre les 3 partitions :

```
SELECT aid FROM pgbench_accounts_1 LIMIT 3 ;

 aid
-----
   2
   6
   8

SELECT aid FROM pgbench_accounts_2 LIMIT 3 ;

 aid
-----
   3
   7
  10

SELECT aid FROM pgbench_accounts_3 LIMIT 3 ;

 aid
-----
   1
   9
  11
```

Après la validation du script, on voit apparaître les lignes dans les nouvelles partitions :

```
SELECT relname, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_accounts%' ;

       relname        | n_live_tup
----------------------+------------
 pgbench_accounts_old |    1000002
 pgbench_accounts_1   |     333263
 pgbench_accounts_2   |     333497
 pgbench_accounts_3   |     333240
```

> Supprimer l'ancienne table `pgbench_accounts_old`.

```
DROP TABLE pgbench_accounts_old ;
```

0.2.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

> La table `pgbench_history` se remplit avec le temps. Elle doit être partitionnée par date (champ `mtime`). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.
>
> - Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.
>
> NB : Pour éviter de coder des dates en dur, il est possible, avec `psql`, d'utiliser une variable :
>
> ```
> SELECT ( now()+ interval '60s') AS date_frontiere \gset
> SELECT :'date_frontiere'::timestamptz ;
> ```

La « date frontière » doit être dans le futur (proche). En effet, pgbench va modifier les tables en permanence, on ne sait pas exactement à quel moment la transition aura lieu (et de toute façon on ne maîtrise pas les valeurs de `mtime`) : il continuera donc à écrire dans l'ancienne table, devenue partition, pendant encore quelques secondes.

Cette date est arbitrairement à 1 minute dans le futur, pour dérouler le script manuellement :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
```

Et on peut réutiliser cette variable ainsi ;

```
SELECT :'date_frontiere'::timestamptz ;
```

Le script peut être celui-ci :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;

BEGIN ;

-- Nouvelle table partitionnée
CREATE TABLE pgbench_history_part (LIKE pgbench_history INCLUDING ALL)
PARTITION BY RANGE (mtime) ;

-- Des partitions pour les prochaines minutes
```

```
CREATE TABLE pgbench_history_1
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz  )
             TO (:'date_frontiere'::timestamptz + interval '1min' ) ;

CREATE TABLE pgbench_history_2
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz + interval '1min' )
             TO (:'date_frontiere'::timestamptz + interval '2min' ) ;

-- Au cas où le service perdure au-delà des partitions prévues,
-- on débordera dans cette table
CREATE TABLE pgbench_history_default
PARTITION OF pgbench_history_part DEFAULT ;

-- Jusqu'ici pgbench continue de tourner en arrière plan

-- La table devient une simple partition
-- Ce renommage pose un verrou, les sessions pgbench sont bloquées
ALTER TABLE pgbench_history RENAME TO pgbench_history_orig ;

ALTER TABLE pgbench_history_part
ATTACH PARTITION pgbench_history_orig
FOR VALUES FROM (MINVALUE) TO (:'date_frontiere'::timestamptz) ;

-- Contrôle
\dP

-- Substitution de la table partitionnée à celle d'origine.
ALTER TABLE pgbench_history_part RENAME TO pgbench_history ;

-- Contrôle
\d+ pgbench_history


COMMIT ;
```

> Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

Pour surveiller le contenu des tables jusqu'à la transition :

```
SELECT relname, n_live_tup, now()
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_history%' ;

\watch 3
```

Un \d+ doit renvoyer ceci :

```
                                 Liste des relations
 Schéma |           Nom           |        Type        | Propriétaire | Taille  | …
--------+-------------------------+--------------------+--------------+---------+--
 public | pgbench_accounts        | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1      | table              | postgres     | 44 MB   |
 public | pgbench_accounts_2      | table              | postgres     | 44 MB   |
```

```
 public | pgbench_accounts_3      | table              | postgres     | 44 MB   |
 public | pgbench_branches        | table              | postgres     | 136 kB  |
 public | pgbench_history         | table partitionnée | postgres     | 0 bytes |
 public | pgbench_history_1       | table              | postgres     | 672 kB  |
 public | pgbench_history_2       | table              | postgres     | 0 bytes |
 public | pgbench_history_default | table              | postgres     | 0 bytes |
 public | pgbench_history_orig    | table              | postgres     | 8736 kB |
 public | pgbench_tellers         | table              | postgres     | 216 kB  |
```

0.2.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

```
ALTER TABLE pgbench_history
DETACH PARTITION pgbench_history_orig ;

-- On pourrait faire le DROP directement

DROP TABLE pgbench_history_orig ;
```

0.2.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre pgbench_accounts et pgbench_history. Voir les contraintes créées.

NB : les clés étrangères entre tables partitionnées ne sont pas disponibles avant PostgreSQL 12.

```
SET lock_timeout TO '3s' ;
SET statement_timeout TO '10s' ;

CREATE INDEX ON pgbench_history (aid) ;

ALTER TABLE pgbench_history
ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES
 ↪  pgbench_accounts ;
```

On voit que chaque partition porte un index comme la table mère. La contrainte est portée par chaque partition.

```
pgbench=# \d+ pgbench_history
  Table partitionnée « public.pgbench_history »
 …
Clé de partition : RANGE (mtime)
Index :
    "pgbench_history_aid_idx" btree (aid)
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Partitions: pgbench_history_1 FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                              TO ('2020-02-14 17:42:08.298445'),
            pgbench_history_2 FOR VALUES FROM ('2020-02-14 17:42:08.298445')
                              TO ('2020-02-14 17:43:08.298445'),
            pgbench_history_default DEFAULT
```

```
pgbench=# \d+ pgbench_history_1
   Table « public.pgbench_history_1 »
…
Partition de : pgbench_history FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                                          TO ('2020-02-14 17:42:08.298445')
Contrainte de partition : ((mtime IS NOT NULL)
    AND(mtime >= '2020-02-14 17:41:08.298445'::timestamp without time zone)
    AND (mtime < '2020-02-14 17:42:08.298445'::timestamp without time zone))
Index :
    "pgbench_history_1_aid_idx" btree (aid)
Contraintes de clés étrangères :
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                            FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Méthode d'accès : heap
```

> Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

Le `statement_timeout` peut être un problème :

```
SET
pgbench=# ALTER TABLE pgbench_history
          ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid)
          REFERENCES pgbench_accounts ;
ERROR:  canceling statement due to statement timeout
```

On peut créer les contraintes séparément sur les tables. Cela permet de ne poser un verrou sur la partition active (sans doute `pgbench_history_default`) que pendant le strict minimum de temps (les autres partitions de `pgbench_history` ne sont pas utilisées).

```
SET statement_timeout to '1s' ;
ALTER TABLE pgbench_history_1 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_2 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_default ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

La contrainte au niveau global sera alors posée presque instantanément :

```
ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

0.2.2.6 Index global

> On veut créer un index sur `pgbench_history (aid)`.
>
> Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

Construire un index de manière concurrente (clause `CONCURRENTLY`) permet de ne pas bloquer la table en écriture pendant la création de l'index, qui peut être très longue. Mais il n'est pas possible de le faire sur la table partitionnée :

```
CREATE INDEX CONCURRENTLY ON pgbench_history (aid) ;

ERROR:  cannot create index on partitioned table "pgbench_history" concurrently
```

Mais on peut créer l'index sur chaque partition séparément :

```
CREATE INDEX CONCURRENTLY ON pgbench_history_1 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_2 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_default (aid) ;
```

S'il y a beaucoup de partitions, on peut générer dynamiquement ces ordres :

```
SELECT 'CREATE INDEX CONCURRENTLY ON ' ||
       c.oid::regclass::text || ' (aid) ; '
FROM pg_class c
WHERE relname like 'pgbench_history%' AND relispartition \gexec
```

Comme lors de toute création concurrente, il faut vérifier que les index sont bien valides : la requête suivante ne doit rien retourner.

```
SELECT indexrelid::regclass FROM pg_index WHERE NOT  indisvalid ;
```

Enfin on crée l'index au niveau de la table partitionnée : il réutilise les index existants et sera donc créé presque instantanément :

```
CREATE INDEX ON pgbench_history(aid) ;

pgbench=# \d+ pgbench_history
..
Partition key: RANGE (mtime)
Indexes:
    "pgbench_history_aid_idx" btree (aid)
…
```

1/ Connexions distantes

1.1 ACCÈS À DISTANCE À D'AUTRES SOURCES DE DONNÉES

- Modules historiques : dblink
- SQL/MED & *Foreign Data Wrappers*
- Sharding par fonctions : PL/Proxy
- Le sharding est *Work In Progress*

Nativement, lorsqu'un utilisateur est connecté à une base de données PostgreSQL, sa vision du monde est contenue hermétiquement dans cette base. Il n'a pas accès aux objets des autres bases de la même instance ou d'une autre instance.

Cependant, il existe principalement 3 méthodes pour accéder à des données externes à la base sous PostgreSQL.

La norme SQL/MED est la méthode recommandée pour accéder à des objets distants. Elle permet l'accès à de nombreuses sources de données différentes grâce l'utilisation de connecteurs appelés *Foreign Data Wrappers*.

Historiquement, les utilisateurs de PostgreSQL passaient par l'extension `dblink`, qui permet l'accès à des données externes. Cependant, cet accès ne concerne que des serveurs PostgreSQL. De plus, son utilisation prête facilement à accès moins performant et moins sécurisés que la norme SQL/MED.

PL/Proxy est un cas d'utilisation très différent : cette extension, au départ développée par Skype, permet de distribuer des appels de fonctions PL sur plusieurs nœuds.

Le sharding n'est pas intégré de manière simple à PostgreSQL dans sa version communautaire. Il est déjà possible d'en faire une version primitive avec des partitions basées sur des tables distantes (donc avec SQL/MED), mais nous n'en sommes qu'au début. Des éditeurs proposent des extensions, propriétaires ou expérimentales, ou des *forks* de PostgreSQL dédiés. Comme souvent, il faut se poser la question du besoin réel par rapport à une instance PostgreSQL bien optimisée avant d'utiliser des outils qui vont ajouter une couche supplémentaire de complexité dans votre infrastructure.

1.2 SQL/MED

- *Management of External Data*
- Extension de la norme SQL ISO
- Données externes présentées comme des tables
- Grand nombre de fonctionnalités disponibles
  - mais tous les connecteurs n'implémentent pas tout
- Données accessibles par l'intermédiaire de tables
  - ces tables ne contiennent pas les données localement
  - l'accès à ces tables provoque une récupération des données distantes

SQL/MED est un des tomes de la norme SQL, traitant de l'accès aux données externes (Management of External Data).

Elle fournit donc un certain nombre d'éléments conceptuels, et de syntaxe, permettant la déclaration d'accès à des données externes. Ces données externes sont bien sûr présentées comme des tables.

PostgreSQL suit cette norme et est ainsi capable de requêter des tables distantes à travers des pilotes (appelés *Foreign Data Wrapper*). Les seuls connecteurs livrés par défaut sont `file_fdw` (pour lire des fichiers plats de type CSV accessibles du serveur PostgreSQL) et `postgres_fdw` (qui permet de se connecter à un autre serveur PostgreSQL.

1.2.1 Objets proposés par SQL/MED

- Foreign Data Wrapper
  - connecteur permettant la connexion à un serveur externe et l'exécution de requête
- Foreign Server
  - serveur distant
- User Mapping
  - correspondance d'utilisateur local vers distant
- Foreign Table
  - table distante (ou table externe)

La norme SQL/MED définit quatre types d'objets.

Le *Foreign Data Wrapper* est le connecteur permettant la connexion à un serveur distant, l'exécution de requêtes sur ce serveur, et la récupération des résultats par l'intermédiaire d'une table distante.

Le *Foreign Server* est la définition d'un serveur distant. Il est lié à un *Foreign Data Wrapper* lors de sa création, des options sont disponibles pour indiquer le fichier ou l'adresse IP et le port, ainsi que d'autres informations d'importance pour le connecteur.

Un *User Mapping* permet de définir qui localement a le droit de se connecter sur un serveur distant en tant que tel utilisateur sur le serveur distant. La définition d'un *User Mapping* est optionnel.

Une *Foreign Table* contient la définition de la table distante : nom des colonnes, et type. Elle est liée à un *Foreign Server*.

1.2.2 Foreign Data Wrapper

- Pilote d'accès aux données
- Couverture variable des fonctionnalités
- Qualité variable
- Exemples de connecteurs
  - PostgreSQL, SQLite, Oracle, MySQL (lecture/écriture)
  - fichier CSV, fichier fixe (en lecture)
  - ODBC, JDBC
  - CouchDB, Redis (NoSQL)
- Disponible généralement sous la forme d'une extension
  - ajouter l'extension ajoute le Foreign Data Wrapper à une base

Les trois *Foreign Data Wrappers* les plus aboutis sont sans conteste ceux pour PostgreSQL (disponible en module contrib), Oracle et SQLite. Ces trois pilotes supportent un grand nombre de fonctionnalités (si ce n'est pas toutes) de l'implémentation SQL/MED par PostgreSQL.

