Formation SQL3

Développer avec PostgreSQL

17.12

Dalibo SCOP

https://dalibo.com/formations

Développer avec PostgreSQL

Formation SQL3

TITRE : Développer avec PostgreSQL
SOUS-TITRE : Formation SQL3

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-07-4

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, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale: Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note: Ceci est un résumé de la licence.

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

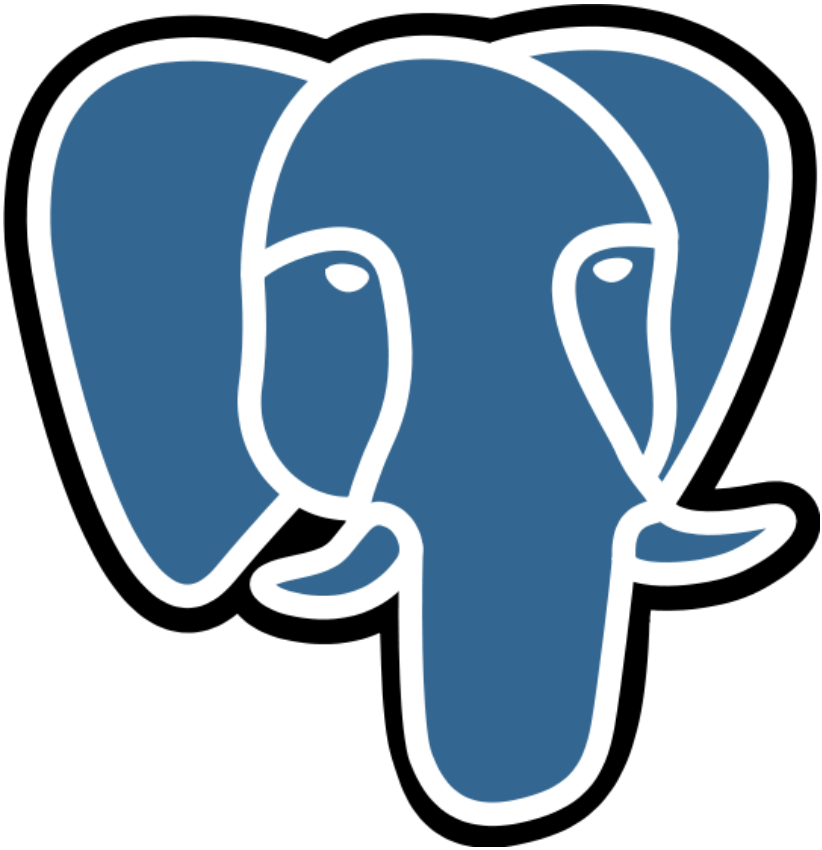
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 DÉCOUVRIR POSTGRESQL



2.1 PRÉAMBULE

- Quelle histoire !
  - parmi les plus vieux logiciels libres
  - et les plus sophistiqués
- Souvent cité comme exemple
  - qualité du code

  - indépendance des développeurs
  - réactivité de la communauté

L'histoire de PostgreSQL est longue, riche et passionnante. Au côté des projets libres Apache et Linux, PostgreSQL est l'un des plus vieux logiciels libres en activité et fait partie des SGBD les plus sophistiqués à l'heure actuelle.

Au sein des différentes communautés libres, PostgreSQL est souvent utilisé comme exemple à différents niveaux :

- qualité du code ;
- indépendance des développeurs et gouvernance du projet ;
- réactivité de la communauté ;
- stabilité et puissance du logiciel.

Tous ces atouts font que PostgreSQL est désormais reconnu et adopté par des milliers de grandes sociétés de par le monde.

2.1.1 AU MENU

1. Origines et historique du projet
2. Versions et feuille de route
3. Concepts de base
4. Fonctionnalités
5. Sponsors et références

Cette première partie est un tour d'horizon pour découvrir les multiples facettes du système de base de données libre PostgreSQL.

Les deux premières parties expliquent la genèse du projet et détaillent les différences entres les versions successives du logiciel. Puis nous ferons un rappel théorique sur les principes fondateurs de PostgreSQL (ACID, MVCC, transactions, journaux de transactions) ainsi que sur les fonctionnalités essentielles (schémas, index, tablespaces, triggers).

Nous terminerons par un panorama d'utilisateurs renommés et de cas d' utilisations remarquables.

2.1.2 OBJECTIFS

- Comprendre les origines du projet

- Revoir les principes fondamentaux
- Découvrir des exemples concrets

PostgreSQL est un des plus vieux logiciels Open-Source ! Comprendre son histoire permet de mieux réaliser le chemin parcouru et les raisons de son succès. Par ailleurs, un rappel des concepts de base permet d'avancer plus facilement lors des modules suivants. Enfin, une série de cas d'utilisation et de références sont toujours utiles pour faire le choix de PostgreSQL en ayant des repères concrets.

2.2 UN PEU D'HISTOIRE...

- La licence
- L'origine du nom
- Les origines du projet
- Les principes
- La philosophie

2.2.1 LICENCE

- Licence PostgreSQL
  - BSD / MIT
  - http://www.postgresql.org/about/licence/
- Droit de
  - utiliser, copier, modifier, distribuer sans coût de licence
- Reconnu par l'Open Source Initiative
  - http://opensource.org/licenses/PostgreSQL
- Utilisé par un grand nombre de projets de l'écosystème

PostgreSQL est distribué sous une licence spécifique, combinant la licence BSD et la licence MIT. Cette licence spécifique est reconnue comme une licence libre par l'Open Source Initiative.

