Module A2

Découverte des fonctionnalités

24.04

Table des matières

Sur ce document

Formation	Module A2
Titre	Découverte des fonctionnalités
Révision	24.04
PDF	https://dali.bo/a2_pdf
EPUB	https://dali.bo/a2_epub
HTML	https://dali.bo/a2_html
Slides	https://dali.bo/a2_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 :

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

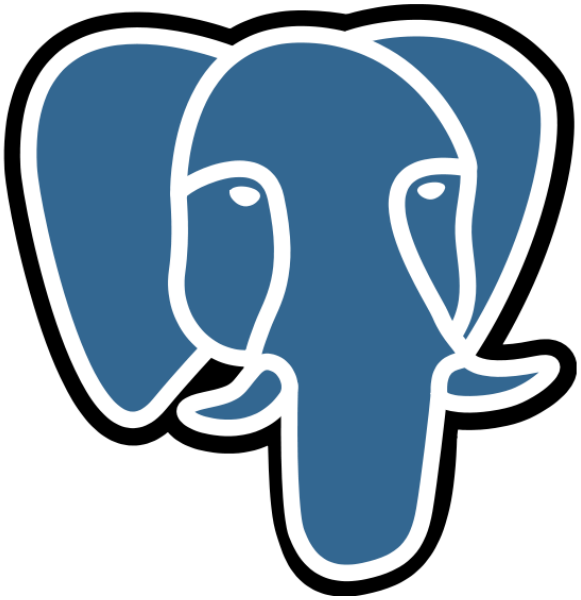
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Découverte des fonctionnalités



1.1 AU MENU

- Fonctionnalités du moteur
- Objets SQL
- Connaître les différentes fonctionnalités et possibilités
- Découvrir des exemples concrets

Ce module propose un tour rapide des fonctionnalités principales du moteur : ACID, MVCC, transactions, journaux de transactions… ainsi que des objets SQL gérés (schémas, index, tablespaces, triggers…). Ce rappel des concepts de base permet d'avancer plus facilement lors des modules suivants.

1.2 FONCTIONNALITÉS DU MOTEUR

- Standard SQL
- ACID : la gestion transactionnelle
- Niveaux d'isolation
- Journaux de transactions
- Administration
- Sauvegardes
- Réplication
- Supervision
- Sécurité
- Extensibilité

Cette partie couvre les différentes fonctionnalités d'un moteur de bases de données. Il ne s'agit pas d'aller dans le détail de chacune, mais de donner une idée de ce qui est disponible. Les modules suivants de cette formation et des autres formations détaillent certaines de ces fonctionnalités.

1.2.1 Respect du standard SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, routines, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.

La dernière version du standard SQL est SQL:2023[1]. À ce jour, aucun SGBD ne la supporte complètement, *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de la norme est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

[1] https://en.wikipedia.org/wiki/SQL:2023

1.2.2 ACID

Gestion transactionnelle : la force des bases de données relationnelles :

- **Atomicité** (*Atomic*)
- **Cohérence** (*Consistent*)
- **Isolation** (*Isolated*)
- **Durabilité** (*Durable*)

Les propriétés ACID sont le fondement même de toute bonne base de données. Il s'agit de l'acronyme des quatre règles que toute transaction (c'est-à-dire une suite d'ordres modifiant les données) doit respecter :

- **A** : Une transaction est appliquée en « tout ou rien ».
- **C** : Une transaction amène la base d'un état stable à un autre.
- **I** : Les transactions n'agissent pas les unes sur les autres.
- **D** : Une transaction validée sera conservée de manière permanente.

Les bases de données relationnelles les plus courantes depuis des décennies (PostgreSQL bien sûr, mais aussi Oracle, MySQL, SQL Server, SQLite…) se basent sur ces principes, même si elles font chacune des compromis différents suivant leurs cas d'usage, les compromis acceptés à chaque époque avec la performance et les versions.

Atomicité :

Une transaction doit être exécutée entièrement ou pas du tout, et surtout pas partiellement, même si elle est longue et complexe, même en cas d'incident majeur sur la base de données. L'exemple basique est une transaction bancaire : le montant d'un virement doit être sur un compte ou un autre, et en cas de problème ne pas disparaître ou apparaître en double. Ce principe garantit que les données modifiées par des transactions valides seront toujours visibles dans un état stable, et évite nombre de problèmes fonctionnels comme techniques.

Cohérence :

Un état cohérent respecte les règles de validité définies dans le modèle, c'est-à-dire les contraintes définies dans le modèle : types, plages de valeurs admissibles, unicité, liens entre tables (clés étrangères), etc. Le non-respect de ces règles par l'applicatif entraîne une erreur et un rejet de la transaction.

Isolation :

Des transactions simultanées doivent agir comme si elles étaient seules sur la base. Surtout, elles ne voient pas les données *non validées* des autres transactions. Ainsi une transaction peut travailler sur un état stable et fixe, et durer assez longtemps sans risque de gêner les autres transactions.

Il existe plusieurs « niveaux d'isolation » pour définir précisément le comportement en cas de lectures ou écritures simultanées sur les mêmes données et pour arbitrer avec les contraintes de per-

formances ; le niveau le plus contraignant exige que tout se passe comme si toutes les transactions se déroulaient successivement.

Durabilité :

Une fois une transaction validée par le serveur (typiquement : `COMMIT` ne retourne pas d'erreur, ce qui valide la cohérence et l'enregistrement physique), l'utilisateur doit avoir la garantie que la donnée ne sera pas perdue ; du moins jusqu'à ce qu'il décide de la modifier à nouveau. Cette garantie doit valoir même en cas d'événement catastrophique : plantage de la base, perte d'un disque… C'est donc au serveur de s'assurer autant que possible que les différents éléments (disque, système d'exploitation…) ont bien rempli leur office. C'est à l'humain d'arbitrer entre le niveau de criticité requis et les contraintes de performances et de ressources adéquates (et fiables) à fournir à la base de données.

NoSQL :

À l'inverse, les outils de la mouvance (« NoSQL », par exemple MongoDB ou Cassandra), ne fournissent pas les garanties ACID. C'est le cas de la plupart des bases non-relationnelles, qui reprennent le modèle BASE[2] (*Basically Available*, *Soft State*, *Eventually Consistent*, soit succintement : disponibilité d'abord ; incohérence possible entre les réplicas ; cohérence… à terme, après un délai). Un intérêt est de débarasser le développeur de certaines lourdeurs apparentes liées à la modélisation assez stricte d'une base de données relationnelle. Cependant, la plupart des applications ont d'abord besoin des garanties de sécurité et cohérence qu'offrent un moteur transactionnel classique, et la décision d'utiliser un système ne les garantissant pas ne doit pas être prise à la légère ; sans parler d'autres critères comme la fragmentation du domaine par rapport au monde relationnel et son SQL (à peu près) standardisé. Avec le temps, les moteurs transactionnels ont acquis des fonctionnalités qui faisaient l'intérêt des bases NoSQL (en premier lieu la facilité de réplication et le stockage de JSON), et ces dernières ont tenté d'intégrer un peu plus de sécurité dans leur modèle.

1.2.3 MVCC

- MultiVersion Concurrency Control
- Le « noyau » de PostgreSQL
- Garantit les propriétés ACID
- Permet les accès concurrents sur la même table
  - une lecture ne bloque pas une écriture
  - une écriture ne bloque pas une lecture
  - une écriture ne bloque pas les autres écritures…
  - …sauf pour la mise à jour de la **même ligne**

[2] https://en.wikipedia.org/wiki/Eventual_consistency

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

MVCC maintient toutes les versions nécessaires de chaque ligne, ainsi **chaque transaction voit une image figée de la base** (appelée *snapshot*). Cette image correspond à l'état de la base lors du démarrage de la requête ou de la transaction, suivant le niveau d'*isolation* demandé par l'utilisateur à PostgreSQL pour la transaction.