De nombreux pilotes spécialisés existent, entre autres pour accéder à des bases NoSQL comme MongDB, CouchDB ou Redis, ou à des fichiers.

Il existe aussi des drivers génériques :

- ODBC : utilisation de driver ODBC
- JDBC : utilisation de driver JDBC

La liste complète des *Foreign Data Wrappers* disponibles pour PostgreSQL peut être consultée sur le wiki de postgresql.org[1]. Encore une fois, leur couverture des fonctionnalités disponibles est très variable ainsi que leur qualité. Il convient de rester prudent et de bien tester ces extensions.

Par exemple, pour ajouter le *Foreign Data Wrapper* pour PostgreSQL, on procédera ainsi :

```
CREATE EXTENSION postgres_fdw;
```

La création cette extension dans une base provoquera l'ajout du *Foreign Data Wrapper* :

```
b1=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

b1=# \dx+ postgres_fdw
     Objects in extension "postgres_fdw"
             Object descriptiong
------------------------------------------
```

[1] https://wiki.postgresql.org/wiki/Foreign_data_wrappers

```
 foreign-data wrapper postgres_fdw
 function postgres_fdw_disconnect(text)
 function postgres_fdw_disconnect_all()
 function postgres_fdw_get_connections()
 function postgres_fdw_handler()
 function postgres_fdw_validator(text[],oid)
(6 rows)

b1=# \dew
                       List of foreign-data wrappers
     Name     |  Owner   |       Handler        |       Validatorg
--------------+----------+----------------------+------------------------
 postgres_fdw | postgres | postgres_fdw_handler | postgres_fdw_validator
(1 row)
```

1.2.3 Fonctionnalités disponibles pour un FDW (1/2)

- Support des lecture de tables (SELECT)
- Support des écriture de tables (y compris TRUNCATE)
  - directement pour INSERT
  - récupération de la ligne en local pour un UPDATE/DELETE
- Envoi sur le serveur distant
  - des prédicats
  - des jointures si les deux tables jointes font partie du même serveur distant
  - des agrégations

L'implémentation SQL/MED permet l'ajout de ces fonctionnalités dans un *Foreign Data Wrapper*. Cependant, une majorité de ces fonctionnalités est optionnelle. Seule la lecture des données est obligatoire.

Les chapitres suivant montrent des exemples de ces fonctionnalités sur deux *Foreign Data Wrappers*.

1.2.4 Fonctionnalités disponibles pour un FDW (2/2)

- Mais aussi
  - support du EXPLAIN
  - support du ANALYZE
  - support de la parallélisation
  - support des exécutions asynchrones (v14)
  - possibilité d'importer un schéma complet

1.2.5 Foreign Server

- Encapsule les informations de connexion
- Le Foreign Data Wrapper utilise ces informations pour la connexion
- Chaque Foreign Data Wrapper propose des options spécifiques
  - nom du fichier pour un FDW listant des fichiers
  - adresse IP, port, nom de base pour un serveur SQL
  - autres

Pour accéder aux données d'un autre serveur, il faut pouvoir s'y connecter. Le Foreign Server regroupe les informations permettant cette connexion : par exemple adresse IP et port.

Voici un exemple d'ajout de serveur distant :

```
CREATE SERVER serveur2
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '192.168.122.1',
           port '5432',
           dbname 'b1') ;
```

1.2.6 User Mapping

- Correspondance utilisateur local / utilisateur distant
- Mot de passe stocké chiffré
- Optionnel
  - aucun intérêt pour les FDW fichiers
  - essentiel pour les FDW de bases de données

Définir un *User Mapping* permet d'indiquer au *Foreign Data Wrapper* quel utilisateur utilisé pour la connexion au serveur distant.

Par exemple, avec cette définition :

```
CREATE USER MAPPING FOR bob SERVER serveur2 OPTIONS (user 'alice', password
↪  'secret');
```

Si l'utilisateur bob local accède à une table distante dépendant du serveur distant `serveur2`, la connexion au serveur distant passera par l'utilisateur `alice` sur le serveur distant.

1.2.7 Foreign Table

- Définit une table distante
- Doit comporter les colonnes du bon type
  - pas forcément toutes
  - pas forcément dans le même ordre
- Peut être une partition d'une table partitionnée
- Possibilité d'importer un schéma complet
  - simplifie grandement la création des tables distantes

Voici un premier exemple pour une table simple :

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    titre       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
```

```
    type        varchar(10),
    duree       interval hour to minute
)
SERVER serveur2 ;
```

Lors de l'accès (avec un SELECT par exemple) à la table `films`, PostgreSQL va chercher la définition du serveur `serveur2`, ce qui lui permettra de connaître le *Foreign Data Wrapper* responsable de la récupération des données et donnera la main à ce connecteur.

Et voici un second exemple, cette fois pour une partition :

```
CREATE FOREIGN TABLE stock202112
    PARTITION OF stock FOR VALUES FROM ('2021-12-01') TO ('2022-01-01')
    SERVER serveur2;
```

Dans ce cas, l'accès à la table partitionnée locale `stock` accédera à des données locales (les autres partitions) mais aussi à des données distantes avec au moins la partition `stock202112`.

Cette étape de création des tables distantes est fastidieuse et peut amener des problèmes si on se trompe sur le nom des colonnes ou sur leur type. C'est d'autant plus vrai que le nombre de tables à créer est important. Dans ce cas, elle peut être avantageusement remplacée par un appel à l'ordre `IMPORT FOREIGN SCHEMA`. Disponible à partir de la version 9.5, il permet l'import d'un schéma complet.

1.2.8 Exemple : file_fdw

Foreign Data Wrapper de lecture de fichiers CSV.

```
CREATE EXTENSION file_fdw;

CREATE SERVER fichier FOREIGN DATA WRAPPER file_fdw ;

CREATE FOREIGN TABLE donnees_statistiques (f1 numeric, f2 numeric)
   SERVER fichier
   OPTIONS (filename  '/tmp/fichier_donnees_statistiques.csv',
            format    'csv',
            delimiter ';') ;
```

Quel que soit le connecteur, la création d'un accès se fait en 3 étapes minimum :

- Installation du connecteur : aucun *Foreign Data Wrapper* n'est présent par défaut. Il se peut que vous ayez d'abord à l'installer sur le serveur au niveau du système d'exploitation.
- Création du serveur : permet de spécifier un certain nombre d'informations génériques à un serveur distant, qu'on n'aura pas à repréciser pour chaque objet de ce serveur.
- Création de la table distante : l'objet qu'on souhaite rendre visible.

Éventuellement, on peut vouloir créer un *User Mapping*, mais ce n'est pas nécessaire pour le FDW `file_fdw`.

En reprenant l'exemple ci-dessus et avec un fichier `/tmp/fichier_donnees_statistiques.csv` contenant les lignes suivantes :

```
1;1.2
2;2.4
3;0
4;5.6
```

Voici ce que donnerait quelques opérations sur cette table distante :

```
SELECT * FROM donnees_statistiques;
```

```
 f1 | f2g
----+-----
  1 | 1.2
  2 | 2.4
  3 |   0
  4 | 5.6
(4 rows)
```

```
SELECT * FROM donnees_statistiques WHERE f1=2;
```

```
 f1 | f2g
----+-----
  2 | 2.4
(1 row)
```

```
EXPLAIN SELECT * FROM donnees_statistiques WHERE f1=2;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Foreign Scan on donnees_statistiques  (cost=0.00..1.10 rows=1 width=64)
   Filter: (f1 = '2'::numeric)
   Foreign File: /tmp/fichier_donnees_statistiques.csv
   Foreign File Size: 25 b
(4 rows)
```

```
postgres=# insert into donnees_statistiques values (5,100.23);
ERROR:  cannot insert into foreign table "donnees_statistiques"
```

1.2.9 Exemple : postgres_fdw

- Pilote le plus abouti, et pour cause
  - il permet de tester les nouvelles fonctionnalités de SQL/MED
  - il sert d'exemple pour les autres FDW
- Propose en plus :
  - une gestion des transactions explicites
  - un pooler de connexions

Nous créons une table sur un serveur distant. Par simplicité, nous utiliserons le même serveur mais une base différente. Créons cette base et cette table :

```
dalibo=# CREATE DATABASE distante;
CREATE DATABASE

dalibo=# \c distante
You are now connected to database "distante" as user "dalibo".

distante=# CREATE TABLE personnes (id integer, nom text);
CREATE TABLE

distante=# INSERT INTO personnes (id, nom) VALUES (1, 'alice'),
                  (2, 'bertrand'), (3, 'charlotte'), (4, 'david');
INSERT 0 4

distante=# ANALYZE personnes;
ANALYZE
```

Maintenant nous pouvons revenir à notre base d'origine et mettre en place la relation avec le « serveur distant » :

```
distante=# \c dalibo
You are now connected to database "dalibo" as user "dalibo".

dalibo=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

dalibo=# CREATE SERVER serveur_distant FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (HOST 'localhost',PORT '5432', DBNAME 'distante');
CREATE SERVER

dalibo=# CREATE USER MAPPING FOR dalibo SERVER serveur_distant
OPTIONS (user 'dalibo', password 'mon_mdp');
CREATE USER MAPPING

dalibo=# CREATE FOREIGN TABLE personnes (id integer, nom text)
SERVER serveur_distant;
CREATE FOREIGN TABLE
```

Et c'est tout ! Nous pouvons désormais utiliser la table distante `personnes` comme si elle était une table locale de notre base.

```
SELECT * FROM personnes;

 id |    nom
----+-----------
  1 | alice
  2 | bertrand
  3 | charlotte
  4 | david

EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes;

                              QUERY PLAN
-----------------------------------------------------------------------
```

```
 Foreign Scan on public.personnes  (cost=100.00..150.95 rows=1365 width=36)
                                   (actual time=0.655..0.657 rows=4 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes
 Total runtime: 1.197 ms
```

En plus, si nous filtrons notre requête, le filtre est exécuté sur le serveur distant, réduisant considérablement le trafic réseau et le traitement associé.

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes WHERE id = 3;

                                QUERY PLAN
-------------------------------------------------------------------------
 Foreign Scan on public.personnes  (cost=100.00..127.20 rows=7 width=36)
                                   (actual time=1.778..1.779 rows=1 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes WHERE ((id = 3))
 Total runtime: 2.240 ms
```

Noter qu'EXPLAIN exige l'option VERBOSE pour afficher le code envoyé à l'instance distante.

Il est possible d'écrire vers ces tables aussi, à condition que le connecteur FDW le permette.

En utilisant l'exemple de la section précédente, on note qu'il y a un aller-retour entre la sélection des lignes à modifier (ou supprimer) et la modification (suppression) de ces lignes :

```
EXPLAIN (ANALYZE, VERBOSE)
UPDATE personnes
SET nom = 'agathe' WHERE id = 1 ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Update on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                             (actual time=2.086..2.086 rows=0 loops=1)
   Remote SQL: UPDATE public.personnes SET nom = $2 WHERE ctid = $1
   ->  Foreign Scan on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                                         (actual time=1.040..1.042 rows=1 loops=1)
         Output: id, 'agathe'::text, ctid
         Remote SQL: SELECT id, ctid FROM public.personnes WHERE ((id = 1))
                     FOR UPDATE
 Total runtime: 2.660 ms

SELECT * FROM personnes;

 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
  1 | agathe
```

On peut aussi constater que l'écriture distante respecte les transactions :

```
dalibo=# BEGIN;
BEGIN

dalibo=# DELETE FROM personnes WHERE id=2;
```

```
DELETE 1

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  3 | charlotte
  4 | david
  1 | agathe
(3 rows)

dalibo=# ROLLBACK;
ROLLBACK

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
  1 | agathe
(4 rows)
```

Attention à ne pas perdre de vue qu'une table distante n'est pas une table locale. L'accès à ses données est plus lent, surtout quand on souhaite récupérer de manière répétitive peu d'enregistrements : on a systématiquement une latence réseau, éventuellement une analyse de la requête envoyée au serveur distant, etc.

Les jointures ne sont pas « poussées » au serveur distant avant PostgreSQL 9.6 et pour des bases PostgreSQL. Un accès par *Nested Loop* (boucle imbriquée entre les deux tables) est habituellement inenvisageable entre deux tables distantes : la boucle interne (celle qui en local serait un accès à une table par index) entraînerait une requête individuelle par itération, ce qui serait horriblement peu performant.

Comme avec tout FDW, il existe des restrictions. Par exemple, avec `postgres_fdw`, un `TRUNCATE` d'une table distante n'est pas possible avant PostgreSQL 14.

Les tables distantes sont donc à réserver à des accès intermittents. Il ne faut pas les utiliser pour développer une application transactionnelle par exemple. Noter qu'entre serveurs PostgreSQL, chaque version améliore les performances (notamment pour « pousser » le maximum d'informations et de critères au serveur distant).