Cette licence vous donne le droit de distribuer PostgreSQL, de l'installer, de le modifier... et même de le vendre. Certaines sociétés, comme EnterpriseDB, produisent leur version de PostgreSQL de cette façon.

Cette licence a ensuite été reprise par de nombreux projets de la communauté
pgAdmin, pgcluu, pgstat, etc.

2.2.2 POSTGRESQL ?!?!

- Michael Stonebraker recode Ingres
- post « ingres » => post~~in~~gres => postgres
- postgres => PostgreSQL

L'origine du nom PostgreSQL remonte à la base de données Ingres, développée à l'université de Berkeley par Michael Stonebraker. En 1985, il prend la décision de reprendre le développement à partir de zéro et nomme ce nouveau logiciel Postgres, comme raccourci de post-Ingres.

En 1995, avec l'ajout du support du langage SQL, Postgres fut renommé Postgres95 puis PostgreSQL.

Aujourd'hui, le nom officiel est « PostgreSQL » (prononcez « post - gresse - Q - L »). Cependant, le nom « Postgres » est accepté comme alias.

Pour aller plus loin :

- Fil de discussion sur les listes de discussion[1]
- Article sur le wiki officiel[2]

2.2.3 PRINCIPES FONDATEURS

- Sécurité des données (ACID)
- Respect des normes (ISO SQL)
- Fonctionnalités
- Performances
- Simplicité du code

Depuis son origine, PostgreSQL a toujours privilégié la stabilité et le respect des standards plutôt que les performances.

Ceci explique en partie la réputation de relative lenteur et de complexité face aux autres SGBD du marché. Cette image est désormais totalement obsolète, notamment grâce aux avancées réalisées depuis les versions 8.x.

[1] http://archives.postgresql.org/pgsql-advocacy/2007-11/msg00109.php
[2] http://wiki.postgresql.org/wiki/Postgres

2.2.4 ORIGINES

- Années 1970 : **Ingres** est développé à Berkeley
- 1985 : **Postgres** succède à Ingres
- 1995 : Ajout du langage **SQL**.
- 1996 : Postgres devient **PostgreSQL**
- 1996 : Création du **PostgreSQL Global Development Group**

L'histoire de PostgreSQL remonte à la base de données Ingres, développée à Berkeley par Michael Stonebraker. Lorsque ce dernier décida en 1985 de recommencer le développement de zéro, il nomma le logiciel Postgres, comme raccourci de post-Ingres. Lors de l'ajout des fonctionnalités SQL en 1995 par deux étudiants chinois de Berkeley, Postgres fut renommé Postgres95. Ce nom fut changé à la fin de 1996 en PostgreSQL lors de la libération du code source de PostgreSQL.

De longs débats enflammés animent toujours la communauté pour savoir s'il faut revenir au nom initial Postgres.

À l'heure actuelle, le nom Postgres est accepté comme un alias du nom officiel PostgreSQL.

Plus d'informations :

- Page associée sur le site officiel[3]

2.2.5 ORIGINES (ANNÉES 2000)

Apparitions de la communauté internationale

- ~ 2000: Communauté japonaise
- 2004 : Communauté francophone
- 2006 : SPI
- 2007 : Communauté italienne
- 2008 : PostgreSQL Europe et US
- 2009 : Boom des PGDay

Les années 2000 voient l'apparition de communautés locales organisées autour d'association ou de manière informelle. Chaque communauté organise la promotion, la diffusion d'information et l'entraide à son propre niveau.

[3] http://www.postgresql.org/about/history

17.12

En 2000 apparaît la communauté japonaise. Elle dispose d'un grand groupe, capable de réaliser des conférences chaque année. Elle compte au dernier recensement connu, plus de 3000 membres.

En 2004 nait l'association française (loi 1901) appelée PostgreSQLfr. Cette association a pour but de fournir un cadre légal pour pouvoir participer à certains événements comme Solutions Linux, les RMLL ou le pgDay 2008 à Toulouse. Elle permet aussi de récolter des fonds pour aider à la promotion de PostgreSQL.

En 2006, le PGDG intègre le « Software in the Public Interest », Inc. (SPI), une organisation à but non lucratif chargée de collecter et redistribuer des financements. Ce n'est pas une organisation spécifique à PostgreSQL. Elle a été créée à l'initiative de Debian et dispose aussi de membres comme OpenOffice.org.