MVCC fluidifie les mises à jour en évitant les blocages trop contraignants (verrous sur `UPDATE`) entre sessions et par conséquent de meilleures performances en contexte transactionnel.

C'est notamment MVCC qui permet d'exporter facilement une base *à chaud* et d'obtenir un export cohérent alors même que plusieurs utilisateurs sont potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs, les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, moins de contention et un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

1.2.4 Transactions

- Une transaction = ensemble **atomique** d'opérations
- « Tout ou rien »
- `BEGIN` obligatoire pour grouper des modifications
- `COMMIT` pour valider
  - y compris le DDL
- Perte des modifications si :
  - `ROLLBACK` / perte de la connexion / arrêt (brutal ou non) du serveur
- `SAVEPOINT` pour sauvegarde des modifications d'une transaction à un instant `t`
- Pas de transactions imbriquées

L'exemple habituel et très connu des transactions est celui du virement d'une somme d'argent du compte de Bob vers le compte d'Alice. Le total du compte de Bob ne doit pas montrer qu'il a été débité de X euros tant que le compte d'Alice n'a pas été crédité de X euros. Nous souhaitons en fait que les deux opérations apparaissent aux yeux du reste du système comme une seule opération unitaire. D'où l'emploi d'une transaction explicite. En voici un exemple :

```
BEGIN;
UPDATE comptes SET solde=solde-200 WHERE proprietaire='Bob';
UPDATE comptes SET solde=solde+200 WHERE proprietaire='Alice';
COMMIT;
```

Contrairement à d'autres moteurs de bases de données, PostgreSQL accepte aussi les instructions DDL dans une transaction. En voici un exemple :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);

SELECT age FROM capitaines;

 age
  35

ROLLBACK;
SELECT age FROM capitaines;

ERROR:  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines;
                        ^
```

Nous voyons que la table `capitaines` a existé **à l'intérieur** de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final.

Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL : PostgreSQL n'effectue aucun `COMMIT` implicite sur des ordres DDL tels que `CREATE TABLE`, `DROP TABLE` ou `TRUNCATE TABLE`. De ce fait, ces ordres peuvent être annulés au sein d'une transaction.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);
SAVEPOINT insert_sp;
UPDATE capitaines SET age = 45 WHERE nom = 'Haddock';
ROLLBACK TO SAVEPOINT insert_sp;
COMMIT;

SELECT age FROM capitaines WHERE nom = 'Haddock';

 age
  35
```

Malgré le `COMMIT` après l'`UPDATE`, la mise à jour n'est pas prise en compte. En effet, le `ROLLBACK TO SAVEPOINT` a permis d'annuler cet `UPDATE` mais pas les opérations précédant le `SAVEPOINT`.

À partir de la version 12, il est possible de chaîner les transactions avec `COMMIT AND CHAIN` ou `ROLLBACK AND CHAIN`. Cela veut dire terminer une transaction et en démarrer une autre immédiatement après avec les mêmes propriétés (par exemple, le niveau d'isolation).

1.2.5 Niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Nous pouvons spécifier le niveau d'isolation au démarrage d'une transaction
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited` (défaut)
  - `repeatable read`
  - `serializable`

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

Le standard SQL spécifie quatre niveaux, mais PostgreSQL n'en supporte que trois (il n'y a pas de `read uncommitted` : les lignes non encore committées par les autres transactions sont toujours invisibles).

1.2.6 Fiabilité : journaux de transactions

- *Write Ahead Logs* (WAL)
- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Avantages :
  - WAL : écriture séquentielle
  - un seul *sync* sur le WAL
  - fichiers de données : en asynchrone
  - sauvegarde PITR et de la réplication fiables

Les journaux de transactions (appelés souvent WAL, autrefois XLOG) sont une garantie contre les pertes de données.

Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions. Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait en deux temps :

- écriture immédiate dans le journal de transactions ;
- écriture dans le fichier de données, plus tard, lors du prochain *checkpoint*.

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- PostgreSQL vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent.

Plus d'informations, lire cet article[3].

Les écritures dans le journal se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture (sur un disque dur classique, c'est l'opération la plus coûteuse).

De plus, comme nous n'écrivons que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, si le système de fichiers le supporte.

L'écriture définitive dans les fichiers de données, asynchrone et généralement de manière lissée, permet là aussi de gagner du temps.

Mais les performances ne sont pas la seule raison des journaux de transactions. Ces journaux ont aussi permis l'apparition de nouvelles fonctionnalités très intéressantes, comme le PITR et la réplication physique, basés sur le rejeu des informations stockées dans ces journaux.

1.2.7 Sauvegardes

- Sauvegarde des fichiers à froid
  - outils système
- Import/Export logique
  - `pg_dump`, `pg_dumpall`, `pg_restore`
- Sauvegarde physique à chaud
  - `pg_basebackup`
  - sauvegarde PITR

PostgreSQL supporte différentes solutions pour la sauvegarde.

La plus simple revient à sauvegarder à froid tous les fichiers des différents répertoires de données mais cela nécessite d'arrêter le serveur, ce qui occasionne une mise hors production plus ou moins longue, suivant la volumétrie à sauvegarder.

L'export logique se fait avec le serveur démarré. Plusieurs outils sont proposés : `pg_dump` pour sauvegarder une base, `pg_dumpall` pour sauvegarder toutes les bases. Suivant le format de l'export, l'import se fera avec les outils `psql` ou `pg_restore`. Les sauvegardes se font à chaud et sont cohérentes sans blocage de l'activité (seuls la suppression des tables et le changement de leur définition sont interdits).

Enfin, il est possible de sauvegarder les fichiers à chaud. Cela nécessite de mettre en place l'archivage des journaux de transactions. L'outil `pg_basebackup` est conseillé pour ce type de sauvegarde.

Il est à noter qu'il existe un grand nombre d'outils développés par la communauté pour faciliter encore plus la gestion des sauvegardes avec des fonctionnalités avancées comme le PITR (*Point In Time Recovery*) ou la gestion de la rétention, notamment pg_back (sauvegarde logique), pgBackRest ou barman (sauvegarde physique).

1.2.8 Réplication

- Réplication physique
  - instance complète
  - même architecture
- Réplication logique (PG 10+)
  - table par table / colonne par colonne avec ou sans filtre (PG 15)
  - voire opération par opération
- Asynchrones ou synchrone
- Asymétriques

PostgreSQL dispose de la réplication depuis de nombreuses années.

Le premier type de réplication intégrée est la réplication physique. Il n'y a pas de granularité, c'est forcément l'instance complète (toutes les bases de données), et au niveau des fichiers de données. Cette réplication est asymétrique : un seul serveur primaire effectue lectures comme écritures, et les serveurs secondaires n'acceptent que des lectures.

Le deuxième type de réplication est bien plus récent vu qu'il a été ajouté en version 10. Il s'agit d'une réplication logique, où les données elles-mêmes sont répliquées. Cette réplication est elle aussi asymétrique. Cependant, ceci se configure table par table (et non pas au niveau de l'instance comme pour la réplication physique). Avec la version 15, il devient possible de choisir quelles colonnes sont publiées et de filtre les lignes à publier.

La réplication logique n'est pas intéressante quand nous voulons un serveur sur lequel basculer en cas de problème sur le primaire. Dans ce cas, il vaut mieux utiliser la réplication physique. Par contre, c'est le bon type de réplication pour une réplication partielle ou pour une mise à jour de version majeure.

Dans les deux cas, les modifications sont transmises en asynchrone (avec un délai possible). Il est cependant possible de la configurer en synchrone pour tous les serveurs ou seulement certains.

1.2.9 Extensibilité

- Extensions
  - `CREATE EXTENSION monextension ;`
  - nombreuses : contrib, packagées… selon provenance
  - notion de confiance (v13+)
  - dont langages de procédures stockées !
- Système des *hooks*
- *Background workers*

Faute de pouvoir intégrer toutes les fonctionnalités demandées dans PostgreSQL, ses développeurs se sont attachés à permettre à l'utilisateur d'étendre lui-même les fonctionnalités sans avoir à modifier le code principal.