1.2.10 SQL/MED : Performances

- Tous les FDW : vues matérialisées et indexations
- postgres_fdw : `fetch_size`

Pour améliorer les performances lors de l'utilisation de *Foreign Data Wrapper*, une pratique courante est de faire une vue matérialisée de l'objet distant. Les données sont récupérées en bloc et cette vue matérialisée peut être indexée. C'est une sorte de mise en cache. Évidemment cela ne convient pas à toutes les applications.

La documentation de postgres_fdw[2] mentionne plusieurs paramètres, et le plus intéressant pour des requêtes de gros volume est `fetch_size` : la valeur par défaut n'est que de 100, et l'augmenter permet de réduire les aller-retours à travers le réseau.

1.2.11 SQL/MED : héritage

- Une table locale peut hériter d'une table distante et inversement
- Permet le partitionnement sur plusieurs serveurs
- Pour rappel, l'héritage ne permet pas de conserver
  - les contraintes d'unicité et référentielles
  - les index
  - les droits

Cette fonctionnalité utilise le mécanisme d'héritage de PostgreSQL.

Exemple d'une table locale qui hérite d'une table distante

La table parent (ici une table distante) sera la table `fgn_stock_londre` et la table enfant sera la table `local_stock` (locale). Ainsi la lecture de la table `fgn_stock_londre` retournera les enregistrements de la table `fgn_stock_londre` et de la table `local_stock`.

Sur l'instance distante :

Créer une table `stock_londre` sur l'instance distante dans la base nommée « cave » et insérer des valeurs :

[2] https://docs.postgresql.fr/current/postgres-fdw.html

```
CREATE TABLE stock_londre (c1 int);
INSERT INTO stock_londre VALUES (1),(2),(4),(5);
```

Sur l'instance locale :

Créer le serveur et la correspondance des droits :

```
CREATE EXTENSION postgres_fdw ;

CREATE SERVER pgdistant
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.42', port '5432', dbname 'cave');

CREATE USER MAPPING FOR mon_utilisateur
SERVER pgdistant
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Créer une table distante `fgn_stock_londre` correspondant à la table `stock_londre` de l'autre instance :

```
CREATE FOREIGN TABLE fgn_stock_londre (c1 int) SERVER pgdistant
OPTIONS (schema_name 'public' , table_name 'stock_londre');
```

On peut bien lire les données :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
```

```
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
(4 lignes)
```

Voici le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Foreign Scan on fgn_stock_londre  (cost=100.00..197.75 rows=2925 width=4)
                                   (actual time=0.388..0.389 rows=4 loops=1)
```

Créer une table `local_stock` sur l'instance locale qui va hériter de la table mère :

```
CREATE TABLE local_stock () INHERITS (fgn_stock_londre);
```

On insère des valeurs dans la table `local_stock` :

```
INSERT INTO local_stock VALUES  (10),(15);
INSERT 0 2
```

La table `local_stock` ne contient bien que 2 valeurs :

```
SELECT * FROM local_stock ;
```

```
 c1
----
 10
 15
(2 lignes)
```

En revanche, la table `fgn_stock_londre` ne contient plus 4 valeurs mais 6 valeurs :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
```

```
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
 local_stock      | 10
 local_stock      | 15
(6 lignes)
```

Dans le plan d'exécution on remarque bien la lecture des deux tables :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Append  (cost=100.00..233.25 rows=5475 width=4)
         (actual time=0.438..0.444 rows=6 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.438..0.438 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.004..0.005 rows=2 loops=1)
 Planning time: 0.066 ms
 Execution time: 0.821 ms
(5 lignes)
```

Note : Les données de la table `stock_londre` sur l'instance distante n'ont pas été modifiées.

Exemple d'une table distante qui hérite d'une table locale

La table parent sera la table `master_stock` et la table fille (ici distante) sera la table `fgn_stock_londre`. Ainsi une lecture de la table `master_stock` retournera les valeurs de la table `master_stock` et de la table `fgn_stock_londre`, sachant qu'une lecture de la table `fgn_stock_londre` retourne les valeurs de la table `fgn_stock_londre` et `local_stock`. Une lecture de la table `master_stock` retournera les valeurs des 3 tables : `master_stock`, `fgn_stock_londre`, `local_stock`.

Créer une table `master_stock`, insérer des valeurs dedans :

```
CREATE TABLE master_stock (LIKE fgn_stock_londre);
INSERT INTO master_stock VALUES (100),(200);

SELECT tableoid::regclass,* FROM master_stock;
```

```
   tableoid   | c1
--------------+-----
```

```
 master_stock | 100
 master_stock | 200
(2 rows)
```

Modifier la table `fgn_stock_londre` pour qu'elle hérite de la table `master_stock` :

```
ALTER TABLE fgn_stock_londre INHERIT master_stock ;
```

La lecture de la table `master_stock` nous montre bien les valeurs des 3 tables :

```
SELECT tableoid::regclass,* FROM master_stock ;
```

```
     tableoid     | c1
------------------+-----
 master_stock     | 100
 master_stock     | 200
 fgn_stock_londre |   1
 fgn_stock_londre |   2
 fgn_stock_londre |   4
 fgn_stock_londre |   5
 local_stock      |  10
 local_stock      |  15
(8 lignes)
```

Le plan d'exécution confirme bien la lecture des 3 tables :

```
EXPLAIN ANALYSE SELECT * FROM master_stock ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=0.00..236.80 rows=5730 width=4)
         (actual time=0.004..0.440 rows=8 loops=1)
   ->  Seq Scan on master_stock  (cost=0.00..3.55 rows=255 width=4)
                                 (actual time=0.003..0.003 rows=2 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.430..0.430 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.003..0.004 rows=2 loops=1)
 Planning time: 0.073 ms
 Execution time: 0.865 ms
(6 lignes)
```

Dans cet exemple, on a un héritage « imbriqué » :

- la table `master_stock` est parent de la table distante `fgn_stock_londre`
- la table distante `fgn_stock_londre` est enfant de la table `master_stock` et parent de la table `local_stock`
- ma table `local_stock` est enfant de la table distante `fgn_stock_londre`

```
master_stock
  ├─fgn_stock_londre => stock_londre
      ├─local_stock
```

Créons un index sur `master_stock` et ajoutons des données dans la table `master_stock` :

```
CREATE INDEX fgn_idx ON master_stock(c1);
INSERT INTO master_stock (SELECT generate_series(1,10000));
```

Maintenant effectuons une simple requête de sélection :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
 local_stock  | 10
(2 lignes)
```

Étudions le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.29..192.44 rows=27 width=8)
         (actual time=0.010..0.485 rows=2 loops=1)
   ->  Append  (cost=0.29..192.44 rows=27 width=8)
               (actual time=0.009..0.483 rows=2 loops=1)
         ->  Index Scan using fgn_idx on master_stock
                     (cost=0.29..8.30 rows=1 width=8)
                     (actual time=0.009..0.010 rows=1 loops=1)
               Index Cond: (c1 = 10)
         ->  Foreign Scan on fgn_stock_londre
                     (cost=100.00..142.26 rows=13 width=8)
                     (actual time=0.466..0.466 rows=0 loops=1)
         ->  Seq Scan on local_stock  (cost=0.00..41.88 rows=13 width=8)
                                      (actual time=0.007..0.007 rows=1 loops=1)
               Filter: (c1 = 10)
               Rows Removed by Filter: 1
```

L'index ne se fait que sur `master_stock`.

En ajoutant l'option `ONLY` après la clause `FROM`, on demande au moteur de n'afficher que la table `master_stock` et pas les tables filles :

```
SELECT tableoid::regclass,* FROM ONLY master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
(1 ligne)
```

Attention, si on supprime les données sur la table parent, la suppression se fait aussi sur les tables filles :

```
BEGIN;
DELETE FROM master_stock;
-- [DELETE 10008]
SELECT * FROM master_stock ;

 c1
----
```

```
(0 ligne)

ROLLBACK;
```

En revanche avec l'option ONLY, on ne supprime que les données de la table parent :

```
BEGIN;
DELETE FROM ONLY master_stock;
-- [DELETE 10002]
ROLLBACK;
```

Enfin, si nous ajoutons une contrainte CHECK sur la table distante, l'exclusion de partition basées sur ces contraintes s'appliquent naturellement :

```
ALTER TABLE fgn_stock_londre ADD CHECK (c1 < 100);
ALTER TABLE local_stock ADD CHECK (c1 < 100);
    --local_stock hérite de fgn_stock_londre !

EXPLAIN (ANALYZE,verbose) SELECT tableoid::regclass,*g
FROM master_stock WHERE c1=200;

                          QUERY PLAN
---------------------------------------------------------------
 Result  (cost=0.29..8.32 rows=2 width=8)
         (actual time=0.009..0.011 rows=2 loops=1)
   Output: (master_stock.tableoid)::regclass, master_stock.c1
   ->  Append  (cost=0.29..8.32 rows=2 width=8)
               (actual time=0.008..0.009 rows=2 loops=1)
         ->  Index Scan using fgn_idx on public.master_stock
                     (cost=0.29..8.32 rows=2 width=8)
                     (actual time=0.008..0.008 rows=2 loops=1)
               Output: master_stock.tableoid, master_stock.c1
               Index Cond: (master_stock.c1 = 200)
 Planning time: 0.157 ms
 Execution time: 0.025 ms
(8 rows)
```

Attention : La contrainte CHECK sur `fgn_stock_londre` est **locale** seulement. Si cette contrainte n'existe pas sur la table distants, le résultat de la requête pourra alors être faux !

Sur le serveur distant :

```
INSERT INTO stock_londre VALUES (200);
```

Sur le serveur local :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;

   tableoid   | c1
--------------+-----
 master_stock | 200
 master_stock | 200

ALTER TABLE fgn_stock_londre DROP CONSTRAINT fgn_stock_londre_c1_check;

SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;
```

```
     tableoid     | c1
------------------+-----
 master_stock     | 200
 master_stock     | 200
 fgn_stock_londre | 200
```

1.3 DBLINK

- Permet le requêtage inter-bases PostgreSQL
- Simple et bien documenté
- En lecture seule sauf à écrire des triggers sur vue
- Ne transmet pas les prédicats
  - tout l'objet est systématiquement récupéré
- Préférer `postgres_fdw`

Documentation officielle[3].

Le module `dblink` de PostgreSQL a une logique différente de SQL/MED : ce dernier crée des tables virtuelles qui masquent des accès distants, alors qu'avec `dblink`, une requête est fournie à une fonction, qui l'exécute à distance puis renvoie le résultat.

Voici un exemple d'utilisation :

```
SELECT *
FROM dblink('host=serveur port=5432 user=postgres dbname=b1',
            'SELECT proname, prosrc FROM pg_proc')
     AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

L'appel à la fonction `dblink()` va réaliser une connexion à la base `b1` et l'exécution de la requête indiquée dans le deuxième argument. Le résultat de cette requête est renvoyé comme résultat de la fonction. Noter qu'il faut nommer les champs obtenus.

Généralement, on encapsule l'appel à `dblink()` dans une vue, ce qui donnerait par exemple :

```
CREATE VIEW pgproc_b1 AS
  SELECT *
  FROM dblink('host=serveur port=5432 user=postgres dbname=b1',
              'SELECT proname, prosrc FROM pg_proc')
       AS t1(proname name, prosrc text);