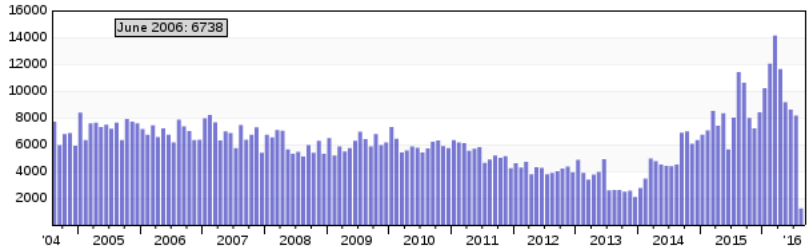
En 2008, douze ans après la création du projet, des associations d' utilisateurs apparaissent pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. PostgreSQL UK organise une journée de conférences à Londres, PostgreSQL Fr en organise une à Toulouse. Des « sur-groupes » apparaissent aussi pour aider les groupes. PGUS apparaît pour consolider les différents groupes américains d'utilisateurs PostgreSQL. Ces derniers étaient plutôt créés géographiquement, par état ou grosse ville. Ils peuvent rejoindre et être aidés par cette organisation. De même en Europe arrive PostgreSQL Europe, une association chargée d'aider les utilisateurs de PostgreSQL souhaitant mettre en place des événements. Son principal travail est l'organisation d'un événement majeur en Europe tous les ans : pgconf.eu. Cet événement a eu lieu la première fois en France (sous le nom pgday.eu) à Paris, en 2009, puis en Allemagne à Stuttgart en 2010, en 2011 à Amsterdam, à Prague en 2012, à Dublin en 2013 et à Madrid en 2014. Cependant, elle aide aussi les communautés allemandes et suédoise à monter leur propre événément (respectivement pgconf.de et nordic pgday).

En 2010, on dénombre plus d'une conférence par mois consacrée uniquement à PostgreSQL dans le monde.

- Communauté japonaise[4]
- Communauté francophone[5]
- Communauté italienne[6]
- Communauté européenne[7]
- Communauté US[8]

[4] http://www.postgresql.jp
[5] http://www.postgresql.fr
[6] http://www.itpug.org
[7] http://www.postgresql.eu
[8] http://www.postgresql.us

2.2.6 PROGRESSION DU PROJET - ÉCHANGES DE MAIL



Ce graphe représente l'évolution du trafic des listes de diffusion du projet qui est corollaire du nombre d'utilisateurs du logiciel.

On remarque une augmentation très importante jusqu'en 2005 puis une petite chute en 2008, un peu récupérée en 2009, un nouveau creux fin 2012 jusqu'en début 2013, un trafic stable autour de 4500 mails par mois en 2014 et depuis une progression constante pour arriver en 2016 à des pics de 12000 mails par mois.

La moyenne actuelle est d'environ 250 messages par jour sur les 23 listes actives.

Source du graphe[9]

On peut voir l'importance de ces chiffres en comparant le trafic des listes PostgreSQL et MySQL (datant de février 2008) sur ce lien[10] .

Début 2014, il y a moins de 1 message par jour sur les listes MySQL tel que mesuré sur Markmail.

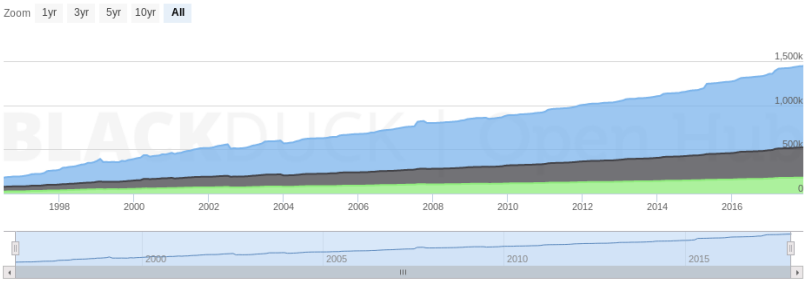
Pour aller plus loin : On peut également visualiser l'évolution des contributions de la communauté PostgreSQL grâce au projet Code Swarm[11] .

---

[9] http://postgresql.markmail.org/
[10] http://markmail.blogspot.com/2008/02/postgresql-more-traffic-than-mysql-and.html
[11] http://www.vimeo.com/1081680

17.12

2.2.7 PROGRESSION DU CODE



Ce graphe représente l'évolution du nombre de lignes de code dans les sources de PostgreSQL. Cela permet de bien visualiser l'évolution du projet en terme de développement.

On note une augmentation constante depuis 2000 avec une croissance régulière d 'environ 25000 lignes de code C par an. Le plus intéressant est certainement de noter que l'évolution est constante.

Actuellement, PostgreSQL est composé de plus de 1.000.000 de lignes (dont 270.000 lignes de commentaires), pour environ 200 développeurs actifs.

Source[12] .

2.3 LES VERSIONS

- Versions obsolètes : 9.2 et antérieures
- Versions actuelles : de 9.3 à 10
- Version en cours de développement : 11
- Versions dérivées

2.3.1 HISTORIQUE

- 1996 : v1.0 -> première version publiée
- 2003 : v7.4 -> première version *réellement* stable
- 2005 : v8.0 -> arrivée sur Windows

[12] https://www.openhub.net/p/postgres

- 2008 : v8.3 -> performance et fonctionnalités
- 2010 : v9.0 -> réplication intégrée
- 2016 : v9.6 -> parallélisation
- 2017 : v10 -> réplication logique

La version 7.4 est la première version réellement stable. La gestion des journaux de transactions a été nettement améliorée, et de nombreuses optimisations ont été apportées au moteur.

La version 8.0 marque l'entrée tant attendue de PostgreSQL dans le marché des SGDB de haut niveau, en apportant des fonctionnalités telles que les tablespaces, les procédures stockées en Java, le Point In Time Recovery, la réplication asynchrone ainsi qu'une version native pour Windows.

La version 8.3 se focalise sur les performances et les nouvelles fonctionnalités. C'est aussi la version qui a causé un changement important dans l'organisation du développement : gestion des commit fests, création de l 'outil web commitfest, etc.

