Module PL0

PL/pgSQL & langages PL : introduction

25.09

Table des matières

Sur ce document

Formation	Module PL0
Titre	PL/pgSQL & langages PL : introduction
Révision	25.09
PDF	https://dali.bo/pl0_pdf
EPUB	https://dali.bo/pl0_epub
HTML	https://dali.bo/pl0_html
Slides	https://dali.bo/pl0_slides

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre les diapositives, les manuels eux-mêmes et les travaux pratiques.

Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

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

Versions de PostgreSQL couvertes

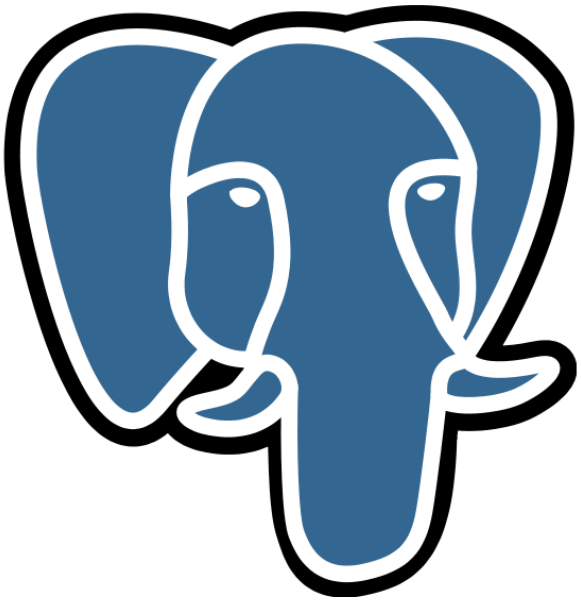
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ PL/pgSQL & langages PL : introduction



1.1 INTRODUCTION

Il n'y a pas que le SQL.

1.1.1 Au menu

- Langages PL : PL/pgSQL & les autres
- Exemples
- Intérêt

1.1.2 Objectifs

- Savoir ce qu'est un langage PL
- Savoir quand les utiliser
- Savoir coder une fonction simple

1.2 LANGAGES PL : INTRODUCTION

1.2.1 Qu'est-ce qu'un langage PL ?

- PostgreSQL accepte des fonctions et procédures en SQL et C
  - utilisables depuis une requête SQL
- PL = *Procedural Language*

PostgreSQL permet d'écrire des fonctions dans divers langages. Nativement, sont supportées les fonctions en SQL et en C.

PL est l'acronyme de *Procedural Languages*. En dehors du C et du SQL, tous les langages acceptés par PostgreSQL sont des PL.

Une fois en place, une fonction peut être appelée depuis une requête SQL, ou une autre fonction SQL ou PL/pgSQL. Le langage utilisé est totalement transparent pour l'utilisateur.

1.2.2 Quels langages PL sont disponibles ?

- Installé par défaut :
  - PL/pgSQL
- Intégrés au projet :
  - PL/Perl
  - PL/Python
  - PL/Tcl
- Extensions tierces :
  - PL/java, PL/R, PL/v8 (JavaScript), PL/R, PL/sh…
  - extensible à volonté

Les quatre langages PL supportés nativement (en plus du C et du SQL bien sûr) sont décrits en détail dans la documentation officielle :

- PL/pgSQL[1] est intégré par défaut dans toute nouvelle base (de par sa présence dans la base modèle **template1**) ;
- PL/Tcl[2] ;
- PL/Perl[3] ;
- PL/Python[4].

[1] https://docs.postgresql.fr/current/plpgsql.html
[2] https://docs.postgresql.fr/current/pltcl.html
[3] https://docs.postgresql.fr/current/plperl.html
[4] https://docs.postgresql.fr/current/plpython.html

D'autres langages PL sont accessibles en tant qu'extensions tierces. Ils réclament généralement d'installer les bibliothèques du langage sur le serveur. Souvent il suffit d'installer des paquets fournis par le PGDG.