SELECT *
FROM pgprocb1
WHERE proname LIKE 'bytea%';
```

Un problème est que, rapidement, on ne se rappelle plus que c'est une table externe et que, même si le résultat contient peu de lignes, tout le contenu de la table distante est récupérés avant que le filtre ne soit exécuté. Donc même s'il y a un index qui aurait pu être utilisé pour ce prédicat, il ne pourra pas être utilisé. Il est rapidement difficile d'obtenir de bonnes performances avec cette extension.

Noter que `dblink` n'est pas aussi riche que son homonyme dans d'autres SGBD concurrents.

[3] https://docs.postgresql.fr/current/contrib-dblink-function.html

De plus, cette extension est un peu ancienne et ne bénéficie pas de nouvelles fonctionnalités sur les dernières versions de PostgreSQL. On préférera utiliser à la place l'implémentation de SQL/MED de PostgreSQL et le *Foreign Data Wrapper* `postgres_fdw` qui évoluent de concert à chaque version majeure et deviennent de plus en plus puissants au fil des versions. Cependant, `dblink` a encore l'intérêt d'émuler des transactions autonomes ou d'appeler des fonctions sur le serveur distant, ce qui est impossible directement avec `postgres_fdw`.

`dblink` fournit quelques fonctions plus évoluées que l'exemple ci-dessus, décrites dans la documentation[4].

[4] https://docs.postgresql.fr/current/dblink.html

1.4 PL/PROXY

- Langage de procédures
  - développée à la base par Skype
- Fonctionnalités
  - connexion à un serveur ou à un ensemble de serveurs
  - exécution de fonctions, pas de requêtes
- Possibilité de distribuer les requêtes
- Utile pour le « partionnement horizontal »
- Uniquement si votre application n'utilise que des appels de fonction
  - dans le cas contraire, il faut revoir l'application

PL/Proxy propose d'exécuter une fonction suivant un mode parmi trois :

- ANY : la fonction est exécutée sur un seul nœud au hasard
- ALL : la fonction est exécutée sur tous les nœuds
- EXACT : la fonction est exécutée sur un nœud précis, défini dans le corps de la fonction

On peut mettre en place un ensemble de fonctions PL/Proxy pour « découper » une table volumineuse et la répartir sur plusieurs instances PostgreSQL.

Le langage PL/Proxy offre alors la possibilité de développer une couche d'abstraction transparente pour l'utilisateur final qui peut alors consulter et manipuler les données comme si elles se trouvaient dans une seule table sur une seule instance PostgreSQL.

On peut néanmoins se demander l'avenir de ce projet. La dernière version date de septembre 2020, et il n'y a eu aucune modification des sources depuis cette version. La société qui a développé ce langage au départ a été rachetée par Microsoft. Le développement du langage dépend donc d'un très petit nombre de contributeurs.

1.5 CONCLUSION

- Privilégier SQL/MED
- `dblink` et PL/Proxy en perte de vitesse
  - à n'utiliser que s'ils résolvent un problème non gérable avec SQL/MED

1.6 TRAVAUX PRATIQUES

1.6.1 Foreign Data Wrapper sur un fichier

But : Lire un fichier extérieur depuis PostgreSQL par un FDW

Avec le *foreign data wrapper* `file_fdw`, créer une table distante qui présente les champs du fichier `/etc/passwd` sous forme de table.

Vérifier son bon fonctionnement avec un simple `SELECT`.

1.6.2 Foreign Data Wrapper sur une autre base

But : Accéder à une autre base par un FDW

Accéder à une table de votre choix d'une autre machine, par exemple `stock` dans la base `cave`, à travers une table distante (`postgres_fdw`) : configuration du `pg_hba.conf`, installation de l'extension dans une base locale, création du serveur, de la table, du mapping pour les droits.

Visualiser l'accès par un `EXPLAIN (ANALYZE VERBOSE) SELECT ...`.

1.7 TRAVAUX PRATIQUES (SOLUTIONS)

1.7.1 Foreign Data Wrapper sur un fichier

> Avec le *foreign data wrapper* `file_fdw`, créer une table distante qui présente les champs du fichier `/etc/passwd` sous forme de table.

> Vérifier son bon fonctionnement avec un simple `SELECT`.

```
CREATE EXTENSION file_fdw;

CREATE SERVER files FOREIGN DATA WRAPPER file_fdw;

CREATE FOREIGN TABLE passwd (
  login text,
  passwd text,
  uid int,
  gid int,
  username text,
  homedir text,
  shell text)
SERVER files
OPTIONS (filename '/etc/passwd', format 'csv', delimiter ':');
```

1.7.2 Foreign Data Wrapper sur une autre base

> Accéder à une table de votre choix d'une autre machine, par exemple `stock` dans la base `cave`, à travers une table distante (`postgres_fdw`) : configuration du `pg_hba.conf`, installation de l'extension dans une base locale, création du serveur, de la table, du mapping pour les droits.

> Visualiser l'accès par un `EXPLAIN (ANALYZE VERBOSE) SELECT ...`.

Tout d'abord, vérifier que la connexion se fait sans mot de passe à la cible depuis le compte **postgres** de l'instance locale vers la base distante où se trouve la table cible.

Si cela ne fonctionne pas, vérifier le `listen_addresses`, le fichier `pg_hba.conf` et le *firewall* de la base distance, et éventuellement le `~postgres/.pgpass` sur le serveur local.

Une fois la connexion en place, dans la base locale voulue, installer le *foreign data wrapper* :

```
CREATE EXTENSION postgres_fdw ;
```

Créer le *foreign server* vers le serveur cible (ajuster les options) :

```
CREATE SERVER serveur_voisin
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.18', port '5432', dbname 'cave');
```

Créer un *user mapping*, c'est-à-dire une correspondance entre l'utilisateur local et l'utilisateur distant :

```
CREATE USER MAPPING FOR mon_utilisateur
SERVER serveur_voisin
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Puis créer la *foreign table* :

```
CREATE FOREIGN TABLE stock_voisin (
vin_id integer, contenant_id integer, annee integer, nombre integer)
SERVER serveur_voisin
OPTIONS (schema_name 'public', table_name 'stock_old');
```

Vérifier le bon fonctionnement :

```
SELECT * FROM stock_voisin WHERE vin_id=12;
```

Vérifier le plan :

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM stock_voisin WHERE vin_id=12 ;
```

Il faut l'option VERBOSE pour voir la requête envoyée au serveur distant. Vous constatez que le prédicat sur `vin_id` a été transmis, ce qui est le principal avantage de cette implémentation sur les DBLinks.

2/ Extensions PostgreSQL pour l'utilisateur

2.1 QU'EST-CE QU'UNE EXTENSION ?

- Pour ajouter :
  - types de données
  - méthodes d'indexation
  - fonctions et opérateurs
  - tables, vues…
- Tous sujets, tous publics
- Intégrées (« contribs ») ou projets externes

Les extensions sont un gros point fort de PostgreSQL. Elles permettent de rajouter des fonctionnalités, aussi bien pour les utilisateurs que pour les administrateurs, sur tous les sujets : fonctions utilitaires, types supplémentaires, outils d'administration avancés, voire applications quasi-complètes. Certaines sont intégrées par le projet, mais n'importe qui peut en proposer et en intégrer une.

2.2 ADMINISTRATION DES EXTENSIONS

Techniquement :

- « packages » pour PostgreSQL, en C, SQL, PL/pgSQL…
- Langages : SQL, PL/pgSQL, C (!)…
- Ensemble d'objets livrés ensemble
- contrib <> extension

Une extension est un objet du catalogue, englobant d'autres objets. On peut la comparer à un paquetage Linux.

Une extension peut provenir d'un projet séparé de PostgreSQL (PostGIS, par exemple, ou le *Foreign Data Wrapper* Oracle).

Les extensions les plus simples peuvent se limiter à quelques objets en SQL, certaines sont en PL/pgSQL, beaucoup sont en C. Dans ce dernier cas, il faut être conscient que la stabilité du serveur est encore plus en jeu !

2.2.1 Installation des extensions

- Packagées ou à compiler
- Par base :
  - `CREATE EXTENSION … CASCADE`
  - `ALTER EXTENSION UPDATE`
  - `DROP EXTENSION`
  - `\dx`
- Listées dans `pg_available_extensions`

Au niveau du système d'exploitation, une extension nécessite des objets (binaires, scripts…) dans l'arborescence de PostgreSQL. De nombreuses extensions sont déjà fournies sous forme de paquets dans les distributions courantes ou par le PGDG, ou encore l'outil PGXN. Dans certains cas, il faudra aller sur le site du projet et l'installer soi-même, ce qui peut nécessiter une compilation.

L'extension doit être ensuite déclarée dans chaque base où elle est jugée nécessaire avec `CREATE EXTENSION nom_extension`. Les scripts fournis avec l'extension vont alors créer les objets nécessaires (vues, procédures, tables…). En cas de désinstallation avec `DROP EXTENSION`, ils seront supprimés. Une extension peut avoir besoin d'autres extensions : l'option `CASCADE` permet de les installer automatiquement.

Le mécanisme couvre aussi la mise à jour des extensions : `ALTER EXTENSION UPDATE` permet de mettre à jour une extension dans PostgreSQL suite à la mise à jour de ses binaires. Cela peut être nécessaire si elle contient des tables à mettre à jour, par exemple. Les versions des extensions disponibles sur le système et celles installées dans la base en cours sont visibles dans la vue `pg_available_extensions`.

Les extensions peuvent être exportées et importées par `pg_dump`/`pg_restore`. Un export par `pg_dump` contient un `CREATE EXTENSION nom_extension`, ce qui permettra de recréer d'éventuelles tables, et le *contenu* de ces tables. Une mise à jour de version majeure, par exemple, permettra donc de migrer les extensions dans leur dernière version installée sur le serveur (changement de prototypes de fonctions, nouvelles vues, etc.).

Sous psql, les extensions présentes dans la base sont visibles avec `\dx` :

```
# \dx
                          Liste des extensions installées
        Nom          | Version |   Schéma   |                    Description
---------------------+---------+------------+-----------------------------------------
 ↪  ----------
 amcheck             | 1.2     | public     | functions for verifying relation integrity
 file_fdw            | 1.0     | public     | foreign-data wrapper for flat file access
 hstore              | 1.6     | public     | data type for storing sets of (key,
  ↪  value) pairs
 pageinspect         | 1.9     | public     | inspect the contents of database pages
  ↪  at...
 pg_buffercache      | 1.3     | public     | examine the shared buffer cache
 pg_prewarm          | 1.2     | public     | prewarm relation data
 pg_rational         | 0.0.1   | public     | bigint fractions
 pg_stat_statements  | 1.10    | public     | track execution statistics of all SQL
  ↪  statements...
 plpgsql             | 1.0     | pg_catalog | PL/pgSQL procedural language
 plpython3u          | 1.0     | pg_catalog | PL/Python3U untrusted procedural language
 postgres_fdw        | 1.0     | public     | foreign-data wrapper for remote
  ↪  PostgreSQL servers
 unaccent            | 1.1     | public     | text search dictionary that removes
  ↪  accents
```

2.3 CONTRIBS - FONCTIONNALITÉS

- Livrées avec le code source de PostgreSQL
- Habituellement packagées (`postgresql-*-contrib`)
- De qualité garantie car maintenues par le projet
- Optionnelles, désactivées par défaut
- Ou en cours de stabilisation
- Documentées : Chapitre F : « Modules supplémentaires fournis »[1]

Une « contrib » est habituellement une extension, sauf quelques exceptions qui ne créent pas d'objets de catalogue (`auto_explain` par exemple). Elles sont fournies directement dans l'arborescence de PostgreSQL, et suivent donc strictement son rythme de révision. Leur compatibilité est ainsi garantie. Les distributions les proposent parfois dans des paquets séparés (`postgresql-contrib-9.6`, `postgresql14-contrib`…), dont l'installation est fortement conseillée.

Il s'agit soit de fonctionnalités qui n'intéressent pas tout le monde (`hstore`, `uuid`, `pg_trgm`, `pgstattuple`…), ou en cours de stabilisation (comme l'autovacuum avant PostgreSQL 8.1), ou à l'inverse de dépréciation (`xml2`).

La documentation des contribs est dans le chapitre F des annexes[2], et est donc fréquemment oubliée par les nouveaux utilisateurs.

[2] https://docs.postgresql.fr/current/contrib.html

2.4 QUELQUES EXTENSIONS

…plus ou moins connues

2.4.1 pgcrypto

Module contrib de chiffrement :

- Nombreuses fonctions pour chiffrer et déchiffrer des données
- Gros inconvénient : oubliez les index sur les données chiffrées !
- N'oubliez pas de chiffrer la connexion (SSL)
- Permet d'avoir une seule méthode de chiffrement pour tout ce qui accède à la base

Fourni avec PostgreSQL, vous permet de chiffrer vos données[3] :

- directement ;
- avec une clé PGP (gérée par exemple avec GnuPG), ce qui est préférable ;
- selon divers algorithmes courants ;
- différemment selon chaque ligne/champ.

Voici un exemple de code:

```
CREATE EXTENSION pgcrypto;
UPDATE utilisateurs SET mdp = crypt('mon nouveau mot de passe',gen_salt('md5'));
INSERT INTO table_secrete (encrypted)
VALUES (pgp_sym_encrypt('mon secret','motdepasse'));
```

L'appel à `gen_salt` permet de rajouter une partie aléatoire à la chaîne à chiffrer, ce qui évite que la même chaîne chiffrée deux fois retourne le même résultat. Cela limite donc les attaques par dictionnaire.

La base effectuant le (dé)chiffrement, cela évite certains allers-retours. Il est préférable que la clé de déchiffrement ne soit pas *dans* l'instance, et soit connue et fournie par l'applicatif. La communication avec cet applicatif doit être sécurisée par SSL pour que les clés et données ne transitent pas en clair.

Un gros inconvénient des données chiffrées dans la table est l'impossibilité complète de les indexer, même avec un index fonctionnel : les données déchiffrées seraient en clair dans cet index ! Une recherche implique donc de parcourir et déchiffrer chaque ligne…

[3] https://docs.postgresql.fr/current/pgcrypto.html

2.4.2 hstore : stockage clé/valeur

- Contrib
- Type `hstore`
- Stockage clé-valeur
- Plus simple que JSON

```
INSERT INTO demo_hstore (meta) VALUES ('river=>t');
SELECT * FROM demo_hstore WHERE meta@>'river=>t';
```

`hstore` fournit un type très simple pour stocker des clés/valeur :

```
CREATE EXTENSION hstore ;

CREATE TABLE demo_hstore(id serial, meta hstore);
INSERT INTO demo_hstore (meta) VALUES ('river=>t');
INSERT INTO demo_hstore (meta) VALUES ('road=>t,secondary=>t');
INSERT INTO demo_hstore (meta) VALUES ('road=>t,primary=>t');
CREATE INDEX idxhstore ON demo_hstore USING gist (meta);

