Module V2

Connexions distantes

24.04

Table des matières

Sur ce document

Formation	Module V2
Titre	Connexions distantes
Révision	24.04
PDF	https://dali.bo/v2_pdf
EPUB	https://dali.bo/v2_epub
HTML	https://dali.bo/v2_html
Slides	https://dali.bo/v2_slides
TP	https://dali.bo/v2_tp
TP (solutions)	https://dali.bo/v2_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

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

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

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

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

1/ 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` (amélioration en v16)
  - support des triggers
  - support de la parallélisation
  - support des exécutions asynchrones (v14)
  - possibilité d'importer un schéma complet

Les *Foreign Data Wrappers* sont fréquement améliorés. La dernière optimisation en date concerne la gestion de la commande `ANALYZE` en version 16 de PostgreSQL.

Jusque PostgreSQL 15, lorsque `ANALYZE` était exécuté sur une table distante, l'échantillonnage était effectué localement à l'instance. Les données étaient donc intégralement rapatriées avant que ne soient effectuées les opérations d'échantillonnage. Pour des grosses tables, cette manière de faire était tout sauf optimisée. À partir de PostgreSQL 16, l'échantillonnage des lignes se fait par défaut sur le serveur distant grâce à l'option `analyze_sampling`. La volumétrie transférée est alors bien plus basse. Le calcul des statistiques des données sur cet échantillon se fait toujours sur l'instance qui lance `ANALYZE`.

Cette option peut prendre les valeurs `off`, `auto`, `system`, `bernoulli` et `random`. La valeur par défaut est `auto` qui permettra d'utiliser soit `bernoulli` (cas général) soit `random` (pour des serveurs distants plus anciens que PostgreSQL 9.5). Par exemple :

```
ALTER FOREIGN TABLE t1_fdw OPTIONS ( analyze_sampling 'auto' );
ALTER FOREIGN TABLE t1_fdw OPTIONS ( SET analyze_sampling 'off' );
```

`analyze_sampling` peut être appliqué sur le *foreign server* ou la *foreign table* directement. sur la `FOREIGN TABLE` directement.

Il est possible de créer des triggers locaux sur des tables étrangères. Un trigger sur `TRUNCATE` n'est pas possible avant PostgreSQL 16.

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.

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

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 :

```
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 distante, 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.

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

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

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 ;
```

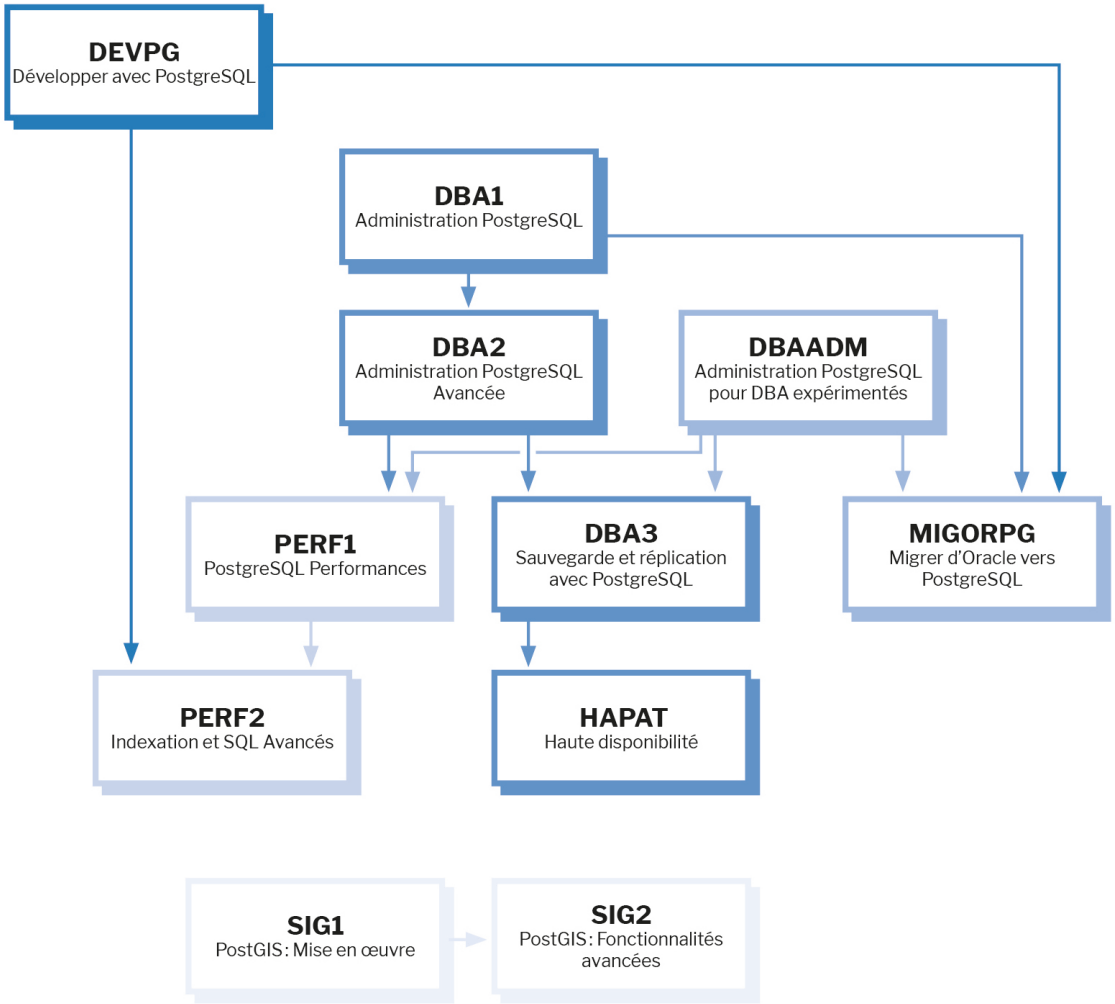
Il faut l'option `VERBOSE` pour voir la requête envoyée au serveur distant. Vous constatez que le prédicat sur `vin_id` a été transmis, ce qui est le principal avantage de cette implémentation sur les DBLinks.

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.