Une liste plus large est par ailleurs disponible sur le wiki PostgreSQL[5], à des stades divers de fiabilité. Il en ressort qu'une vingtaine de langages sont disponibles, dont dix installables en production, même s'il faut vérifier que le langage reste bien maintenu. Les plus connus sont PL/Java[6] ou PL/R[7]. S'ajoutent aussi PL/Rust[8], JavaScript (PLV8[9]), PL/Haskell[10], ou n'importe quel langage shell (via PL/sh[11]).

De plus, il est possible d'en ajouter d'autres, comme décrit dans la documentation[12].

1.2.3 Langages *trusted* vs *untrusted*

- *Trusted* = langage de confiance :
  - ne permet que l'accès à la base de données
  - pas aux fichiers, réseau, etc.
  - SQL, PL/pgSQL, PL/Perl, PL/Tcl
- *Untrusted* :
  - PL/Python, C, PL/TclU, PL/PerlU
  - potentiellement dangereux !

Les langages *trusted* (« de confiance ») ne peuvent accéder qu'aux données de la base de la session en cours. Ils ne peuvent pas accéder aux autres bases, aux systèmes de fichiers, au réseau, réaliser certaines opérations comme `fork`, etc. Ils ne peuvent donc pas compromettre le système, mais de ce fait, ils sont bridés dans leurs possibilités. PL/pgSQL et SQL sont des exemples typiques.

La plupart des autres langages sont *untrusted*, n'ont pas ces limites et sont donc potentiellement dangereux. Certains sont même très dangereux par ce qu'ils permettent de faire (PL/sh). Certains langages sont disponibles en version complète *untrusted*, et en version bridée *trusted* (comme pl/Perl et pl/PerlU, ou Rust).

Seuls les superutilisateurs peuvent créer une routine dans un langage *untrusted*. Par contre, ils peuvent ensuite donner les droits d'exécution à ces routines aux autres rôles de l'instance :

```
GRANT EXECUTE ON FUNCTION nom_fonction TO un_role ;
```

[5] https://wiki.postgresql.org/wiki/PL_Matrix
[6] https://tada.github.io/pljava/
[7] https://github.com/postgres-plr/plr
[8] https://plrust.io/
[9] https://github.com/plv8/plv8
[10] https://github.com/ed-o-saurus/PLHaskell
[11] https://github.com/petere/plsh
[12] https://docs.postgresql.fr/current/plhandler.html

Rappelons qu'un superutilisateur a autant de droits sur le serveur que l'utilisateur système **postgres**. Donner à un utilisateur les droits sur une telle fonction ne se fait pas à la légère. Il faut être sûr que l'utilisateur ne peut détourner la fonction pour des buts malveillants.

1.2.4 Les langages PL de PostgreSQL

Les langages PL fournissent :
- des fonctionnalités procédurales dans un univers relationnel
- les fonctionnalités avancées du langage PL choisi
- les librairies associées
- des performances supérieures au code côté client
  - latence
  - indexation

La question se pose souvent de placer la logique applicative du côté de la base, dans un langage PL, ou des clients. La tendance, ces dernières années, est de coder intégralement la logique du côté client. Mais il peut y avoir de nombreuses raisons en faveur de la première option.

Centralisation du code :

Simplifier et centraliser des traitements clients directement dans la base est l'argument le plus fréquent. Par exemple, une insertion complexe dans plusieurs tables, avec la génération et la récupération d'identifiants pour lier entre elles des tables, peut évidemment être écrite côté client. Il est quelquefois plus pratique de l'écrire sous forme d'une routine PL. Si plusieurs applications ont potentiellement besoin d'opérer un même traitement, à fortiori dans des langages différents, porter cette logique dans la base réduit d'autant les risques de *bugs* et facilite la maintenance.

Une règle peut être que tout ce qui a trait à l'intégrité des données devrait être exécuté au niveau de la base.