SELECT * FROM demo_hstore WHERE meta@>'river=>t';

 id |     meta
----+--------------
 15 | "river"=>"t"
```

Cette extension a rendu, et rend encore, bien des services. Cependant le type JSON (avec le type binaire `jsonb`) est généralement préféré.

2.4.3 PostgreSQL Anonymizer

- Extension externe (Dalibo)
- Masquage statique et dynamique
- Export anonyme (`pg_dump_anon`)
- Les règles de masquage sont écrites en SQL
- Autodétection de colonnes identifiantes
- Plus simple et plus sûr qu'un ETL

Postgresql Anonymizer[4] est une extension pour masquer ou remplacer les données personnelles[5] dans une base PostgreSQL. Elle est développée par Damien Clochard de Dalibo.

Le projet fonctionne selon une **approche déclarative**, c'est à dire que les règles de masquage[6] sont déclarées directement dans le modèle de données avec des ordres DDL.

Une fois que les règles de masquage sont définies, on peut accéder aux données masquées de 3 façons différentes :

- export anonyme[7] : extraire les données masquées dans un fichier SQL ;
- masquage statique[8] : supprimer une fois pour toutes les données personnelles ;
- masquage dynamique[9] : cacher les données personnelles seulement pour les utilisateurs masqués.

Par ailleurs, l'extension fournit toute une gamme de fonctions de masquage[10] : randomisation, génération de données factices, destruction partielle, brassage, ajout de bruit, etc. On peut également écrire ses propres fonctions de masquage !

Au-delà du masquage, il est également possible d'utiliser une autre approche appelée généralisation[11] qui est bien adaptée pour les statistiques et l'analyse de données.

Enfin, l'extension offre un panel de fonctions de détection[12] qui tentent de deviner quelles colonnes doivent être anonymisées.

Un module de formation lui est consacré[13].

Exemple :

```
=# SELECT * FROM people;

 id | firstname | lastname |   phone
----+-----------+----------+------------
 T1 | Sarah     | Conor    | 0609110911
```

Étape 1 : activer le masquage dynamique

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;
=# SELECT anon.start_dynamic_masking();
```

Étape 2 : déclarer un utilisateur masqué

```
=# CREATE ROLE skynet LOGIN;
=# SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';
```

Étape 3 : déclarer les règles de masquage

[4] https://postgresql-anonymizer.readthedocs.io/en/stable/
[5] https://en.wikipedia.org/wiki/Personally_identifiable_information
[6] https://postgresql-anonymizer.readthedocs.io/en/stable/declare_masking_rules/
[7] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps/
[8] https://postgresql-anonymizer.readthedocs.io/en/latest/static_masking/
[9] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking/
[10] https://postgresql-anonymizer.readthedocs.io/en/latest/masking_functions/
[11] https://postgresql-anonymizer.readthedocs.io/en/stable/generalization/
[12] https://postgresql-anonymizer.readthedocs.io/en/stable/detection/
[13] https://dali.bo/y5_html

```
=# SECURITY LABEL FOR anon ON COLUMN people.lastname
-# IS 'MASKED WITH FUNCTION anon.fake_last_name()';

=# SECURITY LABEL FOR anon ON COLUMN people.phone
-# IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Étape 4 : se connecter avec l'utilisateur masqué

```
=# \c - skynet
=# SELECT * FROM people;

 id | firstname | lastname  |   phone
----+-----------+-----------+------------
 T1 | Sarah     | Stranahan | 06******11
```

2.4.4 PostGIS

- Projet indépendant, GPL, https://postgis.net/
- Module spatial pour PostgreSQL
  - Extension pour types géométriques/géographiques & outils
  - La référence des bases de données spatiales
  - « quelles sont les routes qui coupent le Rhône ? »
  - « quelles sont les villes adjacentes à Toulouse ? »
  - « quels sont les restaurants situés à moins de 3 km de la Nationale 12 ? »

PostGIS ajoute le support d'objets géographiques à PostgreSQL. C'est un projet totalement indépendant développé par la société Refractions Research sous licence GPL, soutenu par une communauté

active, utilisée par des spécialistes du domaine géospatial (IGN, BRGM, AirBNB, Mappy, Openstreetmap, Agence de l'eau…), mais qui peut convenir pour des projets plus modestes.

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

PostGIS permet d'écrire des requêtes de ce type :

```
SELECT restaurants.geom, restaurants.name FROM restaurants
  WHERE EXISTS (SELECT 1 FROM routes
                  WHERE ST_DWithin(restaurants.geom, routes.geom, 3000)
                AND route.name = 'Nationale 12')
```

PostGIS fournit les fonctions d'indexation qui permettent d'accéder rapidement aux objets géométriques, au moyen d'index GiST. La requête ci-dessus n'a évidemment pas besoin de parcourir tous les restaurants à la recherche de ceux correspondant aux critères de recherche.

La liste des fonctionnalités comprend le support des coordonnées géodésiques ; des projections et reprojections dans divers systèmes de coordonnées locaux (Lambert93 en France par exemple) ; des opérateurs d'analyse géométrique (enveloppe convexe, simplification…)

PostGIS est intégré aux principaux serveurs de carte, ETL, et outils de manipulation.

La version 3.0 apporte la gestion du parallélisme, un meilleur support de l'indexation SP-GiST et GiST, ainsi qu'un meilleur support du type GeoJSON.

2.4.5 Mais encore…

- **uuid-ossp** : gérer des UUID
- **unaccent** : supprime des accents
- **citex** : recherche insensible à la casse

2.4.6 Autres extensions connues

- Compatibilité :
  - orafce
- Extensions propriétaires évitant un *fork* :
  - Citus (*sharding*)
  - TimescaleDB (*time series*)
  - être sûr que PostgreSQL a atteint ses limites !

Les extensions permettent de diffuser des bibliothèques de fonction pour la compatibilité avec du code d'autres produits : orafce est un exemple bien connu.

Pour éviter de maintenir un *fork* complet de PostgreSQL, certains éditeurs offrent leur produit sous forme d'extension, souvent avec une version communautaire intégrant les principales fonctionnalités. Par exemple :

- Citus permet du *sharding* ;
- TimescaleDB gère les séries temporelles.

Face à des extensions extérieures, on gardera à l'esprit qu'il s'agit d'un produit supplémentaire à maîtriser et administrer, et l'on cherchera d'abord à tirer le maximum du PostgreSQL communautaire.

2.5 EXTENSIONS POUR DE NOUVEAUX LANGAGES

- PL/pgSQL par défaut
- Ajouter des langages :
  - PL/python
  - PL/perl
  - PL/lua
  - PL/sh
  - PL/R
  - PL/Java
  - etc.

SQL et PL/pgSQL ne sont pas les seuls langages utilisables au niveau d'un serveur PostgreSQL. PL/pgSQL est installé par défaut en tant qu'extension. Il est possible de rajouter les langages python, perl, R, etc. et de coder des fonctions dans ces langages. Ces langages ne sont pas fournis par l'installation standard de PostgreSQL. Une installation via les paquets du système d'exploitation est sans doute le plus simple.

2.6 ACCÈS DISTANTS

Accès à des bases distantes

- Contribs :
  - dblink (ancien)
  - les *foreign data wrappers* : postgresql_fdw, mysql_fdw…
- Sharding :
  - PL/Proxy
  - Citus

Les accès distants à d'autres bases de données sont généralement disponibles par des extensions. L'extension dblink permet d'accéder à une autre instance PostgreSQL mais elle est ancienne, et l'on préférera le *foreign data wrapper* postgresql_fdw, disponible dans les contribs. D'autres FDW sont des projets extérieurs : ora_fdw, mysql_fdw, etc.

Une solution de *sharding* n'est pas encore intégrée à PostgreSQL mais des outils existent : PL/Proxy fournit des fonctions pour répartir des accès mais implique de refondre le code. Citus est une extension plus récente et plus transparente.

2.7 CONTRIBS ORIENTÉS DBA

Accès à des informations ou des fonctions de bas niveau :

- **pg_prewarm** : sauvegarde & restauration de l'état du cache de la base
- **pg_buffercache** : état du cache
- **pgstattuple** (fragmentation des tables et index), **pg_freespacemap** (blocs libres), **pg_visibility** (*visibility map*)
- **pageinspect** : inspection du contenu d'une page
- **pgrowlocks** : informations détaillées sur les enregistrements verrouillés
- **pg_stat_statement** (requêtes normalisées), **auto_explain** (plans)
- **amcheck** : validation des index
- … et de nombreux projets externes

Tous ces modules permettent de manipuler une facette de PostgreSQL à laquelle on n'a normalement pas accès. Leur utilisation est parfois très spécialisée et pointue.

En plus des contribs listés ci-dessus, de nombreux projets externes existent : toastinfo, pg_stat_kcache, pg_qualstats, PoWa, pg_wait_sampling, hypopg…

Pour plus de détails, consulter les modules X2[14] et X3[15].

[14] https://dali.bo/x2_html
[15] https://dali.bo/x3_html

2.8 PGXN

PostgreSQL eXtension Network :

- Site web : pgxn.org[16]
  - nombreuses extensions
  - volontariat
  - aucune garantie de qualité
  - tests soigneux requis
- Et optionnellement client en python pour automatisation de déploiement
- Ancêtre : pgFoundry
- Beaucoup de projets sont aussi sur github

Le site PGXN fournit une vitrine à de nombreux projets gravitant autour de PostgreSQL.

PGXN a de nombreux avantages, dont celui de demander aux projets participants de respecter un certain cahier des charges permettant l'installation automatisée des modules hébergés. Ceci peut par exemple être réalisé avec le client `pgxn` fourni :

```
> pgxn search --dist fdw
multicdr_fdw 1.2.2
    MultiCDR *FDW* ================== Foreign Data Wrapper for representing
    CDR files stream as an external SQL table. CDR files from a directory
    can be read into a table with a specified field-to-column...

redis_fdw 1.0.0
    Redis *FDW* for PostgreSQL 9.1+ ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for the
    Redis key/value database: http://redis.io/ This code is...

jdbc_fdw 1.0.0
    Also,since the JVM being used in jdbc *fdw* is created only once for the
    entire psql session,therefore,the first query issued that uses jdbc
    +fdw* shall set the value of maximum heap size of the JVM(if...

mysql_fdw 2.1.2
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for [MySQL][1]. Please note that this version of mysql_fdw only works
    with PostgreSQL Version 9.3 and greater, for previous version...

www_fdw 0.1.8
    ... library contains a PostgreSQL extension, a Foreign Data Wrapper
    (*FDW*) handler of PostgreSQL which provides easy way for interacting
    with different web-services.

mongo_fdw 2.0.0
    MongoDB *FDW* for PostgreSQL 9.2 ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for
```

```
    MongoDB.

firebird_fdw 0.1.0
    ...  -
    http://www.postgresql.org/docs/current/interactive/postgres-*fdw*.html *
    Other FDWs  - https://wiki.postgresql.org/wiki/*Fdw*  -
    http://pgxn.org/tag/*fdw*/

json_fdw 1.0.0
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for JSON files. The extension doesn't require any data to be loaded into
    the database, and supports analytic queries against array...

postgres_fdw 1.0.0
    This port provides a read-only Postgres *FDW* to PostgreSQL servers in
    the 9.2 series. It is a port of the official postgres_fdw contrib module
    available in PostgreSQL version 9.3 and later.

osm_fdw 3.0.0
    ... "Openstreetmap pbf foreign data wrapper") (*FDW*) for reading
    [Openstreetmap PBF](http://wiki.openstreetmap.org/wiki/PBF_Format
    "Openstreetmap PBF") file format (*.osm.pbf) ## Requirements  *...

odbc_fdw 0.1.0
    ODBC *FDW* (beta) for PostgreSQL 9.1+
    ================================== This PostgreSQL extension implements
    a Foreign Data Wrapper (*FDW*) for remote databases using Open Database
    Connectivity(ODBC)...

couchdb_fdw 0.1.0
    CouchDB *FDW* (beta) for PostgreSQL 9.1+
    ===================================== This PostgreSQL extension
    implements a Foreign Data Wrapper (*FDW*) for the CouchDB document-
    oriented database...

treasuredata_fdw 1.2.14
    ## INSERT INTO statement This *FDW* supports `INSERT INTO` statement.
    With `atomic_import` is `false`, the *FDW* imports INSERTed rows as
    follows.

twitter_fdw 1.1.1
    Installation ------------  $ make && make install  $ psql -c "CREATE
    EXTENSION twitter_fdw" db The CREATE EXTENSION statement creates not
    only *FDW* handlers but also Data Wrapper, Foreign Server, User...

ldap_fdw 0.1.1
    ... is an initial working on a PostgreSQL's Foreign Data Wrapper (*FDW*)
    to query LDAP servers. By all means use it, but do so entirely at your
    own risk! You have been warned! Do you like to use it in...