Ils ont donc ajouté la possibilité de créer des extensions. Une extension contient un ensemble de types de données, de fonctions, d'opérateurs, etc. en un seul objet logique. Il suffit de créer ou de supprimer cet objet logique pour intégrer ou supprimer tous les objets qu'il contient. Cela facilite grandement l'installation et la désinstallation de nombreux objets. Les extensions peuvent être codées en différents langages, généralement en C ou en PL/SQL. Elles ont eu un grand succès.

La possibilité de développer des routines dans différents langages en est un exemple : perl, python, PHP, Ruby ou JavaScript sont disponibles. PL/pgSQL est lui-même une extension à proprement parler, toujours présente.

Autre exemple : la possibilité d'ajouter des types de données, des routines et des opérateurs a permis l'émergence de la couche spatiale de PostgreSQL (appelée PostGIS).

Les provenances, rôle et niveau de finition des extensions sont très variables. Certaines sont des utilitaires éprouvés fournis avec PostgreSQL (parmi les « contrib »). D'autres sont des utilitaires aussi complexes que PostGIS ou un langage de procédures stockées. Des éditeurs diffusent leur produit comme une extension plutôt que *forker* PostgreSQL (Citus, timescaledb…). Beaucoup d'extensions peuvent être installées très simplement depuis des paquets disponibles dans les dépôts habituels (de la distribution ou du PGDG), ou le site du concepteur. Certaines sont diffusées comme code source à compiler. Comme tout logiciel, il faut faire attention à en vérifier la source, la qualité, la réputation et la pérennité.

Une fois les binaires de l'extension en place sur le serveur, l'ordre `CREATE EXTENSION` suffit généralement dans la base cible, et les fonctionnalités sont immédiatement exploitables.

Les extensions sont habituellement installées par un administrateur (un utilisateur doté de l'attribut `SUPERUSER`). À partir de la version 13, certaines extensions sont déclarées de confiance `trusted`). Ces extensions peuvent être installées par un utilisateur standard (à condition qu'il dispose des droits de création dans la base et le ou les schémas concernés).

Les développeurs de PostgreSQL ont aussi ajouté des *hooks* pour accrocher du code à exécuter sur certains cas. Cela a permis entre autres de créer l'extension `pg_stat_statements` qui s'accroche au code de l'exécuteur de requêtes pour savoir quelles sont les requêtes exécutées et pour récupérer des statistiques sur ces requêtes.

Enfin, les *background workers* ont vu le jour. Ce sont des processus spécifiques lancés par le serveur PostgreSQL lors de son démarrage et stoppés lors de son arrêt. Cela a permis la création de PoWA (outil qui historise les statistiques sur les requêtes) et une amélioration très intéressante de `pg_prewarm` (sauvegarde du contenu du cache disque à l'arrêt de PostgreSQL, restauration du contenu au démarrage).

Des exemples d'extensions sont décrites dans nos modules Extensions PostgreSQL pour l'utilisateur[4], Extensions PostgreSQL pour la performance[5], Extensions PostgreSQL pour les DBA[6].

1.2.10 Sécurité

- Fichier `pg_hba.conf`
- Filtrage IP
- Authentification interne (MD5, SCRAM-SHA-256)
- Authentification externe (identd, LDAP, Kerberos…)
- Support natif de SSL

Le filtrage des connexions se paramètre dans le fichier de configuration `pg_hba.conf`. Nous pouvons y définir quels utilisateurs (déclarés auprès de PostgreSQL) peuvent se connecter à quelles bases, et depuis quelles adresses IP.

L'authentification peut se baser sur des mots de passe chiffrés propres à PostgreSQL (`md5` ou le plus récent et plus sécurisé `scram-sha-256` en version 10), ou se baser sur une méthode externe (auprès de l'OS, ou notamment LDAP ou Kerberos qui couvre aussi Active Directory).

L'authentification et le chiffrement de la connexion par SSL sont couverts.

[4] https://dali.bo/x1_html
[5] https://dali.bo/x2_html
[6] https://dali.bo/x3_html

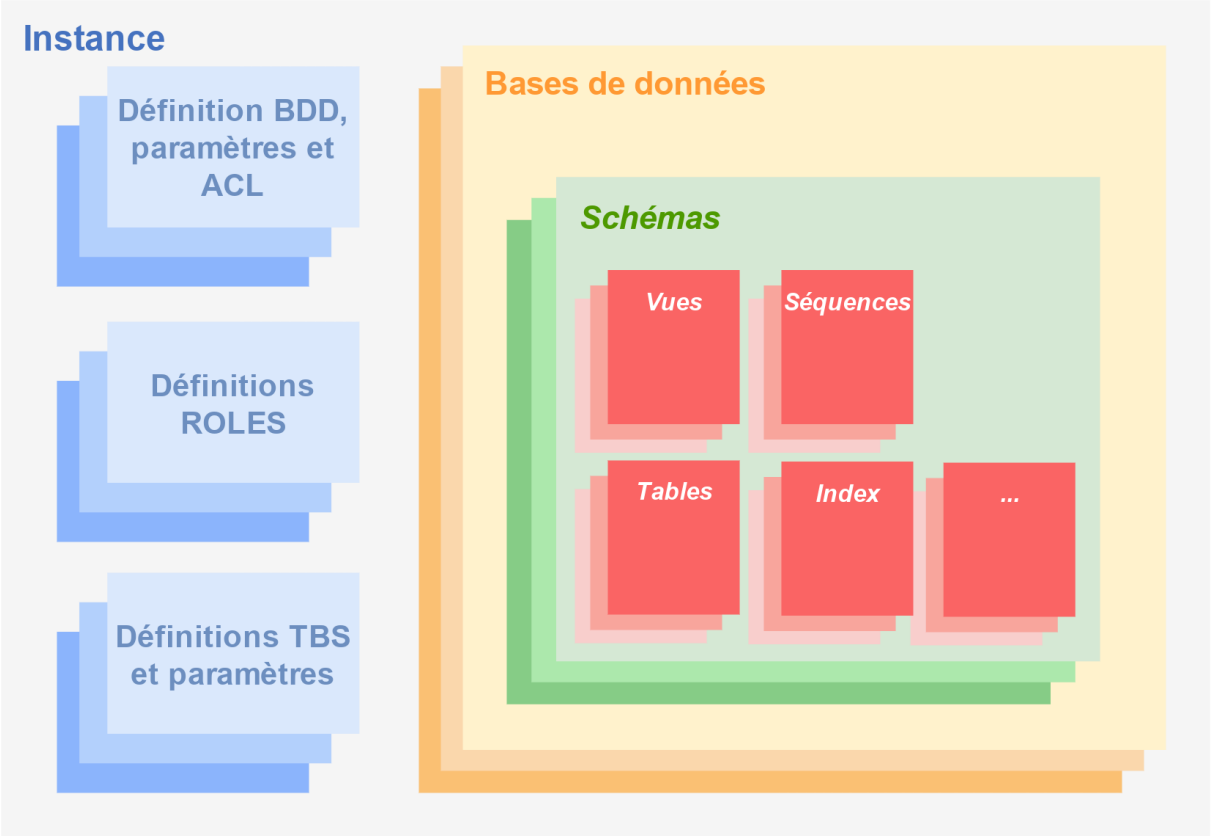
1.3 OBJETS SQL

- Instances
- Objets globaux :
  - Bases
  - Rôles
  - Tablespaces
- Objets locaux :
  - Schémas
  - Tables
  - Vues
  - Index
  - Routines
  - …

Le but de cette partie est de passer en revue les différents objets logiques maniés par un moteur de bases de données PostgreSQL.

Nous allons donc aborder la notion d'instance, les différents objets globaux et les objets locaux. Tous ne seront pas vus, mais le but est de donner une idée globale des objets et des fonctionnalités de PostgreSQL.