Performances :

Le code s'exécute localement, directement dans le moteur de la base. Il n'y a donc pas tous les changements de contexte et échanges de messages réseaux dus à l'exécution de nombreux ordres SQL consécutifs.

L'impact de la latence due au trafic réseau entre la base au client est souvent sous-estimé !

Les langages PL permettent aussi d'accéder à leurs bibliothèques spécifiques (extrêmement nombreuses en python ou perl, entre autres).

Une fonction en PL peut également servir à l'indexation des données. Cela est impossible si elle se calcule sur une autre machine.

Simplicité :

Suivant le besoin, un langage PL peut être bien plus pratique que le langage client.

Il est par exemple très simple d'écrire un traitement d'insertion/mise à jour en PL/pgSQL, le langage étant créé pour simplifier ce genre de traitements, et la gestion des exceptions pouvant s'y produire.

PL/Perl et PL/Python sont bien plus évolués que PL/PgSQL. Par exemple, ils sont bien plus efficaces en matière de traitement de chaînes de caractères, possèdent des structures avancées comme des tables de hachage, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leur version *untrusted*, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un service web par exemple, ou d'écrire des données dans un fichier externe.

Il existe des langages PL spécialisés. Le plus emblématique d'entre eux est PL/R[13]. R est un langage utilisé par les statisticiens pour manipuler de gros jeux de données. PL/R permet donc d'effectuer ces traitements R directement en base, traitements qui seraient très pénibles à écrire dans d'autres langages, et avec une latence dans le transfert des données. Un autre intérêt est de réutiliser des fonctions existantes, que ce soit des librairies du langage ou des développements exécutés auparavant sur un poste client. PL/Python possède aussi de nombreuses librairies mathématiques et statistiques.

Un autre langage (pas PL à proprement parler) est, du moins sur le papier, plus rapide que tous les langages cités précédemment : le C[14]. Des procédures stockées en C peuvent être compilées à l'extérieur de PostgreSQL, en respectant un certain formalisme, puis chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour.

Mais attention : toute erreur dans le code C est susceptible d'accéder à toute la mémoire visible par le processus PostgreSQL qui l'exécute, et donc de corrompre les données. Il est donc conseillé de n'utiliser le C qu'en dernier recours.

La grande variété des différents langages PL supportés par PostgreSQL permet normalement d'en trouver un correspondant aux besoins et aux langages déjà maîtrisés dans l'entreprise.

[13] https://github.com/postgres-plr/plr/blob/master/userguide.md
[14] https://docs.postgresql.fr/current/xfunc-c.html

1.2.5 Intérêts de PL/pgSQL en particulier

- **Dédié au traitement des données et au SQL**
- Maniement direct des données
- Ajout de structures de contrôle au langage SQL
- Traitements complexes, transactions
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- *Trusted*
- Facile à utiliser
- Mais assez pauvre

Le langage étant assez ancien, proche du Pascal et de l'ADA, sa syntaxe ne choquera personne. Elle est d'ailleurs très proche de celle du PLSQL d'Oracle.

Les gros atouts de PL/pgSQL est son intégration au SQL, et son intégration à la base de données. PL/pgSQL permet d'écrire des requêtes SQL directement dans le code PL sans déclaration préalable, sans appel à des méthodes complexes, ni rien de cette sorte. Le code SQL est mélangé naturellement au code PL, et on a donc un sur-ensemble procédural de SQL. Les curseurs sont totalement optionnels.

À l'inverse, le gros défaut commun de tous les autres langages est qu'ils ont besoin de fonctions supplémentaires pour accéder à la base de données. L'accès aux données est donc assez fastidieux au niveau syntaxique comparé à PL/pgSQL.

PL/pgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux rajoutés par l'utilisateur. Il peut les manipuler de façon transparente. Là encore, les autres langages peuvent poser problème à cause de différences de types.

Un autre atout du PL/pgSQL (et du C et du SQL) est l'absence d'interpréteur externe à lancer, au contraire de python, Java, etc.