git_fdw 1.0.2
    # PostgreSQL Git Foreign Data Wrapper [![Build Status](https://travis-
    ci.org/franckverrot/git_fdw.svg?branch=master)](https://travis-
    ci.org/franckverrot/git_fdw) git\_fdw is a Git Foreign Data...

oracle_fdw 2.0.0
```

```
    Foreign Data Wrapper for Oracle ==============================
    oracle_fdw is a PostgreSQL extension that provides a Foreign Data
    Wrapper for easy and efficient access to Oracle databases, including...

foreign_table_exposer 1.0.0
    # foreign_table_exposer This PostgreSQL extension exposes foreign tables
    like a normal table with rewriting Query tree. Some BI tools can't
    detect foreign tables since they don't consider them when...

cstore_fdw 1.6.0
    cstore_fdw ========== [![Build Status](https://travis-
    ci.org/citusdata/cstore_fdw.svg?branch=master)][status] [![Coverage](htt
    p://img.shields.io/coveralls/citusdata/cstore_fdw/master.svg)][coverage]
    ...

multicorn 1.3.5
    [![PGXN version](https://badge.fury.io/pg/multicorn.svg)](https://badge.
    fury.io/pg/multicorn) [![Build
    Status](https://jenkins.dalibo.info/buildStatus/public/Multicorn)]()
    Multicorn =========...

tds_fdw 1.0.7
    # TDS Foreign data wrapper * **Author:** Geoff Montee * **Name:**
    tds_fdw * **File:** tds_fdw/README.md ## About This is a [PostgreSQL
    foreign data...

pmpp 1.2.3
    ... Having foreign server definitions and user mappings makes for
    cleaner function invocations.

file_textarray_fdw 1.0.1
    ### File Text Array Foreign Data Wrapper for PostgreSQL This *FDW* is
    similar to the provided file_fdw, except that instead of the foreign
    table having named fields to match the fields in the data...

floatfile 1.3.0
    Also I'd need to compare the performance of this vs an *FDW*. If I do
    switch to an *FDW*, I'll probably use [Andrew Dunstan's
    `file_text_array_fdw`](https://github.com/adunstan/file_text_array_fdw)
    as a...

pg_pathman 1.4.13
    ... event handling;  * Non-blocking concurrent table partitioning;  *
    +FDW* support (foreign partitions);  * Various GUC toggles and
    configurable settings.
```

Pour peu que le Instant Client d'Oracle soit installé, on peut par exemple lancer :

```
> pgxn install oracle_fdw
INFO: best version: oracle_fdw 1.1.0
INFO: saving /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: unpacking: /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: building extension
gcc -O3 -O0 -Wall -Wmissing-prototypes -Wpointer-arith [...]
[...]
INFO: installing extension
```

```
/usr/bin/mkdir -p '/opt/postgres/lib'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/doc/extension'
/usr/bin/install -c -m 755  oracle_fdw.so '/opt/postgres/lib/oracle_fdw.so'
/usr/bin/install -c -m 644 oracle_fdw.control '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 oracle_fdw--1.1.sql\oracle_fdw--1.0--1.1.sql
                           '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 README.oracle_fdw \
                           '/opt/postgres/share/doc/extension/'
```

Attention : le fait qu'un projet soit hébergé sur PGXN n'est absolument pas une validation de la part du projet PostgreSQL. De nombreux projets hébergés sur PGXN sont encore en phase de développement, voire abandonnés. Il faut avoir le même recul que pour n'importe quel autre brique libre.

2.9 CRÉER SON EXTENSION

- Pas si compliqué
- Peut-être juste quelques fonctions SQL
- Référence : documentation, *Empaqueter des objets dans une extension*[17]
- Exemples SQL et C : blog Dalibo[18]

Il n'est pas très compliqué de créer sa propre extension pour diffuser aisément des outils. Elle peut se limiter à des fonctions en SQL ou PL/pgSQL. Le versionnement des extensions et la facilité de mise à jour peuvent être extrêmement utiles.

Deux exemples de création de fonctions en SQL ou C sont disponibles sur le blog Dalibo[19]. Un autre billet de blog présente une extension utilisable pour l'archivage[20].

La référence reste évidemment la documentation de PostgreSQL, chapitre *Empaqueter des objets dans une extension*[21].

[19] https://blog.dalibo.com/2023/06/08/hackingpg1.html
[20] https://blog.dalibo.com/2023/07/28/hackingpg2.html
[21] https://docs.postgresql.fr/current/extend-extensions.html

2.10 CONCLUSION

- Un nombre toujours plus important d'extension pour étendre les possibilités de PostgreSQL
- Un site central pour les extensions : PGXN.org
- Rajoutez les vôtres !

Cette possibilité d'étendre les fonctionnalités de PostgreSQL est vraiment un atout majeur du projet PostgreSQL. Cela permet de tester des fonctionnalités sans avoir à toucher au moteur de PostgreSQL et risquer des états instables.

Une fois l'extension mature, elle peut être intégrée directement dans le code de PostgreSQL si elle est considérée utile au moteur.

N'hésitez pas à créer vos propres extensions et à les diffuser !

2.10.1 Questions

N'hésitez pas, c'est le moment !

2.11 TRAVAUX PRATIQUES

2.11.1 Masquage statique de données avec PostgreSQL Anonymizer

But : Découverte de l'extension PostgreSQL Anonymizer et du masquage statique

Installer l'extension PostgreSQL Anonymizer en suivant la procédure décrite sur la page Installation[a] de la documentation.

[a] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL/

Créer une table `customer` :

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

Ajouter des individus dans la table :

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;
```

Lire la documentation sur comment déclarer une règle de masquage[a] et placer une règle pour générer un faux nom de famille sur la colonne `lastname`. Déclarer une règle de masquage statique sur la colonne `lastname` et l'appliquer. Vérifier le contenu de la table.

[a] https://postgresql-anonymizer.readthedocs.io/en/latest/declare_masking_rules/

Réappliquer le masquage statique[a]. Qu'observez-vous ?

[a] https://postgresql-anonymizer.readthedocs.io/en/latest/static_masking/

2.11.2 Masquage dynamique de données avec PostgreSQL Anonymizer

But : Mettre en place un masquage dynamique avec PostgreSQL Anonymizer

> Parcourir la liste des fonctions de masquage[a] et écrire une règle pour cacher partiellement le numéro de téléphone. Activer le masquage dynamique. Appliquer le masquage dynamique uniquement sur la colonne phone pour un nouvel utilisateur nommé **soustraitant**.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/latest/masking_functions/

2.11.3 Masquage statique de données avec PostgreSQL Anonymizer

> Installer l'extension PostgreSQL Anonymizer en suivant la procédure décrite sur la page Installation[a] de la documentation.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL/

Sur Rocky Linux ou autre dérivé Red Hat, depuis les dépôts du PGDG :

```
sudo dnf install postgresql_anonymizer_14
```

Au besoin, remplacer 14 par la version de l'instance PostgreSQL.

La base de travail ici se nomme **sensible**. Se connecter à l'instance pour initialiser l'extension :

```
ALTER DATABASE sensible SET session_preload_libraries = 'anon' ;
```

Après reconnexion à la base **sensible** :

```
CREATE EXTENSION anon CASCADE;

SELECT anon.init();  -- ne pas oublier !
```

> Créer une table `customer` :

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

> Ajouter des individus dans la table :

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;

SELECT * FROM customer ;
```

```
 id  | firstname | lastname  |    phone     |   birth    | postcode
-----+-----------+-----------+--------------+------------+----------
 107 | Sarah     | Conor     | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Skywalker |              | 1951-09-25 | 90120
 341 | Don       | Draper    | 347-515-3423 | 1926-06-01 | 04520
```

> Lire la documentation sur comment déclarer une règle de masquage[a] et placer une règle pour générer un faux nom de famille sur la colonne `lastname`. Déclarer une règle de masquage statique sur la colonne `lastname` et l'appliquer. Vérifier le contenu de la table.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/latest/declare_masking_rules/

```
SECURITY LABEL FOR anon ON COLUMN customer.lastname
IS 'MASKED WITH FUNCTION anon.fake_last_name()' ;
```

Si on consulte la table avec :

```
SELECT * FROM customer ;
```

les données ne sont pas encore masquées car la règle n'est pas appliquée. L'application se fait avec :

```
SELECT anon.anonymize_table('customer') ;
```

```
SELECT * FROM customer;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Waelchi  | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Lemke    |              | 1951-09-25 | 90120
 341 | Don       | Shanahan | 347-515-3423 | 1926-06-01 | 04520
```

NB : les données de la table ont ici bien été modifiées sur le disque.

> Réappliquer le masquage statique[a]. Qu'observez-vous ?
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/latest/static_masking/

Si l'on relance l'anonymisation plusieurs fois, les données factices vont changer car la fonction `fake_last_name()` renvoie des valeurs différentes à chaque appel.

```
SELECT anon.anonymize_table('customer');
```

```
SELECT * FROM customer;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Smith    | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Sanford  |              | 1951-09-25 | 90120
 341 | Don       | Goldner  | 347-515-3423 | 1926-06-01 | 04520
```

2.11.4 Masquage dynamique de données avec PostgreSQL Anonymizer

> Parcourir la liste des fonctions de masquage[a] et écrire une règle pour cacher partiellement le numéro de téléphone. Activer le masquage dynamique. Appliquer le masquage dynamique uniquement sur la colonne phone pour un nouvel utilisateur nommé **soustraitant**.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/latest/masking_functions/

```
SELECT anon.start_dynamic_masking();

SECURITY LABEL FOR anon ON COLUMN customer.phone
IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)';

SELECT anon.anonymize_column('customer','phone');

SELECT * FROM customer ;
```

Les numéros de téléphone apparaissent encore car ils ne sont pas masqués à l'utilisateur en cours. Il faut le déclarer pour les utilisateurs concernés :

```
CREATE ROLE soustraitant LOGIN ;
\password soustraitant

GRANT SELECT ON customer TO soustraitant ;
SECURITY LABEL FOR anon ON ROLE soustraitant  IS 'MASKED';
```

Ce nouvel utilisateur verra à chaque fois des noms différents (masquage dynamique), et des numéros de téléphone partiellement masqués :

```
\c sensible soustraitant
SELECT * FROM customer ;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Kovacek  | 06X-XXX-XX11 | 1965-10-10 | 90016
 258 | Luke      | Effertz  | ø            | 1951-09-25 | 90120
 341 | Don       | Turcotte | 34X-XXX-XX23 | 1926-06-01 | 04520
```

Pour consulter la configuration de masquage en place, utiliser une des vues fournies dans le schéma anon :

```
=# SELECT * FROM anon.pg_masks \gx

-[ RECORD 1 ]----+-----------------------------------------------------------
attrelid         | 41853
attnum           | 3
relnamespace     | public
relname          | customer
attname          | lastname
format_type      | text
col_description  | MASKED WITH FUNCTION anon.fake_last_name()
masking_function | anon.fake_last_name()
masking_value    |
priority         | 100
masking_filter   | anon.fake_last_name()
trusted_schema   | t
-[ RECORD 2 ]----+-----------------------------------------------------------
attrelid         | 41853
```

```
attnum            | 4
relnamespace      | public
relname           | customer
attname           | phone
format_type       | text
col_description   | MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)
masking_function | anon.partial(phone,2,$$X-XXX-XX$$,2)
masking_value     |
priority          | 100
masking_filter    | anon.partial(phone,2,$$X-XXX-XX$$,2)
trusted_schema    | t
```

3/ Extensions PostgreSQL pour les DBA

3.1 PRÉAMBULE

- Nombreuses extensions pour observer le comportement de PostgreSQL
- Contribs ou projets externes

De nombreux permettent de manipuler une facette de PostgreSQL à laquelle on n'a normalement pas accès. Leur utilisation est parfois très spécialisée et pointue.

3.2 PGSTATTUPLE

pgstattuple fournit une mesure (par parcours complet de l'objet) sur:

- Pour une table
  - remplissage des blocs
  - enregistrements morts
  - espace libre
- Pour un index
  - profondeur de l'index
  - remplissage des feuilles
  - fragmentation (feuilles non consécutives)

Par exemple :

```
# CREATE EXTENSION
# SELECT * FROM pgstattuple('dspam_token_data');

-[ RECORD 1]-----------------------
 table_len          | 601743360
 tuple_count        | 8587417
 tuple_len          | 412196016
 tuple_percent      | 68.5
 dead_tuple_count   | 401098
 dead_tuple_len     | 19252704
 dead_tuple_percent | 3.2
 free_space         | 93370000
 free_percent       | 15.52