Les versions 9.x sont axées réplication physique. La 9.0 intègre un système de réplication asynchrone asymétrique. La version 9.1 ajoute une réplication synchrone et améliore de nombreux points sur la réplication (notamment pour la partie administration et supervision). La version 9.2 apporte la réplication en cascade. La 9.3 ajoute quelques améliorations supplémentaires. La version 9.4 apporte également un certain nombre d'améliorations, ainsi que les premières briques pour l'intégration de la réplication logique dans PostgreSQL. La version 9.6 apporte la parallélisation, ce qui était attendu par de nombreux utilisateurs.

La version 10 propose beaucoup de nouveautés, comme une amélioration nette de la parallélisation et du partitionnement, mais surtout l'ajout de la réplication logique.

Il est toujours possible de télécharger les sources depuis la version 1.0 jusqu'à la version courante sur postgresql.org[13] .

2.3.2 NUMÉROTATION

- Avant la version 10
  - X.Y : version majeure (8.4, 9.6)
  - X.Y.Z : version mineure (9.6.4)
- Après la version 10
  - X : version majeure (10, 11)

[13] http://www.postgresql.org/ftp/source/

17.12

  - X.Y : version mineure (10.1)

Une version majeure apporte de nouvelles fonctionnalités, des changements de comportement, etc. Une version majeure sort généralement tous les 12/15 mois.

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Elles sont plus fréquentes que les versions majeures, avec un rythme de sortie de l'ordre des trois mois, sauf bugs majeurs ou failles de sécurité. Chaque bug est corrigé dans toutes les versions stables actuellement maintenues par le projet.

2.3.3 VERSIONS COURANTES

- Dernières releases (9 novembre 2017) :
  - version 9.3.20
  - version 9.4.15
  - version 9.5.10
  - version 9.6.6
  - version 10.1
- Prochaine sortie, 8 février 2018

La philosophie générale des développeurs de PostgreSQL peut se résumer ainsi :

« Notre politique se base sur la qualité, pas sur les dates de sortie. »

Toutefois, même si cette philosophie reste très présente parmi les développeurs, depuis quelques années, les choses évoluent et la tendance actuelle est de livrer une version stable majeure tous les 12 à 15 mois, tout en conservant la qualité des versions. De ce fait, toute fonctionnalité supposée pas suffisamment stable est repoussée à la version suivante.

Le support de la version 7.3 a été arrêté au début de l'année 2008. La même chose est arrivée aux versions 7.4 et 8.0 milieu 2010, à la 8.1 en décembre 2010, à la 8.2 en décembre 2011, à la 8.3 en février 2013, à la 8.4 en juillet 2014 et la 9.0 en septembre 2015, la 9.1 en septembre 2016. La prochaine version qui subira ce sort est la 9.2, en septembre 2017.

La tendance actuelle est de garantir un support pour chaque version courante pendant une durée minimale de 5 ans.

Pour plus de détails : Politique de versionnement[14] .

---

[14] http://www.postgresql.org/support/versioning/

2.3.4 VERSION 8.4

- Juillet 2009 - juillet 2014 (cette version n'est plus maintenue)
- Fonctions Window (clause `OVER`)
- CTE (vues non persistantes) et requêtes récursives
- Infrastructure SQL/MED (données externes)
- Paramètres par défaut et nombre variant de paramètres pour les fonctions
- Restauration parallélisée d'une sauvegarde
- Droits sur les colonnes

De plus, cette version apporte des améliorations moins visibles telles que :

- locale configurable par base de données ;
- refonte du FSM ;
- VACUUM sélectif grâce au « visibility map » ;
- support des certificats SSL ;
- statistiques sur les fonctions ;
- fonction `pg_terminate_backend()` ;
- nouveaux modules contrib : `pg_stat_statements`, `auto_explain`, ...

Exemple de la volonté d'intégrer des fonctionnalités totalement matures, le Hot Standby, fonctionnalité très attendue par les utilisateurs, a finalement été repoussé pour la version suivante car les développeurs estimaient qu' elle n'était pas assez fiable.

Pour plus de détails :

- Release notes[15]
- Article GLMF[16]

Cette version n'est plus maintenue depuis juillet 2014.

2.3.5 VERSION 9.0

- Septembre 2010 - septembre 2015
- Hot Standby + Streaming Replication
- Contraintes d'exclusion
- Améliorations pour l'`EXPLAIN`
- Contrainte `UNIQUE` différable
- Droits par défaut, `GRANT ALL`

[15] http://www.postgresql.org/docs/current/interactive/release-8-4.html
[16] http://www.dalibo.org/hs44_postgresql_8.4

17.12

- Triggers sur colonne, et clause `WHEN`

Mais aussi :

- droit d'accès aux « Large Objects » ;
- configurations par utilisateurs, par bases de données mais aussi par couple utilisateur/base ;
- bloc de code anonyme.

Pour plus de détails :

- Traduction annonce 9.0[17]
- Présentation sur la version 9.0[18]
- Article GLMF sur la réplication, partie 1[19]
- Article GLMF sur la réplication, partie 2[20]
- Article GLMF sur les autres nouveautés de la versino 9.0[21]

L'arrêt du support de cette version surviendra en septembre 2015.