PL/pgSQL possède les structures de contrôle habituelles comme `IF… THEN… END IF`, ou les boucles `LOOP…END LOOP`, y compris les boucles directement sur un ensemble de résultat.

PL/pgSQL dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

Les procédures PL/pgSQL (pas les fonctions) permettent de gérer des transactions.

PL/pgSQL est *trusted*. Par défaut, tous les utilisateurs peuvent donc créer des routines dans ce langage. Vous pouvez toujours soit supprimer le langage dans la base, soit retirer les droits à un utilisateur sur ce langage (via la commande SQL `REVOKE`).

PL/pgSQL est donc raisonnablement facile à utiliser : il y a peu de complications et peu de pièges. Par contre, le langage peut sembler pauvre pour des gens habitués aux langages plus récents. Il n'y a pas non plus beaucoup de librairies de fonctions à utiliser. Les autres langages et les diverses extensions comblent cependant ces lacunes.

1.2.6 Routines / Procédures stockées / Fonctions

- **Procédure** stockée
  - pas de retour
  - contrôle transactionnel : `COMMIT` / `ROLLBACK`
- **Fonction**
  - peut renvoyer des données (même des lignes)
  - utilisable dans un `SELECT`
  - peut être de type `TRIGGER`, agrégat, fenêtrage
- **Routine**
  - procédure ou fonction

Les programmes écrits à l'aide des langages PL sont habituellement enregistrés sous forme de « routines » :

- procédures ;
- fonctions ;
- fonctions *trigger* ;
- fonctions d'agrégat ;
- fonctions de fenêtrage (*window functions*).

Le code source de ces objets est stocké dans la table `pg_proc` du catalogue.

Les procédures, apparues avec PostgreSQL 11, sont très similaires aux fonctions. Les principales différences entre les deux sont :

- Les fonctions doivent déclarer des arguments en sortie (`RETURNS` ou arguments `OUT`). Elles peuvent renvoyer n'importe quel type de donnée, ou des ensembles de lignes. Il est possible d'utiliser `void` pour une fonction sans argument de sortie ; c'était d'ailleurs la méthode utilisée pour émuler le comportement d'une procédure avant leur introduction avec PostgreSQL 11. Les procédures n'ont pas de code retour (on peut cependant utiliser des paramètres `OUT` ou `INOUT`).
- Les procédures offrent le support du contrôle transactionnel, c'est-à-dire la capacité de valider (`COMMIT`) ou annuler (`ROLLBACK`) les modifications effectuées jusqu'à ce point par la procédure. L'intégralité d'une fonction s'effectue dans la transaction appelante.
- Les procédures sont appelées exclusivement par la commande SQL `CALL` ; les fonctions peuvent être appelées dans la plupart des ordres DML/DQL (notamment `SELECT`), mais pas par `CALL`.
- Les fonctions peuvent être déclarées de telle manière qu'elles peuvent être utilisées dans des rôles spécifiques (*trigger*, agrégat ou fonction de fenêtrage).

1.3 EXEMPLES DE FONCTION SQL

Les fonctions en pur SQL sont généralement très simples, mais peuvent avoir un intérêt en performance (pour les plus simples, l'optimiseur « voit » leur contenu).

Les exemples suivants ont pour unique ambition de montrer ce qui est possible, sans détailler pour le moment.

1.3.1 Fonction SQL renvoyant un calcul

Déclaration :

```
CREATE FUNCTION employe_eligible_prime_sql (service int, date_embauche
 ↪  date)
RETURNS boolean
LANGUAGE sql
AS $$
  SELECT ( service !=3 AND date_embauche < '2003-01-01')  ;
$$ ;
```

Utilisation :

```
SELECT matricule, num_service, nom, prenom
FROM    employes_big
WHERE   employe_eligible_prime_sql (num_service, date_embauche) = true ;
```

1.3.2 Fonction SQL renvoyant plusieurs champs

```
CREATE OR REPLACE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer)
LANGUAGE sql
AS $$
  SELECT  extract (day FROM d)::int,
                  extract(month FROM d)::int,
                  extract (year FROM d)::int ;
$$ ;

SELECT * FROM explose_date ('31-12-2020');

 jour | mois | annee
------+------+-------
   31 |    0 |  2020
```

Une fonction SQL peut renvoyer plusieurs champs. Il existe plusieurs variantes de cette syntaxe.