# SELECT * FROM pgstatindex('dspam_token_data_uid_key');

-[ RECORD 1]-----------------------
 version            | 2
 tree_level         | 2
 index_size         | 429047808
 root_block_no      | 243
 internal_pages     | 244
 leaf_pages         | 52129
 empty_pages        | 0
 deleted_pages      | 0
 avg_leaf_density   | 51.78
 leaf_fragmentation | 43.87
```

Comme chaque interrogation nécessite une lecture complète de l'objet, ces fonctions ne sont pas à appeler en supervision.

Elles servent de façon ponctuelle pour s'assurer qu'un objet nécessite une réorganisation. Ici, l'index

`dspam_token_data_uid_key` pourrait certainement être reconstruit… il deviendrait 40 % plus petit environ (remplissage à 51 % au lieu de 90 %).

`leaf_fragmentation` indique le pourcentage de pages feuilles qui ne sont pas physiquement contiguës sur le disque. Cela peut être important dans le cas d'un index utilisé pour des `Range Scans` (requête avec des inégalités), mais n'a aucune importance ici puisqu'il s'agit d'une clé primaire technique, donc d'un index qui n'est interrogé que pour récupérer des enregistrements de façon unitaire.

3.3 PG_FREESPACEMAP

La *freespacemap* :

- est renseignée par VACUUM, par objet (table/index)
- est consommée par les sessions modifiant des données (INSERT/UPDATE)
- est interrogée la freespacemap pour connaître l'espace libre
- est rarement utilisée (doute sur l'efficacité de VACUUM)

Voici deux exemples d'utilisation de `pg_freespacemap` :

```
dspam=# SELECT * FROM pg_freespace('dspam_token_data') LIMIT 20;
```

```
 blkno | avail
-------+-------
     0 |    32
     1 |     0
     2 |     0
     3 |    32
     4 |     0
     5 |     0
     6 |     0
     7 |     0
     8 |    32
     9 |    32
    10 |    32
    11 |     0
    12 |     0
    13 |     0
    14 |     0
    15 |     0
    16 |     0
    17 |     0
    18 |    32
    19 |    32
```

```
dspam=# SELECT * FROM pg_freespace('dspam_token_data') ORDER BY avail DESC
LIMIT 20;
```

```
 blkno | avail
-------+-------
 67508 |  7520
 67513 |  7520
 67460 |  7520
 67507 |  7520
 67451 |  7520
 67512 |  7520
 67452 |  7520
 67454 |  7520
 67505 |  7520
 67447 |  7520
```

```
67324 |  7520
67443 |  7520
67303 |  7520
67509 |  7520
67444 |  7520
67448 |  7520
67445 |  7520
66888 |  7520
67516 |  7520
67514 |  7520
```

L'interprétation de « avail » est un peu complexe, et différente suivant qu'on inspecte une table ou un index. Il est préférable de se référer à la documentation.

3.4 PG_VISIBILITY

La *Visibility Map* :

- Est renseignée par VACUUM, par table
- Permet de savoir que l'ensemble des enregistrements de ce bloc est visible
- Indispensable pour les parcours d'index seul
- Interroger la *visibility map* permet de voir si un bloc est :
  - visible
  - gelé
- Rarement utilisé

On crée une table de test avec 451 lignes :

```
CREATE TABLE test_visibility AS SELECT generate_series(0,450) x;
SELECT 451
```

On regarde dans quel état est la *visibility map* :

```
SELECT oid FROM pg_class WHERE relname='test_visibility' ;
```

```
  oid
-------
 18370
```

```
SELECT * FROM pg_visibility(18370);
```

```
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | f           | f          | f
     1 | f           | f          | f
```

Les deux blocs que composent la table `test_visibility` sont à `false`, ce qui est normal puisque l'opération de vacuum n'a jamais été exécutée sur cette table.

On lance donc une opération de vacuum :

```
VACUUM VERBOSE test_visibility ;
```

```
INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables,
                             451 non supprimables
parmi 2 pages sur 2
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 0 pointeurs d éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

Vacuum voit bien nos 451 lignes, et met donc la *visibility map* à jour. Lorsqu'on la consulte, on voit bien que toutes les lignes sont visibles de toutes les transactions :

```
SELECT * FROM pg_visibility(33259);
```

```
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | t           | f          | t
```

La colonne `all_frozen` passera à `t` après un `VACUUM FREEZE`.

3.5 PAGEINSPECT

- Vision du contenu d'un bloc
- Sans le dictionnaire, donc sans décodage des données
- Affichage brut
- Utilisé surtout en debug, ou dans les cas de corruption
- Fonctions de décodage pour les tables, les index (B-tree, hash, GIN, GiST), FSM
- Nécessite de connaître le code de PostgreSQL

Voici quelques exemples :

Contenu d'une page d'une table :

```
# SELECT * FROM heap_page_items(get_raw_page('dspam_token_data',0)) LIMIT 5;

 lp | lp_off | lp_flags | lp_len |   t_xmin   | t_xmax | t_field3 | t_ctid
----+--------+----------+--------+------------+--------+----------+--------
  1 |    201 |        2 |      0 |            |        |          |
  2 |   1424 |        1 |     48 | 1439252980 |      0 |        0 | (0,2)
  3 |    116 |        2 |      0 |            |        |          |
  4 |   7376 |        1 |     48 |          2 |      0 |      140 | (0,4)
  5 |   3536 |        1 |     48 | 1392499801 |      0 |        0 | (0,5)

 lp | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+-------------+------------+--------+--------+-------
  1 |             |            |        |        |
  2 |           5 |       2304 |     24 |        |
  3 |             |            |        |        |
  4 |           5 |      10496 |     24 |        |
  5 |           5 |       2304 |     24 |        |
```

Et son entête :

```
# SELECT * FROM page_header(get_raw_page('dspam_token_data',0));

-[ RECORD 1 ]-------------------------------------------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 1
lower     | 852
upper     | 896
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 1450780148
```

Méta-données d'un index (contenu dans la première page) :

```
# SELECT * FROM bt_metap('dspam_token_data_uid_key');
```

```
 magic  | version | root | level | fastroot | fastlevel
--------+---------+------+-------+----------+-----------
 340322 |       2 |  243 |     2 |      243 |         2
```

La page racine est la 243. Allons la voir :

```
# SELECT * FROM bt_page_items('dspam_token_data_uid_key',243) LIMIT 10;

 offset |   ctid    | len | nulls | vars |                 data
--------+-----------+-----+-------+------+--------------------------------------
      1 | (3,1)     |   8 | f     | f    |
      2 | (44565,1) |  20 | f     | f    | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (242,1)   |  20 | f     | f    | 77 c6 0d 6f a6 92 db 81 28 00 00 00
      4 | (43569,1) |  20 | f     | f    | 47 a6 aa be 29 e3 13 83 18 00 00 00
      5 | (481,1)   |  20 | f     | f    | 30 17 dd 8e d9 72 7d 84 0a 00 00 00
      6 | (43077,1) |  20 | f     | f    | 5c 3c 7b c5 5b 7a 4e 85 0a 00 00 00
      7 | (719,1)   |  20 | f     | f    | 0d 91 d5 78 a9 72 88 86 26 00 00 00
      8 | (41209,1) |  20 | f     | f    | a7 8a da 17 95 17 cd 87 0a 00 00 00
      9 | (957,1)   |  20 | f     | f    | 78 e9 64 e9 64 a9 52 89 26 00 00 00
     10 | (40849,1) |  20 | f     | f    | 53 11 e9 64 e9 1b c3 8a 26 00 00 00
```

La première entrée de la page 243, correspondant à la donnée `f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00` est stockée dans la page 3 de notre index :

```
# SELECT * FROM bt_page_stats('dspam_token_data_uid_key',3);

-[ RECORD 1]------------------
blkno          | 3
type           | i
live_items     | 202
dead_items     | 0
avg_item_size  | 19
page_size      | 8192
free_size      | 3312
btpo_prev      | 0
btpo_next      | 44565
btpo           | 1
btpo_flags     | 0

# SELECT * FROM bt_page_items('dspam_token_data_uid_key',3) LIMIT 10;

 offset |   ctid    | len | nulls | vars |                 data
--------+-----------+-----+-------+------+--------------------------------------
      1 | (38065,1) |  20 | f     | f    |  f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      2 | (1,1)     |   8 | f     | f    |
      3 | (37361,1) |  20 | f     | f    | 30 fd 30 b8 70 c9 01 80 26 00 00 00
      4 | (2,1)     |  20 | f     | f    | 18 2c 37 36 27 03 03 80 27 00 00 00
      5 | (4,1)     |  20 | f     | f    | 36 61 f3 b6 c5 1b 03 80 0f 00 00 00
      6 | (43997,1) |  20 | f     | f    | 30 4a 32 58 c8 44 03 80 27 00 00 00
      7 | (5,1)     |  20 | f     | f    | 88 fe 97 6f 7e 5a 03 80 27 00 00 00
      8 | (51136,1) |  20 | f     | f    | 74 a8 5a 9b 15 5d 03 80 28 00 00 00
      9 | (6,1)     |  20 | f     | f    | 44 41 3c ee c8 fe 03 80 0a 00 00 00
     10 | (45317,1) |  20 | f     | f    | d4 b0 7c fd 5d 8d 05 80 26 00 00 00
```

Le type de la page est `i`, c'est-à-dire « internal », donc une page interne de l'arbre. Continuons notre descente, allons voir la page 38065 :

```
# SELECT * FROM bt_page_stats('dspam_token_data_uid_key',38065);

-[ RECORD 1]----------------------
blkno          | 38065
type           | l
live_items     | 169
dead_items     | 21
avg_item_size  | 20
page_size      | 8192
free_size      | 3588
btpo_prev      | 118
btpo_next      | 119
btpo           | 0
btpo_flags     | 65

# SELECT * FROM bt_page_items('dspam_token_data_uid_key',38065) LIMIT 10;

 offset |    ctid     | len | nulls | vars |                data
--------+-------------+-----+-------+------+-----------------------------------
      1 | (11128,118) |  20 | f     | f    | 33 37 89 95 b9 23 cc 80 0a 00 00 00
      2 | (45713,181) |  20 | f     | f    | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (45424,97)  |  20 | f     | f    | f3 4b 2e 8c 39 a3 cb 80 26 00 00 00
      4 | (45255,28)  |  20 | f     | f    | f3 4b 2e 8c 39 a3 cb 80 27 00 00 00
      5 | (15672,172) |  20 | f     | f    | f3 4b 2e 8c 39 a3 cb 80 28 00 00 00
      6 | (5456,118)  |  20 | f     | f    | f3 bf 29 a2 39 a3 cb 80 0f 00 00 00
      7 | (8356,206)  |  20 | f     | f    | f3 bf 29 a2 39 a3 cb 80 28 00 00 00
      8 | (33895,272) |  20 | f     | f    | f3 4b 8e 37 99 a3 cb 80 0a 00 00 00
      9 | (5176,108)  |  20 | f     | f    | f3 4b 8e 37 99 a3 cb 80 0f 00 00 00
     10 | (5466,41)   |  20 | f     | f    | f3 4b 8e 37 99 a3 cb 80 26 00 00 00
```

Nous avons trouvé une feuille (type `l`). Les ctid pointés sont maintenant les adresses dans la table :

```
# SELECT * FROM dspam_token_data WHERE ctid = '(11128,118)';

 uid |        token         | spam_hits | innocent_hits |  last_hit
-----+----------------------+-----------+---------------+------------
  40 | -6317261189288392210 |         0 |             3 | 2014-11-10
```

3.6 PGROWLOCKS

Les verrous mémoire de PostgreSQL ne verrouillent pas les enregistrements :

- Il est parfois compliqué de comprendre qui verrouille qui, à cause de quel enregistrement
- pgrowlocks inspecte une table pour détecter les enregistrements verrouillés, leur niveau de verrouillage, et qui les verrouille
- scan complet de la table !

Par exemple :

```
# SELECT * FROM pgrowlocks('dspam_token_data');

 locked_row |   locker   | multi |     xids     |       modes        | pids
------------+------------+-------+--------------+--------------------+-------
 (0,2)      | 1452109863 | f     | {1452109863} | {"No Key Update"} | {928}
```

Nous savons donc que l'enregistrement (0,2) est verrouillé par le pid 928. Nous avons le mode de verrouillage, le (ou les) numéro de transaction associés. Un enregistrement peut être verrouillé par plus d'une transaction dans le cas d'un `SELECT FOR SHARE`. Dans ce cas, PostgreSQL crée un « multixact » qui est stocké dans `locker`, `multi` vaut `true`, et `xids` contient plus d'un enregistrement. C'est un cas très rare d'utilisation.

3.7 GESTION DU CACHE

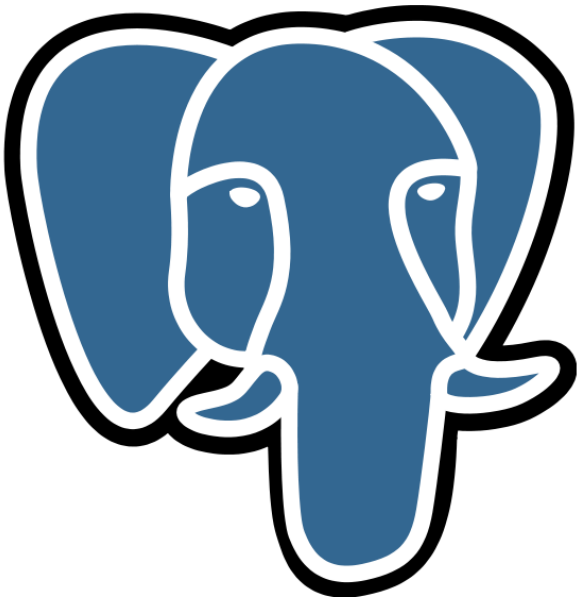
- pg_buffercache : voir ce qu'il y a dans mes *shared buffers*
- pg_prewarm : forcer le chargement du cache

pg_buffercache et pg_prewarm sont des extensions déjà connues de beaucoup de DBA.