1.3.1 Organisation logique



Il est déjà important de bien comprendre une distinction entre les objets. Une instance est un ensemble de bases de données, de rôles et de tablespaces. Ces objets sont appelés des objets globaux parce qu'ils sont disponibles quelque soit la base de données de connexion. Chaque base de données contient ensuite des objets qui lui sont propres. Ils sont spécifiques à cette base de données et accessibles uniquement lorsque l'utilisateur est connecté à la base qui les contient. Il est donc possible de voir les bases comme des conteneurs hermétiques en dehors des objets globaux.

1.3.2 Instances

- Une instance
  - un répertoire de données
  - un port TCP
  - une configuration
  - plusieurs bases de données
- Plusieurs instances possibles sur un serveur

Une instance est un ensemble de bases de données. Après avoir installé PostgreSQL, il est nécessaire de créer un répertoire de données contenant un certain nombre de répertoires et de fichiers qui permettront à PostgreSQL de fonctionner de façon fiable. Le contenu de ce répertoire est créé initialement par la commande `initdb`. Ce répertoire stocke ensuite tous les objets des bases de données de l'instance, ainsi que leur contenu.

Chaque instance a sa propre configuration. Il n'est possible de lancer qu'un seul `postmaster` par instance, et ce dernier acceptera les connexions à partir d'un port TCP spécifique.

Il est possible d'avoir plusieurs instances sur le même serveur, physique ou virtuel. Dans ce cas, chaque instance aura son répertoire de données dédié et son port TCP dédié. Ceci est particulièrement utile quand l'on souhaite disposer de plusieurs versions de PostgreSQL sur le même serveur (par exemple pour tester une application sur ces différentes versions).

1.3.3 Rôles

- Utilisateurs / Groupes
  - Utilisateur : Permet de se connecter
- Différents attributs et droits

Une instance contient un ensemble de rôles. Certains sont prédéfinis et permettent de disposer de droits particuliers (lecture de fichier avec `pg_read_server_files`, annulation d'une requête avec `pg_signal_backend`, etc). Cependant, la majorité est composée de rôles créés pour permettre la connexion des utilisateurs.

Chaque rôle créé peut être utilisé pour se connecter à n'importe quelle base de l'instance, à condition que ce rôle en ait le droit. Ceci se gère directement avec l'attribution du droit `LOGIN` au rôle, et avec

la configuration du fichier d'accès `pg_hba.conf`.

Chaque rôle peut être propriétaire d'objets, auquel cas il a tous les droits sur ces objets. Pour les objets dont il n'est pas propriétaire, il peut se voir donner des droits, en lecture, écriture, exécution, etc par le propriétaire.

Nous parlons aussi d'utilisateurs et de groupes. Un utilisateur est un rôle qui a la possibilité de se connecter aux bases alors qu'un groupe ne le peut pas. Un groupe sert principalement à gérer plus simplement les droits d'accès aux objets.

1.3.4 Tablespaces

- Répertoire physique contenant les fichiers de données de l'instance
- Une base peut
  - se trouver sur un seul tablespace
  - être répartie sur plusieurs tablespaces
- Permet de gérer l'espace disque et les performances
- Pas de quota

Toutes les données des tables, vues matérialisées et index sont stockées dans le répertoire de données principal. Cependant, il est possible de stocker des données ailleurs que dans ce répertoire. Il faut pour cela créer un tablespace. Un tablespace est tout simplement la déclaration d'un autre répertoire de données utilisable par PostgreSQL pour y stocker des données :

```
CREATE TABLESPACE chaud LOCATION '/SSD/tbl/chaud';
```

Il est possible d'avoir un tablespace par défaut pour une base de données, auquel cas tous les objets logiques créés dans cette base seront enregistrés physiquement dans le répertoire lié à ce tablespace. Il est aussi possible de créer des objets en indiquant spécifiquement un tablespace, ou de les déplacer d'un tablespace à un autre. Un objet spécifique ne peut appartenir qu'à un seul tablespace (autrement dit, un index ne pourra pas être enregistré sur deux tablespaces). Cependant, pour les objets partitionnés, le choix du tablespace peut se faire partition par partition.

Le but des tablespaces est de fournir une solution à des problèmes d'espace disque ou de performances. Si la partition où est stocké le répertoire des données principal se remplit fortement, il est possible de créer un tablespace dans une autre partition et donc d'utiliser l'espace disque de cette partition. Si de nouveaux disques plus rapides sont à disposition, il est possible de placer les objets fréquemment utilisés sur le tablespace contenant les disques rapides. Si des disques SSD sont à disposition, il est très intéressant d'y placer les index, les fichiers de tri temporaires, des tables de travail…

Par contre, contrairement à d'autres moteurs de bases de données, PostgreSQL n'a pas de notion de quotas. Les tablespaces ne peuvent donc pas être utilisés pour contraindre l'espace disque utilisé par certaines applications ou certains rôles.

1.3.5 Bases

- Conteneur hermétique
- Un rôle ne se connecte pas à une instance
  - il se connecte forcément à une base
- Une fois connecté, il ne voit que les objets de cette base
  - contournement : foreign data wrappers, dblink

Une base de données est un conteneur hermétique. En dehors des objets globaux, le rôle connecté à une base de données ne voit et ne peut interagir qu'avec les objets contenus dans cette base. De même, il ne voit pas les objets locaux des autres bases. Néanmoins, il est possible de lui donner le droit d'accéder à certains objets d'une autre base (de la même instance ou d'une autre instance) en utilisant les *Foreign Data Wrappers* (`postgres_fdw`) ou l'extension `dblink` .

Un rôle ne se connecte pas à l'instance. Il se connecte forcément à une base spécifique.

1.3.6 Schémas

- Espace de noms
- Sous-ensemble de la base
- Non lié à un utilisateur
- Résolution des objets : `search_path`
- `pg_catalog` , `information_schema`
  - pour catalogues système (lecture seule !)

Les schémas sont des espaces de noms à l'intérieur d'une base de données permettant :

- de grouper logiquement les objets d'une base de données ;
- de séparer les utilisateurs entre eux ;

- de contrôler plus efficacement les accès aux données ;
- d'éviter les conflits de noms dans les grosses bases de données.

Un schéma n'a à priori aucun lien avec un utilisateur donné.

Un schéma est un espace logique sans lien avec les emplacements physiques des données (ne pas confondre avec les *tablespaces*).

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble, tout dépend des droits dont il dispose. Depuis la version 15, un nouvel utilisateur n'a le droit de créer d'objet nulle part. Dans les versions précédentes, il avait accès au schéma `public` de chaque base et pouvait y créer des objets.

Lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête, PostgreSQL recherche les objets dans les schémas listés par le paramètre `search_path` valable pour la session en cours .

Voici un exemple d'utilisation des schémas :

```
-- Création de deux schémas
CREATE SCHEMA s1;
CREATE SCHEMA s2;

-- Création d'une table sans spécification du schéma
CREATE TABLE t1 (id integer);

-- Comme le montre la méta-commande \d, la table est créée dans le schéma public

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres

-- Ceci est dû à la configuration par défaut du paramètre search_path
-- modification du search_path
SET search_path TO s1;

-- création d'une nouvelle table sans spécification du schéma
CREATE TABLE t2 (id integer);

-- Cette fois, le schéma de la nouvelle table est s1
-- car la configuration du search_path est à s1
-- Nous pouvons aussi remarquer que les tables capitaines et s1
-- ne sont plus affichées
-- Ceci est dû au fait que le search_path ne contient que le schéma s1 et
-- n'affiche donc que les objets de ce schéma.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 s1     | t2   | table | postgres
```

```
-- Nouvelle modification du search_path
SET search_path TO s1, public;

-- Cette fois, les deux tables apparaissent

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
CREATE TABLE s2.t3 (id integer);

-- changement du search_path pour voir la table
SET search_path TO s1, s2, public;

-- La table apparaît bien, et le schéma d'appartenance est bien s2

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
-- attention, cette table a un nom déjà utilisé par une autre table
CREATE TABLE s2.t2 (id integer);

-- La création se passe bien car, même si le nom de la table est identique,
-- le schéma est différent
-- Par contre, \d ne montre que la première occurence de la table
-- ici, nous ne voyons t2 que dans s1

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Changeons le search_path pour placer s2 avant s1
SET search_path TO s2, s1, public;

-- Maintenant, la seule table t2 affichée est celle du schéma s2

postgres=# \d
                List of relations
```

```
 Schema |       Name         |   Type   |  Owner
--------+--------------------+----------+----------
 public | capitaines         | table    | postgres
 public | capitaines_id_seq  | sequence | postgres
 public | t1                 | table    | postgres
 s2     | t2                 | table    | postgres
 s2     | t3                 | table    | postgres
```

Tous ces exemples se basent sur des ordres de création de table. Cependant, le comportement serait identique sur d'autres types de commande (`SELECT` , `INSERT` , etc) et sur d'autres types d'objets locaux.