1.3.3 Fonction SQL renvoyant plusieurs lignes d'une table

```
CREATE OR REPLACE FUNCTION  tables_jamais_analyzees ()
RETURNS SETOF pg_stat_user_tables
LANGUAGE sql
AS $$
  SELECT * FROM pg_stat_user_tables
  WHERE coalesce(last_analyze, last_autoanalyze) IS NULL ;
$$ ;
```

Une fonction SQL peut renvoyer plusieurs lignes, et lire une table. Par exemple, celle-ci va chercher dans une vue système les tables dont les statistiques n'ont jamais été analysées. Dans ce cas précis, on a l'équivalent d'une vue.

1.4 EXEMPLES DE ROUTINES PL/PGSQL

1.4.1 Exemple de procédure PL/pgSQL avec gestion transactionnelle

```
CREATE OR REPLACE PROCEDURE vide_tables (dry_run BOOLEAN) AS $$
BEGIN
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
    IF dry_run THEN
        ROLLBACK ;
    END IF ;
EXCEPTION WHEN undefined_table THEN
    RAISE NOTICE 'Table inexistante [%] %', SQLSTATE, SQLERRM;
    ROLLBACK ;
END ; $$ LANGUAGE plpgsql ;

CALL vide_tables (dry_run=>true);
```

Cette procédure exécute des ordres et annule la transaction si le paramètres `dry_run` est à `true`.

Au cas où une table n'existe pas, le bloc `EXCEPTION` intercepte l'erreur, affiche un message, annule et sort sans erreur. Une autre erreur ne serait pas interceptée mais transmise à l'appelant.

1.4.2 Exemple de bloc anonyme avec variable, boucle sur une table, et SQL dynamique

```
DO $$
DECLARE r record ;
        n int := 0 ;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        n := n+1 ;
        RAISE NOTICE '% - Analyze de : %.%', n, r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END $$ ;
```

Les blocs anonymes définis avec `DO` permettent d'exécuter du code PL/pgSQL, intégralement sur le

serveur, sans définir de fonction ou procédure.

Ce bloc utilise une boucle sur une vue système pour sélectionner des tables dont les statistiques n'ont jamais été analysées, puis génère pour chacune un ordre `ANALYZE`, qui sera ensuite exécuté dans la session grâce à `EXECUTE`. Deux variables sont utilisées : un enregistrement (*record*) qui contient la ligne en cours dans la boucle `FOR`, et un entier comme compteur.

1.4.3 Exemple de trigger

```
CREATE OR REPLACE FUNCTION horodatage() RETURNS trigger
AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    NEW.date_ajout := now();
  ELSEIF TG_OP = 'UPDATE' THEN
    NEW.date_modif := now();
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER trig_horodatage
   BEFORE INSERT OR UPDATE ON ma_table
   FOR EACH ROW
   EXECUTE PROCEDURE horodatage();
```

Un trigger est une fonction qui se déclenche sur une insertion, suppression, modification d'une table. L'exemple ci-dessus est une très classique mise à jour d'un champ à chaque insertion ou mise à jour. Les triggers servent aussi à archiver des lignes, tracer des accès… Il ne faut pas en abuser pour des raisons de lisibilité et performances.

1.5 CONCLUSION

- Des langages souvent négligés malgré leurs atouts
- Pour aller plus loin :
  - *PL/pgSQL : Les bases* : https://dali.bo/p1_html
  - *PL/pgSQL avancé* : https://dali.bo/p2_html

Les différents langages PL sont assez peu connus, alors qu'ils peuvent être extrêmement utiles. Il faut savoir dans quelles circonstances le code peut être exécuté côté serveur.