2.3.6 VERSION 9.1

- Septembre 2011 - septembre 2016
- Réplication synchrone
- Supervision et administration plus aisée de la réplication
- Gestion des extensions
- Support des tables distantes via SQL/MED
- Support des labels de sécurité
- Support des tables non journalisées

Et beaucoup d'autres :

- moins de verrous pour les `DDL` (`ALTER TABLE`, `TRIGGER`) ;
- réduction de la taille des champs de type `NUMERIC` sur disque ;
- compteurs du nombre de `VACUUM` et `ANALYZE` pour les tables `pg_stat_*_tables` ;
- le support des triggers sur les vues.

Pour plus de détails :

[17] http://www.dalibo.org/annonce_9.0
[18] http://www.dalibo.org/quoi_de_neuf_dans_postgresql_9
[19] http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_1
[20] http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_2
[21] http://www.dalibo.org/glmf134_postgresql_les_autres_nouveautes

- Page officielle des nouveautés de la version 9.1[22]
- Article GLMF sur la version 9.1, partie 1[23]
- Article GLMF sur la version 9.1, partie 2[24]

2.3.7 VERSION 9.2

- Septembre 2012 - septembre 2017
- Réplication en cascade
  - `pg_basebackup` utilisable sur un esclave
  - réplication synchrone en mémoire seulement sur l'esclave
- Axé performances
  - amélioration de la scalabilité (lecture et écriture)
  - parcours d'index seuls
- Support de la méthode d'accès SP-GiST pour les index
- Support des types d'intervalles de valeurs
- Support du type de données JSON

Et beaucoup d'autres :

- amélioration des `EXPLAIN` ;
- nouveau processus `checkpointer` ;
- utilisation d'algorithmes spécialisés plus rapides pour le tri ;
- nouveau paramètre pour limiter la taille des fichiers temporaires ;
- nouvel outil `pg_receivexlog` (à présent `pg_receivewal`) ;
- annulation de ses propres requêtes avec `pg_cancel_backend()` par un utilisateur de base ;
- possibilité de déplacer plus facilement un tablespace, moteur éteint ;
- plus de traces pour l'activité disque d'autovacuum ;
- nouveaux champs dans les vues statistiques `pg_stat_activity`, `pg_stat_database` et `pg_stat_bgwriter` ;
- ajout des vues avec « barrière de sécurité » ;
- ajout des fonctions `LEAKPROOF` ;
- support des labels de sécurité sur les objets partagés.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.2[25]

[22] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.1/fr
[23] http://www.dalibo.org/glmf145_nouveautes_de_postgresql_9.1_partie_1
[24] http://www.dalibo.org/glmf145_nouveautes_de_postgresql_9.1_partie_2
[25] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.2

17.12

- Workshop Dalibo sur la version 9.2[26]

2.3.8 VERSION 9.3

- Septembre 2013 - septembre 2018 (?)
- Meilleure gestion de la mémoire partagée
- Support de la clause `LATERAL` dans un `SELECT`
- 4 To maxi au lieu de 2 Go pour les Large Objects
- `COPY FREEZE`
- Vues en mise à jour
- Vues matérialisées

Et d'autres encore :

- le FDW PostgreSQL (`postgres_fdw`) ;
- failover rapide ;
- configuration du `recovery.conf` par pg_basebackup ;
- `pg_dump` parallélisé ;
- background workers ;
- etc.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.3[27]
- Workshop Dalibo sur la version 9.3[28]

2.3.9 VERSION 9.4

- décembre 2014 - décembre 2019 (?)
- amélioration des index GIN (taille réduite et meilleures performances)
- nouveau type JSONB
- rafraîchissement sans verrou des vues matérialisées
- possibilité pour une instance répliquée de mémoriser la position des instances secondaires (*replication slots*)
- décodage logique (première briques pour la réplication logique intégrée)

Et beaucoup d'autres :

[26] https://kb.dalibo.com/conferences/postgresql_9.2/presentation
[27] http://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.3
[28] https://kb.dalibo.com/conferences/workshop_9.3/presentation

- clause `WITH CHECK OPTION` pour les vues automatiquement modifiables ;
- clauses `FILTER` et `WITHIN GROUP` pour les agrégats ;
- commande `SQL ALTER SYSTEM` pour modifier `postgresql.conf` ;
- améliorations des fonctions d'agrégat ;
- nouvelle contrib : `pg_prewarm` ;
- nouvelles options de formatage pour `log_line_prefix` ;
- background workers dynamiques...

Il y a eu des soucis détectés pendant la phase de bêta sur la partie JSONB. La correction de ceux-ci a nécessité de repousser la sortie de cette version de trois mois le temps d'effectuer les tests nécessaires. La version stable est sortie le 18 décembre 2014.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.4[29]
- Workshop Dalibo sur la version 9.4[30]

2.3.10 VERSION 9.5

- Janvier 2016 - Janvier 2021 (?)
- Row Level Security
- Index `BRIN`
- `INSERT ... ON CONFLICT { UPDATE | IGNORE }`
- `SKIP LOCKED`
- `SQL/MED`
  - import de schéma, héritage
- Supervision
  - amélioration de `pg_stat_statements`, ajout de `pg_stat_ssl`
- fonctions OLAP (`GROUPING SETS`, `CUBE` et `ROLLUP`)

Pour plus de détails :

- Page officielle des nouveautés de la version 9.5[31]
- Workshop Dalibo sur la version 9.5[32]

---

[29] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.4
[30] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.4
[31] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.5
[32] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.5

2.3.11 VERSION 9.6

- Septembre 2016 - Septembre 2021 (?)
- Parallélisation
  - parcours séquentiel, jointure, agrégation
- SQL/MED
  - tri distant, jointures impliquant deux tables distantes
- Réplication synchrone
- MVCC
  - `VACUUM FREEZE`, `CHECKPOINT`, ancien snapshot
- Maintenance

Le développement de cette version a commencé en mai 2015. La première bêta est sortie en mai 2016. La version stable est sortie le 29 septembre 2016.