Pour des raisons de sécurité, il est très fortement conseillé de laisser le schéma `public` en toute fin du `search_path` . En effet, avant la version 15, s'il est placé au début, comme tout le monde avait le droit de créer des objets dans `public` , quelqu'un de mal intentionné pouvait placer un objet dans le schéma `public` pour servir de proxy à un autre objet d'un schéma situé après `public` . Même si la version 15 élimine ce risque, il reste la bonne pratique d'adapter le `search_path` pour placer les schémas applicatifs en premier.

Les schémas `pg_catalog` et `information_schema` contiennent des tables utilitaires (« catalogues système ») et des vues. Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables, et les colonnes, et des données de suivi interne. Dans PostgreSQL, ce sont de simples tables. Un simple utilisateur lit fréquemment ces tables, plus ou moins directement, mais n'a aucune raison d'y modifier des données. Toutes les opérations habituelles pour un utilisateur ou administrateur sont disponibles sous la forme de commandes SQL.

Ne modifiez jamais directement les tables et vues système dans les schémas `pg_catalog` et `information_schema` ; n'y ajoutez ni n'y effacez jamais rien !

Même si cela est techniquement possible, seules des exceptions particulièrement ésotériques peuvent justifier une modification directe des tables systèmes (par exemple, une correction de vue système, suite à un bug corrigé dans une version mineure). Ces tables n'apparaissent d'ailleurs pas dans une sauvegarde logique (`pg_dump`).

1.3.7 Tables

Par défaut, une table est :

- Permanente
  - si temporaire, vivra le temps de la session (ou de la transaction)
- Journalisée
  - si *unlogged*, perdue en cas de crash, pas de réplication
- Non partitionnée
  - partitionnement possible par intervalle, valeur ou hachage

Par défaut, les tables sont permanentes, journalisées et non partitionnées.

Il est possible de créer des tables temporaires (`CREATE TEMPORARY TABLE`). Celles-ci ne sont visibles que par la session qui les a créées et seront supprimées par défaut à la fin de cette session. Il est aussi possible de les supprimer automatiquement à la fin de la transaction qui les a créées. Il n'existe pas dans PostgreSQL de notion de table temporaire globale. Cependant, une extension[7] existe pour combler leur absence.

Pour des raisons de performance, il est possible de créer une table non journalisée (`CREATE UNLOGGED TABLE`). La définition de la table est journalisée mais pas son contenu. De ce fait, en cas de crash, il est impossible de dire si la table est corrompue ou non, et donc, au redémarrage du serveur, PostgreSQL vide la table de tout contenu. De plus, n'étant pas journalisée, la table n'est pas présente dans les sauvegardes PITR, ni repliquée vers d'éventuels serveurs secondaires.

Enfin, depuis la version 10, il est possible de partitionner les tables suivant un certain type de partitionnement : par intervalle, par valeur ou par hachage.

[7] https://github.com/darold/pgtt

1.3.8 Vues

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Vues matérialisées
  - à rafraîchir à une certaine fréquence

Le but des vues est de masquer une complexité, qu'elle soit du côté de la structure de la base ou de l'organisation des accès. Dans le premier cas, elles permettent de fournir un accès qui ne change pas même si les structures des tables évoluent. Dans le second cas, elles permettent l'accès à seulement certaines colonnes ou certaines lignes. De plus, les vues étant exécutées avec les mêmes droits que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Voici un exemple d'utilisation :

```
SET search_path TO public;

-- création de l'utilisateur guillaume
-- il n'aura pas accès à la table capitaines
-- par contre, il aura accès à la vue capitaines_anon
CREATE ROLE guillaume LOGIN;

-- ajoutons une colonne à la table capitaines
-- et ajoutons-y des données
ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
INSERT INTO capitaines (nom, age, num_cartecredit)
  VALUES ('Robert Surcouf', 20, '1234567890123456');

-- création de la vue
CREATE VIEW capitaines_anon AS
  SELECT nom, age, substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- ajout du droit de lecture à l'utilisateur guillaume
GRANT SELECT ON TABLE capitaines_anon TO guillaume;

-- connexion en tant qu'utilisateur guillaume
SET ROLE TO guillaume;

-- vérification qu'on lit bien la vue mais pas la table
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
```

```
 Robert Surcouf |  20 | 123456789******
```

```
-- tentative de lecture directe de la table
SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

Il est possible de modifier une vue en lui ajoutant des colonnes à la fin, au lieu de devoir les détruire et recréer (ainsi que toutes les vues qui en dépendent, ce qui peut être fastidieux).

Par exemple :

```
SET ROLE postgres;

CREATE OR REPLACE VIEW capitaines_anon AS SELECT
  nom,age,substring(num_cartecredit,0,10)||'******' AS num_cc_anon,
  md5(substring(num_cartecredit,0,10)) AS num_md5_cc
  FROM capitaines;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
      nom       | age |   num_cc_anon    |            num_md5_cc
----------------+-----+------------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

Nous pouvons aussi modifier les données au travers des vues simples, sans ajout de code et de trigger :

```
UPDATE capitaines_anon SET nom = 'Nicolas Surcouf' WHERE nom = 'Robert Surcouf';

SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
       nom       | age |   num_cc_anon    |            num_md5_cc
-----------------+-----+------------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

```
UPDATE capitaines_anon SET num_cc_anon = '123456789xxxxxx'
  WHERE nom = 'Nicolas Surcouf';

ERROR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

PostgreSQL gère le support natif des vues matérialisées (`CREATE MATERIALIZED VIEW nom_vue_mat AS SELECT …`). Les vues matérialisées sont des vues dont le contenu est figé sur disque, permettant de ne pas recalculer leur contenu à chaque appel. De plus, il est possible de les indexer pour accélérer leur consultation. Il faut cependant faire attention à ce que leur contenu reste synchrone avec le reste des données.

Les vues matérialisées ne sont pas mises à jour automatiquement, il faut demander explicitement le rafraîchissement (`REFRESH MATERIALIZED VIEW`). Avec la clause `CONCURRENTLY` , s'il y a un index d'unicité, le rafraîchissement ne bloque pas les sessions lisant en même temps les données d'une vue matérialisée.

```
-- Suppression de la vue
DROP VIEW capitaines_anon;
```

```
-- Création de la vue matérialisée
CREATE MATERIALIZED VIEW capitaines_anon AS
  SELECT nom,
    age,
    substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- Les données sont bien dans la vue matérialisée
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Mise à jour d'une ligne de la table
-- Cette mise à jour est bien effectuée, mais la vue matérialisée
-- n'est pas impactée
UPDATE capitaines SET nom = 'Robert Surcouf' WHERE nom = 'Nicolas Surcouf';

SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';

 id |      nom       | age | num_cartecredit
----+----------------+-----+------------------
  1 | Robert Surcouf |  20 | 1234567890123456

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Le résultat est le même mais le plan montre bien que PostgreSQL ne passe
-- plus par la table mais par la vue matérialisée :
EXPLAIN SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

                         QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on capitaines_anon  (cost=0.00..20.62 rows=1 width=68)
   Filter: (nom ~~ '%Surcouf'::text)

-- Après un rafraîchissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
REFRESH MATERIALIZED VIEW capitaines_anon;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions :

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;

ERROR:  cannot refresh materialized view "public.capitaines_anon" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns
       of the materialized view.
```

```
-- En effet, il faut un index d'unicité pour faire un rafraîchissement
-- sans bloquer les autres sessions.
CREATE UNIQUE INDEX ON capitaines_anon(nom);

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
```

1.3.9 Index

- Algorithmes supportés
  - B-tree (par défaut)
  - Hash
  - GiST / SP-GiST
  - GIN
  - BRIN (version 9.5)
  - Bloom (version 9.6)
- Type
  - Mono ou multi-colonnes
  - Partiel
  - Fonctionnel
  - Couvrant

PostgreSQL propose plusieurs algorithmes d'index.