Rappelons que pg_buffercache permet de lister chaque bloc dans le cache de PostgreSQL, et de savoir notamment s'il est *dirty*.

pg_prewarm est lui très utile pour forcer le chargement d'un objet dans le cache de PostgreSQL ou de l'OS, y compris automatiquement au démarrage.

4/ Pooling



4.1 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.

4.1.1 Objectifs

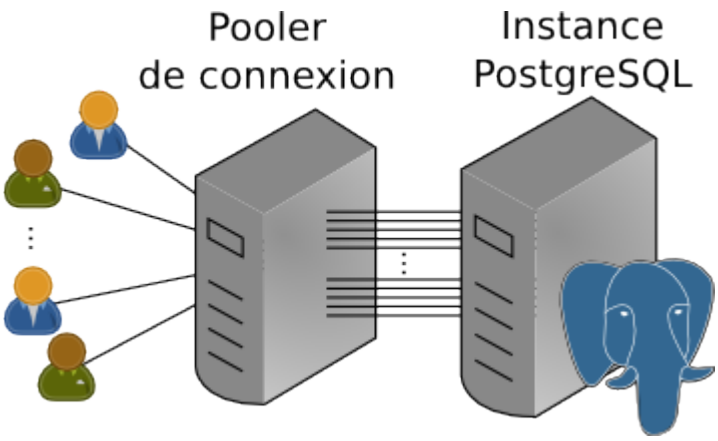
- Savoir ce qu'est un pool de connexion ?
- Avantage, inconvénients & limites
- Savoir mettre en place un pooler de connexion avec PgBouncer

4.2 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.

4.2.1 Serveur de pool de connexions



4.2.2 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 donnée
- 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 donnée. Ainsi, une connexion maintenue par le pooler ne sera attribuée à un utilisateur que si ce couple rôle/base de donnée 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 donnée 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.

4.2.3 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)
  - saturation 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 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…)

4.2.4 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.

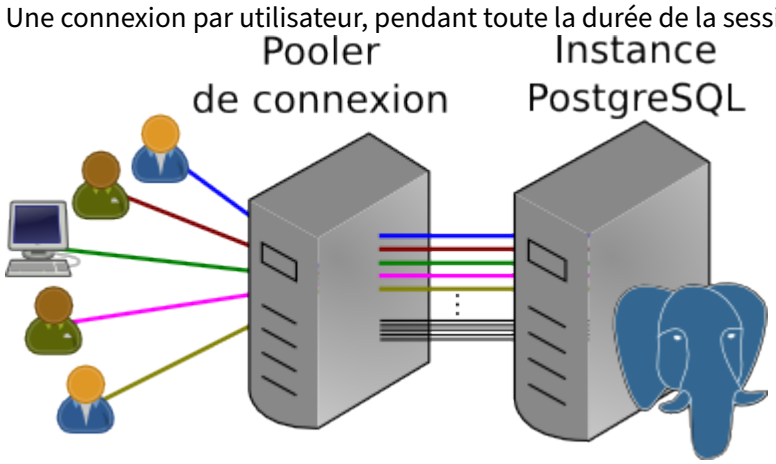
Un pooler est un élément en plus entre l'application et vos données, donc il aura un coût en performances. Il ajoute notamment une certaine latence. On n'introduit donc pas un pooler sans avoir identifié un problème. Si la configuration est bien faite, cet impact est normalement négligeable, ou en tout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

4.3 POOLING 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.

4.3.1 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 active 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.

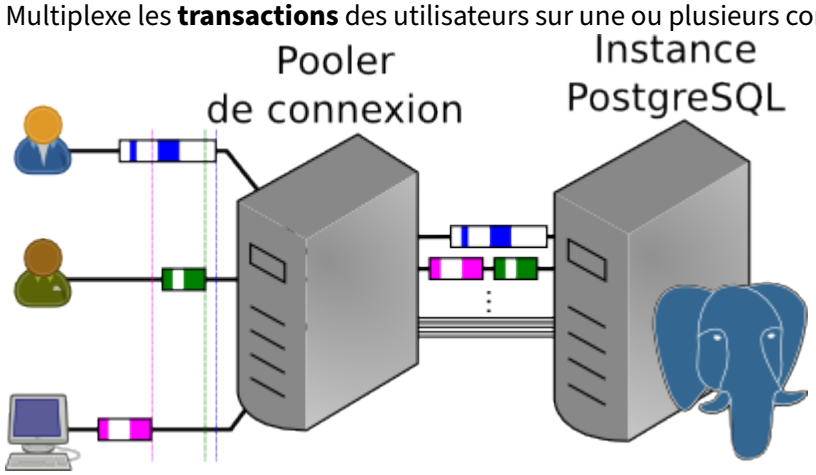
L'autre intérêt est de ne pas rejeter une connexion, même s'il n'y a plus de connexions possibles au SGBD. Contrairement au comportement de PostgreSQL, les connexions sont placées en attente si elles ne peuvent pas être satisfaites immédiatement.

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.

4.4 POOLING 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.

4.4.1 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
  - interdit les requêtes 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. Cet économie de connexions côté SGBD 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 client 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.

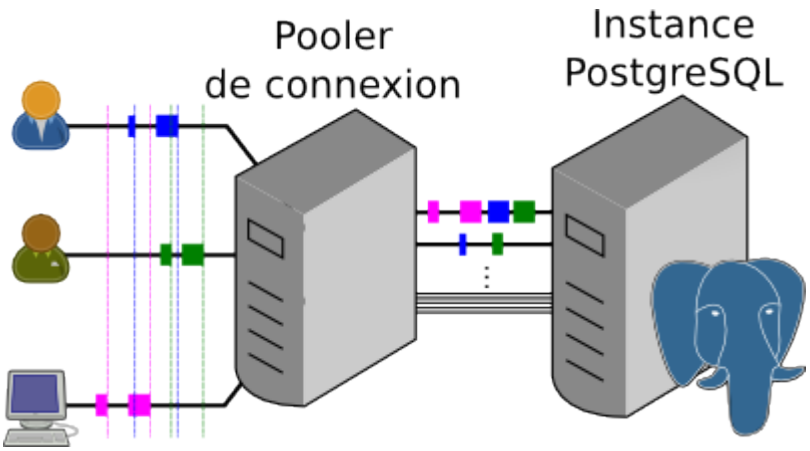
Seules les requêtes au sein d'une même transaction sont assurées d'être exécutées sur la même connexion. Ainsi, au début de cette transaction, la connexion est alors réservée exclusivement à l'utilisateur propriétaire de la transaction. Donc si le client prend son temps entre les différentes

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.

4.5 POOLING 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.

4.5.1 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.

4.6 POOLING 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 terme 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 possibles 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/

4.6.1 PgBouncer : Fonctionnalités

- Techniquement : un démon
- Disponible sous Unix & Windows
- Modes sessions / transactions / requêtes
- 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.

4.6.2 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.

4.6.3 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/

4.6.4 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 réseaux 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.

4.6.5 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).

4.6.6 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.

4.6.7 PgBouncer : Authentification par délégation

- Créer un rôle dédié
- Copier son hash de mot de passe (MD5 !) 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 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`.

Il n'y aura pas de problème avec l'authentification MD5. Par contre, le principe même de SCRAM-SHA-256 interdit de passer par un proxy. Le mot de passe de l'utilisateur dédié devra donc forcément être encodé en MD5.

Exemple de configuration :

```
SET password_encryption = 'md5' ;
CREATE ROLE frontend PASSWORD 'pass' LOGIN ;
SELECT rolpassword FROM pg_authid WHERE rolname = 'frontend' \gx
```

Le hachage obtenu (ici en MD5) est recopié dans `userlist.txt` :

```
"frontend" "md5b935ea59a93354a09864a11ff102b548"
```

Le paramètre `auth_query` définit la requête à exécuter pour ensuite comparer les résultats avec les identifiants de connexion. 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]). Il faut évidemment que l'utilisateur choisi 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.

4.6.8 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 en multiplexage se justifie notamment quand des centaines, voire 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

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

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
```

4.6.9 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`

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

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
```

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
```

4.6.10 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.

4.6.11 PgBouncer : Traces

- Fichier
  - `logfile`
- Événements 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 événements 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
```

4.6.12 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[9] pour permettre une supervision de cet outil.

L'outil d'audit pgCluu[10] 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`.

[9] https://github.com/bucardo/check_postgres
[10] https://pgcluu.darold.net

4.7 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

4.7.1 Questions

```
SELECT * FROM questions ;
```

4.8 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.

4.8.1 Pooling par session

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

4.8.2 Pooling 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.

4.8.3 Pooling par requête

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

Repasser PgBouncer en pooling par session.

4.8.4 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.

4.9 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
```

4.9.1 Pooling 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.

4.9.2 Pooling 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
```

4.9.3 Pooling 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`).

4.9.4 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
…
```

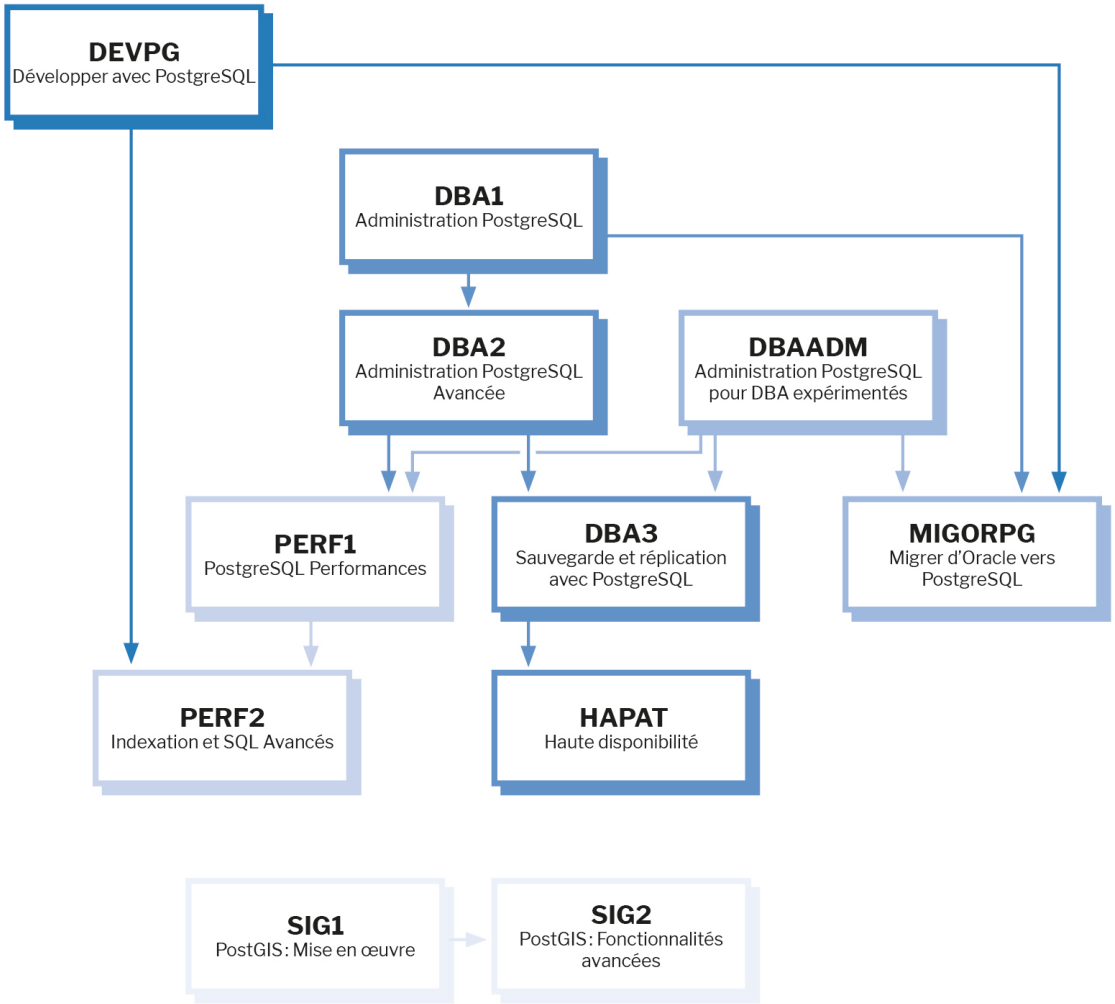
Ces tests n'ont pas pour objectif d'être représentatif mais juste de mettre en évidence le coût d'ouverture/fermeture de connexion. Dans ce cas, le pooler peut apporter un gain très significatif sur les performances.

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.