La fonctionnalité majeure sera certainement l'intégration du parallélisme de certaines parties de l'exécution d'une requête.

Pour plus de détails :

- Page officielle des nouveautés de la version 9.6[33]
- Workshop Dalibo sur la version 9.6[34]

2.3.12 VERSION 10

- Septembre 2017 - Septembre 2022 (?)
- Meilleure parallélisation
  - parcours d'index, jointure MergeJoin, sous-requêtes corrélées
- Réplication logique
- Partitionnement

La fonctionnalité majeure est de loin l'intégration de la réplication logique. Cependant d'autres améliorations devraient attirer les utilisateurs comme celles concernant le partitionnement, les tables de transition ou encore les améliorations sur la parallélisation.

Pour plus de détails : * Page officielle des nouveautés de la version 10[35] * Workshop Dalibo sur la version 10[36]

---

[33] https://wiki.postgresql.org/wiki/NewIn96
[34] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.6
[35] https://wiki.postgresql.org/wiki/New_in_postgres_10
[36] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_10

2.3.13 PETIT RÉSUMÉ

- Versions 7
  - fondations
  - durabilité
- Versions 8
  - fonctionnalités
  - performances
- Versions 9
  - réplication physique
  - extensibilité
- Versions 10
  - réplication logique
  - parallélisation

Si nous essayons de voir cela avec de grosses mailles, les développements des versions 7 ciblaient les fondations d'un moteur de bases de données stable et durable. Ceux des versions 8 avaient pour but de rattraper les gros acteurs du marché en fonctionnalités et en performances. Enfin, pour les versions 9, on est plutôt sur la réplication et l'extensibilité.

La version 10 se base principalement sur la parallélisation des opérations (développement mené principalement par EnterpriseDB) et la réplication logique (par 2ndQuadrant).

2.3.14 QUELLE VERSION UTILISER ?

- 9.2 et inférieures
  - **Danger !**
- 9.3
  - planifier une migration rapidement
- 9.4, 9.5 et 9.6
  - mise à jour uniquement
- 10
  - nouvelles installations et nouveaux développements

Si vous avez une version 9.2 ou inférieure, planifiez le plus rapidement possible une migration vers une version plus récente, comme la 9.3 ou la 9.4.

La 9.2 n'est plus maintenue à compter de septembre 2017. Si vous utilisez cette version, il serait bon de commencer à étudier une migration de version dès que possible.

17.12

Pour les versions 9.3, 9.4, 9.5 et 9.6, le plus important est d'appliquer les mises à jour correctives.

La version 10 est officiellement stable depuis septembre 2017. Cette version peut être utilisée pour les nouvelles installations en production et les nouveaux développements. Son support est assuré jusqu'en septembre 2022.

Tableau comparatif des versions[37] .

2.3.15 VERSIONS DÉRIVÉES / FORKS

- Compatibilité Oracle
  - EnterpriseDB Postgres Plus
- Data warehouse
  - Greenplum
  - Netezza
  - Amazon RedShift

Il existe de nombreuses versions dérivées de PostgreSQL. Elles sont en général destinées à des cas d'utilisation très spécifiques. Leur code est souvent fermé et nécessite l'acquisition d'une licence payante.

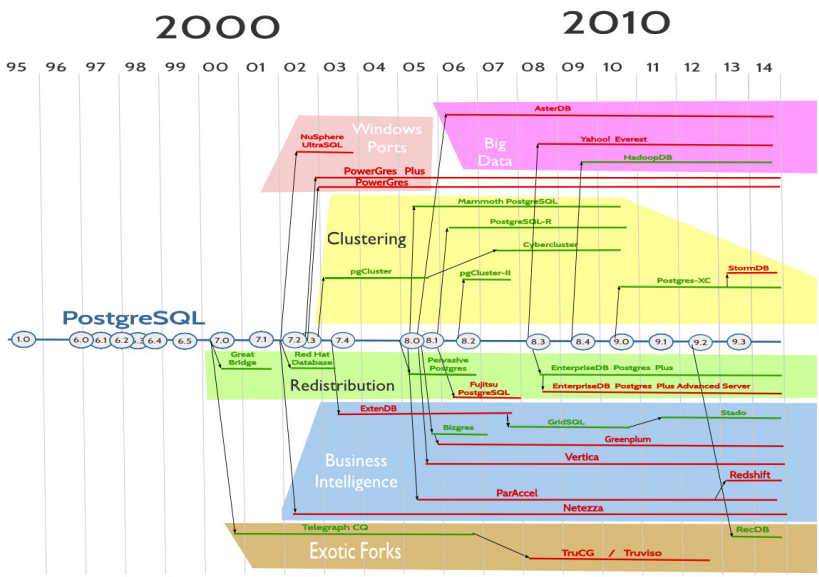
Liste exhaustive des « forks »[38] .