Pour une indexation standard, nous utilisons en général un index Btree, de par ses nombreuses possibilités et ses très bonnes performances.

Les index hash sont peu utilisés, essentiellement dans la comparaison d'égalité de grandes chaînes de caractères.

Moins simples d'abord, les index plus spécifiques (GIN, GIST) sont spécialisés pour les grands volumes de données complexes et multidimensionnelles : indexation textuelle, géométrique, géographique, ou de tableaux de données par exemple.

Les index BRIN sont des index très compacts destinés aux grandes tables où les données sont fortement corrélées par rapport à leur emplacement physique sur les disques.

Les index bloom sont des index probabilistes visant à indexer de nombreuses colonnes interrogées simultanément. Ils nécessitent l'ajout d'une extension (nommée `bloom`). Contrairement aux index btree, les index bloom ne dépendant pas de l'ordre des colonnes.

Le module `pg_trgm` permet l'utilisation d'index dans des cas habituellement impossibles, comme les expressions rationnelles et les `LIKE '%...%'`.

Généralement, l'indexation porte sur la valeur d'une ou plusieurs colonnes. Il est néanmoins possible de n'indexer qu'une partie des lignes (index partiel) ou le résultat d'une fonction sur une ou plusieurs colonnes en paramètre. Enfin, il est aussi possible de modifier les index de certaines contraintes (unicité et clé primaire) pour inclure des colonnes supplémentaires.

Plus d'informations :

- Article Wikipédia sur les arbres B[8] ;
- Article Wikipédia sur les tables de hachage[9] ;
- Documentation officielle française[10].

1.3.10 Types de données

- Types de base
  - natif : `int`, `float`
  - standard SQL : `numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`
- Type complexe
  - tableau
  - JSON (`jsonb`), XML
  - vecteur (données LLM, FTS)
- Types métier
  - réseau, géométrique, etc.
- Types créés par les utilisateurs
  - structure SQL, C, Domaine, Enum

PostgreSQL dispose d'un grand nombre de types de base, certains natifs (comme la famille des `integer` et celle des `float`), et certains issus de la norme SQL (`numeric` , `char` , `varchar` , `date` , `time` , `timestamp` , `bool`).

Il dispose aussi de types plus complexes. Les tableaux (`array`) permettent de lister un ensemble de valeurs discontinues. Les intervalles (`range`) permettent d'indiquer toutes les valeurs comprises entre une valeur de début et une valeur de fin. Ces deux types dépendent évidemment d'un type de

base : tableau d'entiers, intervalle de dates, etc. Existent aussi les types complexes les données XML et JSON (préférer le type optimisé `jsonb`).

PostgreSQL sait travailler avec des vecteurs pour des calculs avancé. De base, le type `tsvector` permet la recherche plein texte, avec calcul de proximité de mots dans un texte, pondération des résultats, etc. L'extension `pgvector` permet de stocker et d'indexer des vecteurs utilisé par les algorithmes LLM implémentés dans les IA génératives.

Enfin, il existe des types métiers ayant trait principalement au réseau (adresse IP, masque réseau), à la géométrie (point, ligne, boite). Certains sont apportés par des extensions.

Tout ce qui vient d'être décrit est natif. Il est cependant possible de créer ses propres types de données, soit en SQL soit en C. Les possibilités et les performances ne sont évidemment pas les mêmes.

Voici comment créer un type en SQL :

```
CREATE TYPE serveur AS (
  nom             text,
  adresse_ip      inet,
  administrateur  text
);
```

Ce type de données va pouvoir être utilisé dans tous les objets SQL habituels : table, routine, opérateur (pour redéfinir l'opérateur `+` par exemple), fonction d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

(Il faut au préalable avoir défini le type `stock` et la fonction `stock_fusion`.)

Il est aussi possible de définir des domaines. Ce sont des types créés par les utilisateurs à partir d'un type de base et en lui ajoutant des contraintes supplémentaires.

En voici un exemple :

```
CREATE DOMAIN code_postal_francais AS text CHECK (value ~ '^\d{5}$');
ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
UPDATE capitaines SET cp = '35400' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET cp = '1420' WHERE nom = 'Haddock';

ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"

UPDATE capitaines SET cp = '01420' WHERE nom = 'Haddock';
SELECT * FROM capitaines;

 id |      nom       | age | num_cartecredit  |  cp
----+----------------+-----+------------------+-------
  1 | Robert Surcouf |  20 | 1234567890123456 | 35400
  1 | Haddock        |  35 |                  | 01420
```

Les domaines permettent d'intégrer la déclaration des contraintes à la déclaration d'un type, et donc de simplifier la maintenance de l'application si ce type peut être utilisé dans plusieurs tables : si la définition du code postal est insuffisante pour une évolution de l'application, il est possible de la modifier par un `ALTER DOMAIN`, et définir de nouvelles contraintes sur le domaine. Ces contraintes seront vérifiées sur l'ensemble des champs ayant le domaine comme type avant que la nouvelle version du type ne soit considérée comme valide.

Le défaut par rapport à des contraintes `CHECK` classiques sur une table est que l'information ne se trouvant pas dans la table, les contraintes sont plus difficiles à lister sur une table.

Enfin, il existe aussi les enums. Ce sont des types créés par les utilisateurs composés d'une liste ordonnée de chaînes de caractères.

En voici un exemple :

```
CREATE TYPE jour_semaine
  AS ENUM ('Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi',
  'Samedi', 'Dimanche');

ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;

UPDATE capitaines SET jour_sortie = 'Mardi' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET jour_sortie = 'Samedi' WHERE nom LIKE 'Haddock';

SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
```

```
 id |   nom   | age | num_cartecredit | cp | jour_sortie
----+---------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
```

Les *enums* permettent de déclarer une liste de valeurs statiques dans le dictionnaire de données plutôt que dans une table externe sur laquelle il faudrait rajouter des jointures : dans l'exemple, nous aurions pu créer une table `jour_de_la_semaine`, et stocker la clé associée dans `planning`. Nous aurions pu tout aussi bien positionner une contrainte `CHECK`, mais nous n'aurions plus eu une liste ordonnée.

Conférence de Heikki Linakangas sur la création d'un type color[11].

1.3.11 Contraintes

- `CHECK`
  - `prix > 0`
- `NOT NULL`
  - `id_client NOT NULL`
- Unicité
  - `id_client UNIQUE`
- Clés primaires
  - `UNIQUE NOT NULL` ==> `PRIMARY KEY (id_client)`
- Clés étrangères
  - `produit_id REFERENCES produits(id_produit)`
- `EXCLUDE`
  - `EXCLUDE USING gist (room WITH =, during WITH &&)`

Les contraintes sont la garantie de conserver des données de qualité ! Elles permettent une vérification qualitative des données, beaucoup plus fine qu'en définissant uniquement un type de données.