Pour aller plus loin, voir nos autres modules de formation.

1.5.1 Questions

N'hésitez pas, c'est le moment !

1.6 QUIZ

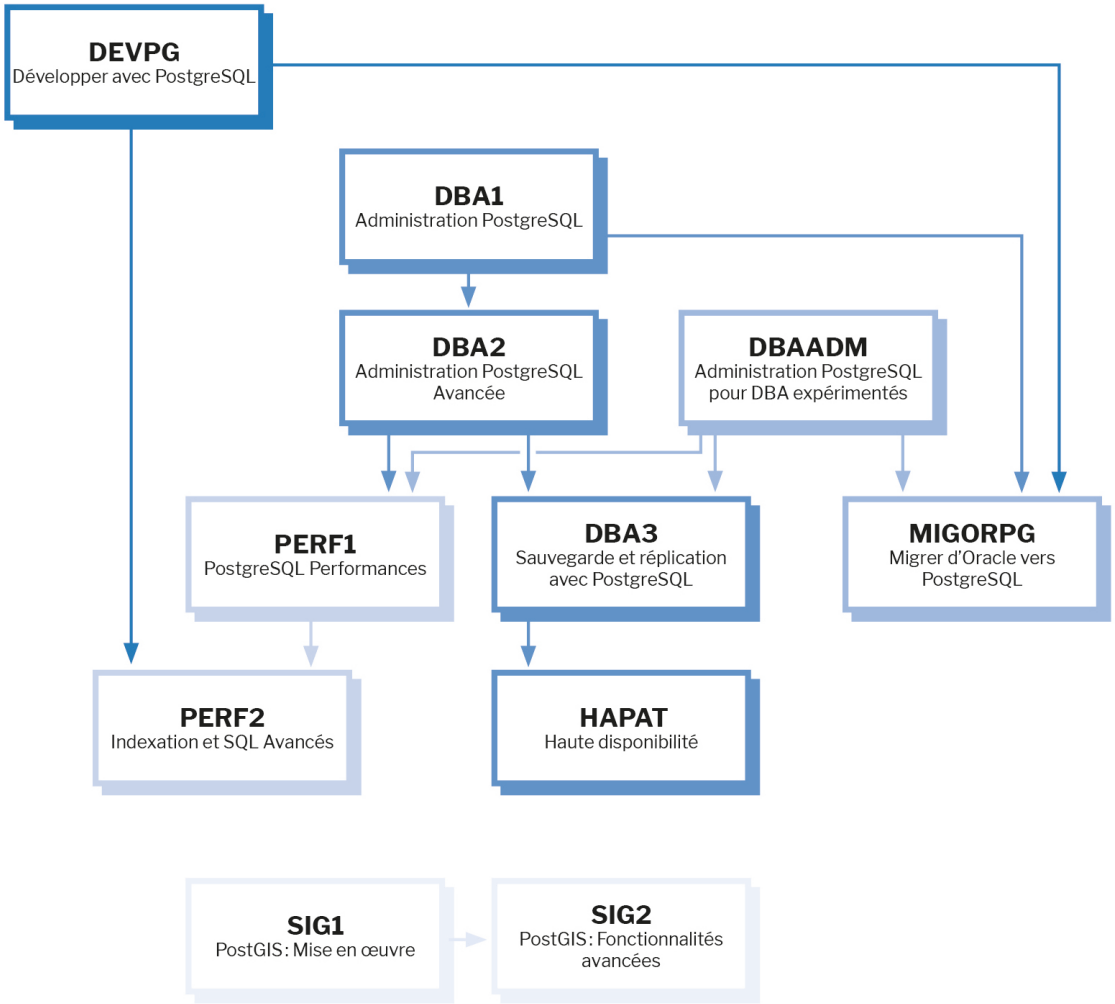
https://dali.bo/pl0_quiz

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
- DEV1 : Introduction à SQL
  https://dali.bo/dev1
- 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.