Sauf cas très précis, il est recommandé d'utiliser la version officielle, libre et gratuite.

---

[37] http://www.postgresql.org/about/featurematrix
[38] http://wiki.postgresql.org/wiki/PostgreSQL_derived_databases

2.3.16 HISTORIQUE DES VERSIONS DÉRIVÉES



Voici un schéma des différentes versions de PostgreSQL ainsi que des versions dérivées. Cela montre principalement l'arrivée annuelle d'une nouvelle version majeure, ainsi que de la faible résistance des versions dérivées. La majorité n'a pas survécu à la vitalité du développement de PostgreSQL.

2.4 CONCEPTS DE BASE

- ACID
- MVCC
- Transactions
- Journaux de transactions

2.4.1 ACID

- **Atomicité** (Atomic)

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

Les propriétés ACID sont le fondement même de tout système transactionnel. Il s'agit de quatre règles fondamentales :

- A : Une transaction est entière : « tout ou rien ».
- C : Une transaction amène le système d'un état stable à un autre.
- I : Les transactions n'agissent pas les unes sur les autres.
- D : Une transaction validée provoque des changements permanents.

Les propriétés ACID sont quatre propriétés essentielles d'un sous-système de traitement de transactions d'un système de gestion de base de données. Certains SGBD ne fournissent pas les garanties ACID. C'est le cas de la plupart des SGBD non-relationnels (« NoSQL »). Cependant, la plupart des applications ont besoin de telles garanties et la décision d'utiliser un système ne garantissant pas ces propriétés ne doit pas être prise à la légère.

2.4.2 MULTIVERSION CONCURRENCY CONTROL (MVCC)

- Le « noyau » de PostgreSQL
- Garantit ACID
- Permet les écritures concurrentes sur la même table

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.

C'est notamment MVCC qui permet de sauvegarder facilement une base *à chaud* et d'obtenir une sauvegarde cohérente 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, un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

2.4.3 MVCC ET LES VERROUS

- 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**.

MVCC maintient toutes les versions nécessaires de chaque tuple, 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.

Voici un exemple concret :

```
# SELECT now();
              now
-------------------------------
 2017-08-23 16:28:13.679663+02
(1 row)

# BEGIN;
BEGIN
# SELECT now();
             now
-------------------------------
 2017-08-23 16:28:34.888728+02
(1 row)

# SELECT pg_sleep(2);
 pg_sleep
----------

(1 row)

# SELECT now();
             now
-------------------------------
 2017-08-23 16:28:34.888728+02
(1 row)
```

2.4.4 TRANSACTIONS

- Intimement liées à ACID et MVCC :
- Une transaction est un ensemble d'opérations atomique
- Le résultat d'une transaction est « tout ou rien »
- SAVEPOINT disponible pour sauvegarde des modifications d'une transaction à un instant t

Voici un exemple de transaction:

```
=> BEGIN;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1, 'Haddock', 35);
INSERT 0 1
=> SELECT age FROM capitaines;
 age
-----
  35
(1 ligne)

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

On voit que la table capitaine 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.

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;
BEGIN
=> CREATE TABLE capitaines (id serial, nom text, age integer);
CREATE TABLE
=> INSERT INTO capitaines VALUES (1,'Haddock',35);
INSERT 0 1
=> SAVEPOINT insert_sp;
SAVEPOINT
=> UPDATE capitaines SET age=45 WHERE nom='Haddock';
```

```
UPDATE 1
=> ROLLBACK TO SAVEPOINT insert_sp;
ROLLBACK
=> COMMIT;
COMMIT
=> SELECT age FROM capitaines WHERE nom='Haddock';
 age
-----
  35
(1 row)
```

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`.

2.4.5 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
- On peut spécifier le niveau d'isolation au démarrage d'une transaction:
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited`
  - `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.

2.4.6 WRITE AHEAD LOGS, AKA WAL

- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Comparable à la journalisation des systèmes de fichiers

17.12

Les journaux de transactions (appelés parfois WAL ou 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 à l'emplacement final 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'information[39] .

2.4.7 AVANTAGES DES WAL

- Un seul *sync* sur le fichier de transactions
- Le fichier de transactions est écrit de manière séquentielle
- Les fichiers de données sont écrits de façon asynchrone
- Point In Time Recovery
- Réplication (*WAL shipping*)

Les écritures se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture. Généralement, le déplacement des têtes d'un disque est l'opération la plus coûteuse. L'éviter est un énorme avantage.

De plus, comme on n'écrit que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, à condition que le système de fichiers le supporte.

L'écriture asynchrone dans les fichiers de données 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.