Les exemples ci-dessus reprennent :

- un prix qui doit être strictement positif ;
- un identifiant qui ne doit pas être vide (sinon des jointures filtreraient des lignes) ;
- une valeur qui doit être unique (comme des numéros de clients ou de facture) ;
- une clé primaire (unique non nulle), qui permet d'identifier précisément une ligne ;
- une clé étrangère vers la clé primaire d'une autre table (là encore pour garantir l'intégrité des jointures) ;
- une contrainte d'exclusion interdisant que deux plages temporelles se recouvrent dans la réservation de la même salle de réunion.

Les contraintes d'exclusion permettent un test sur plusieurs colonnes avec différents opérateurs (et non uniquement l'égalité, comme dans le cas d'une contrainte unique, qui n'est qu'une contrainte d'exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

Une contrainte peut porter sur plusieurs champs et un champ peut être impliqué dans plusieurs

contraintes :

```
CREATE TABLE commandes (
    no_commande     varchar(16) CHECK (no_commande ~ '^[A-Z0-9]*$'),
    id_entite_commerciale int REFERENCES entites_commerciales,
    id_client       int       REFERENCES clients,
    date_commande   date      NOT NULL,
    date_livraison  date      CHECK (date_livraison >= date_commande),
    PRIMARY KEY (no_commande, id_entite_commerciale)
);
```

```
\d commandes
                              Table « public.commandes »
        Colonne        |         Type          | … | NULL-able | Par défaut
-----------------------+-----------------------+---+-----------+------------
 no_commande           | character varying(16) |   | not null  |
 id_entite_commerciale | integer               |   | not null  |
 id_client             | integer               |   |           |
 date_commande         | date                  |   | not null  |
 date_livraison        | date                  |   |           |
Index :
    "commandes_pkey" PRIMARY KEY, btree (no_commande, id_entite_commerciale)
Contraintes de vérification :
    "commandes_check" CHECK (date_livraison >= date_commande)
    "commandes_no_commande_check" CHECK (no_commande::text ~ '^[A-Z0-9]*$'::text)
Contraintes de clés étrangères :
    "commandes_id_client_fkey" FOREIGN KEY (id_client) REFERENCES clients(id_client)
    "commandes_id_entite_commerciale_fkey" FOREIGN KEY (id_entite_commerciale)
 ↪  REFERENCES entites_commerciales(id_entite_commerciale)
```

Les contraintes doivent être vues comme la dernière ligne de défense de votre application face aux bugs. En effet, le code d'une application change beaucoup plus souvent que le schéma, et les données survivent souvent à l'application, qui peut être réécrite entretemps. Quoi qu'il se passe, des contraintes judicieuses garantissent qu'il n'y aura pas d'incohérence logique dans la base.

Si elles sont gênantes pour le développeur (car elles imposent un ordre d'insertion ou de mise à jour), il faut se rappeler que les contraintes peuvent être « débrayées » le temps d'une transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED ;
…
COMMIT ;
```

Les contraintes ne seront validées qu'au `COMMIT`.

Sur le sujet, voir par exemple *Constraints: a Developer's Secret Weapon*[12] de Will Leinweber (pgDay Paris 2018) (slides[13], vidéo[14]).

Du point de vue des performances, les contraintes permettent au planificateur d'optimiser les requêtes. Par exemple, le planificateur sait ne pas prendre en compte certaines jointures, notamment grâce à l'existence d'une contrainte d'unicité. (Sur ce point, la version 15 améliore les contraintes

[12] https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2192-constraints-a-developers-secret-weapon/
[13] https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
[14] https://youtu.be/hWh8QoV8z8k

d'unicité en permettant de choisir si la valeur NULL est considérée comme unique ou pas. Par défaut et historiquement, une valeur NULL n'étant pas égal à une valeur NULL, les valeurs NULL sont considérées distinctes, et donc on peut avoir plusieurs valeurs NULL dans une colonne ayant une contrainte d'unicité.)

1.3.12 Colonnes à valeur générée

- Valeur calculée à l'insertion
- `DEFAULT`
- Identité
  - `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`
- Expression
  - `GENERATED ALWAYS AS ( generation_expr ) STORED`

Une colonne a par défaut la valeur `NULL` si aucune valeur n'est fournie lors de l'insertion de la ligne. Il existe néanmoins trois cas où le moteur peut substituer une autre valeur.

Le plus connu correspond à la clause `DEFAULT`. Dans ce cas, la valeur insérée correspond à la valeur indiquée avec cette clause si aucune valeur n'est indiquée pour la colonne. Si une valeur est précisée, cette valeur surcharge la valeur par défaut. L'exemple suivant montre cela :

```
CREATE TABLE t2 (c1 integer, c2 integer, c3 integer DEFAULT 10);
INSERT INTO t2 (c1, c2, c3) VALUES (1, 2, 3);
INSERT INTO t2 (c1) VALUES (2);
SELECT * FROM t2;
```

```
 c1 | c2 | c3
----+----+----
  1 |  2 |  3
  2 |    | 10
```

La clause `DEFAULT` ne peut pas être utilisée avec des clauses complexes, notamment des clauses comprenant des requêtes.

Pour aller un peu plus loin, à partir de PostgreSQL 12, il est possible d'utiliser `GENERATED ALWAYS AS ( expression )`. Cela permet d'avoir une valeur calculée pour la colonne, valeur qui ne peut pas être surchargée, ni à l'insertion, ni à la mise à jour (mais qui est bien stockée sur le disque).

Comme exemple, nous allons reprendre la table `capitaines` et lui ajouter une colonne ayant comme valeur la version modifiée du numéro de carte de crédit :

```
ALTER TABLE capitaines
  ADD COLUMN num_cc_anon text
```

```
  GENERATED ALWAYS AS (substring(num_cartecredit, 0, 10) || '******') STORED;

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

      nom       | num_cartecredit  |   num_cc_anon
----------------+------------------+-----------------
 Robert Surcouf | 1234567890123456 | 123456789******
 Haddock        |                  |

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi', 'test');

ERROR:  cannot insert into column "num_cc_anon"
DETAIL:  Column "num_cc_anon" is a generated column.

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi');

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

          nom          | num_cartecredit  |   num_cc_anon
-----------------------+------------------+-----------------
 Robert Surcouf        | 1234567890123456 | 123456789******
 Haddock               |                  |
 Joseph Pradere-Niquet | 9876543210987654 | 987654321******
```

Enfin, `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY` permet d'obtenir une colonne d'identité, bien meilleure que ce que le pseudo-type `serial` propose. Si `ALWAYS` est indiqué, la valeur n'est pas modifiable.

```
ALTER TABLE capitaines
  ADD COLUMN id2 integer GENERATED ALWAYS AS IDENTITY;

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3

INSERT INTO capitaines (nom) VALUES ('Tom Souville');

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3
 Tom Souville          |   4
```

Le type `serial` est remplacé par le type `integer` et une séquence comme le montre l'exemple suivant. C'est un problème dans la mesure ou la déclaration qui est faite à la création de la table produit un résultat différent en base et donc dans les exports de données.

```
CREATE TABLE tserial(s serial);
```

```
                          Table "public.tserial"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 s      | integer |           | not null | nextval('tserial_s_seq'::regclass)
```

1.3.13 Langages

- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQL
- Extensions officielles : Perl, Python
- Mais aussi Java, Ruby, Javascript…
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

- PL/pgSQL ;
- PL/Perl[15] ;
- PL/Python[16] ;
- PL/Tcl.

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

- PL/sh[17] ;
- PL/R[18] ;
- PL/Java[19] ;
- PL/lolcode ;
- PL/Scheme ;
- PL/PHP ;
- PL/Ruby ;
- PL/Lua[20] ;
- PL/pgPSM ;
- PL/v8[21] (Javascript).

[15] https://docs.postgresql.fr/current/plperl.html
[16] https://docs.postgresql.fr/current/plpython.html
[17] https://github.com/petere/plsh
[18] https://github.com/postgres-plr/plr
[19] https://tada.github.io/pljava/
[20] https://github.com/pllua/pllua
[21] https://github.com/plv8/plv8

Tableau des langages supportés[22].

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

1.3.14 Fonctions & procédures

- Fonction
  - renvoie une ou plusieurs valeurs
  - `SETOF` ou `TABLE` pour plusieurs lignes
- Procédure (v11+)
  - ne renvoie rien
  - peut gérer le transactionnel dans certains cas

Historiquement, PostgreSQL ne proposait que l'écriture de fonctions. Depuis la version 11, il est aussi possible de créer des procédures. Le terme « routine » est utilisé pour signifier procédure ou fonction.

Une fonction renvoie une donnée. Cette donnée peut comporter une ou plusieurs colonnes. Elle peut aussi avoir plusieurs lignes dans le cas d'une fonction `SETOF` ou `TABLE`.

Une procédure ne renvoie rien. Elle a cependant un gros avantage par rapport aux fonctions dans le fait qu'elle peut gérer le transactionnel. Elle peut valider ou annuler la transaction en cours. Dans ce cas, une nouvelle transaction est ouverte immédiatement après la fin de la transaction précédente.

1.3.15 Opérateurs

- Dépend d'un ou deux types de données
- Utilise une fonction prédéfinie :

```
CREATE OPERATOR //
  (FUNCTION=division0,
  LEFTARG=integer,
  RIGHTARG=integer);
```

Il est possible de créer de nouveaux opérateurs sur un type de base ou sur un type utilisateur. Un opérateur exécute une fonction, soit à un argument pour un opérateur unitaire, soit à deux arguments pour un opérateur binaire.

Voici un exemple d'opérateur acceptant une division par zéro sans erreur :

```
-- définissons une fonction de division en PL/pgSQL
CREATE FUNCTION division0 (p1 integer, p2 integer) RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
  IF p2 = 0 THEN
      RETURN NULL;
  END IF;

  RETURN p1 / p2;
END
$$;

-- créons l'opérateur
CREATE OPERATOR // (FUNCTION = division0, LEFTARG = integer, RIGHTARG = integer);

-- une division normale se passe bien

SELECT 10/5;

 ?column?
----------
        2

SELECT 10//5;

 ?column?
----------
        2
```

```
-- une division par 0 ramène une erreur avec l'opérateur natif
SELECT 10/0;

ERROR:  division by zero

-- une division par 0 renvoie NULL avec notre opérateur
SELECT 10//0;

 ?column?
----------

(1 row)
```

1.3.16 Triggers

- Opérations : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- Trigger sur :
  - une colonne, et/ou avec condition
  - une vue
  - DDL
- Tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)
- N'importe quel langage supporté

Les triggers peuvent être exécutés avant (`BEFORE`) ou après (`AFTER`) une opération.

Il est possible de les déclencher pour chaque ligne impactée (`FOR EACH ROW`) ou une seule fois pour l'ensemble de la requête (`FOR STATEMENT`). Dans le premier cas, il est possible d'accéder à la ligne impactée (ancienne et nouvelle version). Dans le deuxième cas, il a fallu attendre la version 10 pour disposer des tables de transition qui donnent à l'utilisateur une vision des lignes avant et après modification.

Par ailleurs, les triggers peuvent être écrits dans n'importe lequel des langages de routine supportés par PostgreSQL (C, PL/pgSQL, PL/Perl, etc.)

Exemple :

```
ALTER TABLE capitaines ADD COLUMN salaire integer;

CREATE FUNCTION verif_salaire()
RETURNS trigger AS $verif_salaire$
```

```
BEGIN
  -- Nous verifions que les variables ne sont pas vides
  IF NEW.nom IS NULL THEN
    RAISE EXCEPTION 'Le nom ne doit pas être null.';
  END IF;

  IF NEW.salaire IS NULL THEN
    RAISE EXCEPTION 'Le salaire ne doit pas être null.';
  END IF;

  -- pas de baisse de salaires !
  IF NEW.salaire < OLD.salaire THEN
    RAISE EXCEPTION 'Pas de baisse de salaire !';
  END IF;

  RETURN NEW;
END;
$verif_salaire$ LANGUAGE plpgsql;

CREATE TRIGGER verif_salaire BEFORE INSERT OR UPDATE ON capitaines
  FOR EACH ROW EXECUTE PROCEDURE verif_salaire();

UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 3000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';

ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

1.3.17 Questions

N'hésitez pas, c'est le moment !

1.4 QUIZ

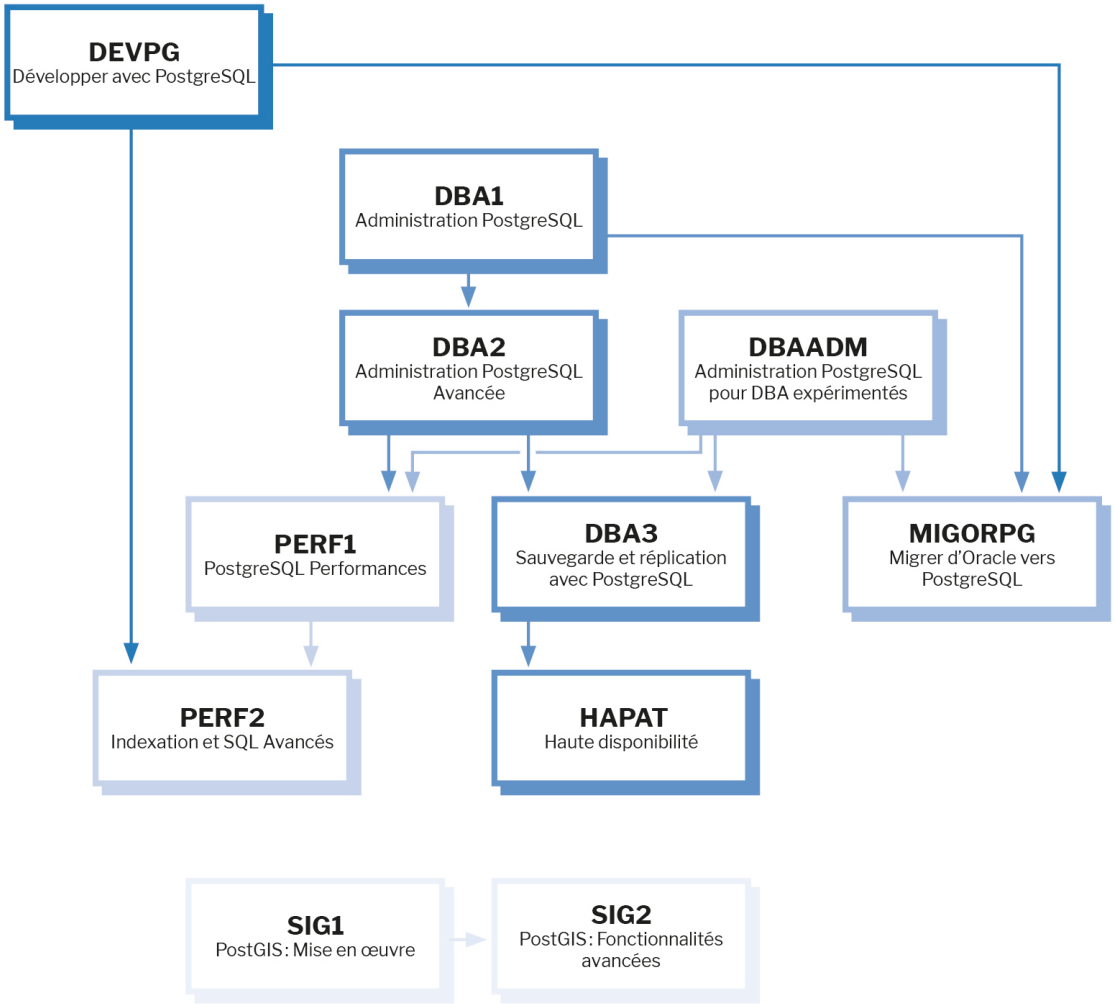
https://dali.bo/a2_quiz

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
- 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.