---

[39] http://www.dalibo.org/glmf108_postgresql_et_ses_journaux_de_transactions

2.5 FONCTIONNALITÉS

- Développement
- Sécurité
- Le « catalogue » d'objets SQL

Depuis toujours, PostgreSQL se distingue par sa licence libre (BSD) et sa robustesse prouvée sur de nombreuses années. Mais sa grande force réside également dans le grand nombre de fonctionnalités intégrées dans le moteur du SGBD :

- L'extensibilité, avec des API très bien documentées
- Une configuration fine et solide de la sécurité des accès
- Un support excellent du standard SQL

2.5.1 FONCTIONNALITÉS : DÉVELOPPEMENT

- PostgreSQL est une plate-forme de développement !
- 15 langages de procédures stockées
- Interfaces natives pour ODBC, JDBC, C, PHP, Perl, etc.
- API ouverte et documentée
- Un nouveau langage peut être ajouté **sans recompilation** de PostgreSQL

Voici la liste non exhaustive des langages procéduraux supportés :

- PL/pgSQL
- PL/Perl
- PL/Python
- PL/Tcl
- PL/sh
- PL/R
- PL/Java
- PL/lolcode
- PL/Scheme
- PL/PHP
- PL/Ruby
- PL/Lua
- PL/pgPSM
- PL/v8 (Javascript)

PostgreSQL peut donc être utilisé comme un serveur d'applications ! Vous pouvez ainsi placer votre code au plus près des données.

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 serait préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une procédure en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir. Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage.

Les applications externes peuvent accéder aux données du serveur PostgreSQL grâce à des connecteurs. Ils peuvent passer par l'interface native, la **libpq**. C'est le cas du connecteur PHP et du connecteur Perl par exemple. Ils peuvent aussi ré-implémenter cette interface, ce que fait le pilote ODBC (psqlODBC) ou le driver JDBC.

Tableau des langages supportés[40] .

2.5.2 FONCTIONNALITÉS : EXTENSIBILITÉ

Création de types de données et

- de leurs fonctions
- de leurs opérateurs
- de leurs règles
- de leurs aggrégats
- de leurs méthodes d'indexations

Il est possible de définir de nouveaux 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 va pouvoir être utilisé dans tous les objets SQL habituels : table, procédure stockée, opérateur (pour redéfinir l'opérateur + par exemple), procédure d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
```

[40] http://wiki.postgresql.org/wiki/PL_Matrix

```
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

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

Conférence de Heikki Linakangas sur la création d'un type color[41] .

2.5.3 SÉCURITÉ

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

Le support des annuaires LDAP est disponible à partir de la version 8.2.

Le support de GSSAPI/SSPI est disponible à partir de la version 8.3. L' interface de programmation GSS API est un standard de l'IETF qui permet de sécuriser les services informatiques. La principale implémentation de GSSAPI est Kerberos. SSPI permet le Single Sign On sous MS Windows, de façon transparente, qu'on soit dans un domaine AD ou NTLM.

La gestion des certificats SSL est disponible à partir de la version 8.4.

Le support de Radius est disponible à partir de la version 9.0.

Le support de `SCRAM-SHA-256` est disponible à partir de la version 10.

2.5.4 RESPECT DU STANDARD SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.
- Unicode et plus de 50 encodages

[41] http://wiki.postgresql.org/images/1/11/FOSDEM2011-Writing_a_User_defined_type.pdf

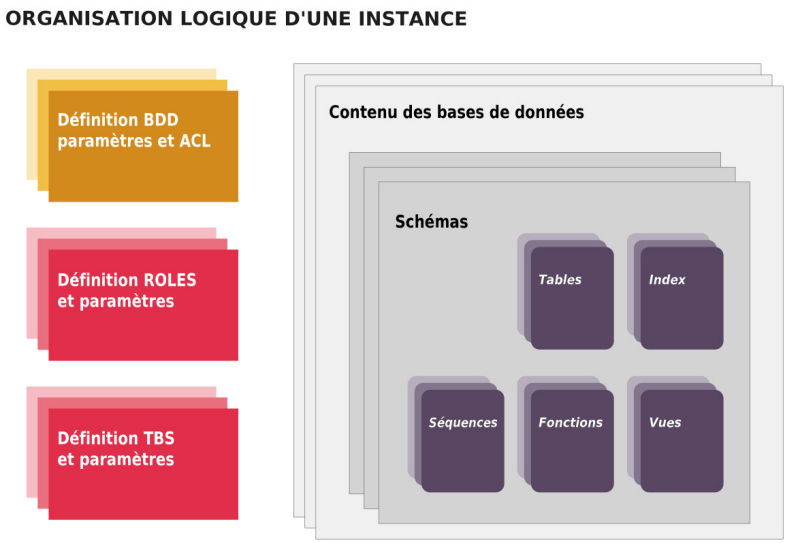
17.12

La dernière version du standard SQL est `SQL:2011`.

À ce jour, aucun SGBD ne supporte complètement `SQL:2011` *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de `SQL:2011` est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

2.5.5 ORGANISATION LOGIQUE



2.5.6 SCHÉMAS

- Espace de noms
- Concept différent des schémas d'Oracle
- Sous-ensemble de la base

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble. Tout dépend des droits dont il dispose. PostgreSQL vérifie la présence des objets par rapport au

paramètre `search_path` valable pour la session en cours lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête.

À la création d'un utilisateur, un schéma n'est pas forcément associé.

Le comportement et l'utilisation des schémas diffèrent donc d'avec Oracle.

Les schémas sont des espaces de noms dans une base de données permettant :

- de grouper 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.

Les schémas sont très utiles pour les systèmes de réplication (Slony, bucardo).

Exemple d'utilisation de schéma :

```
=> CREATE SCHEMA pirates;
CREATE SCHEMA
=> SET search_path TO pirates,public;
SET
=> CREATE TABLE capitaines (id serial, nom text);
CREATE TABLE

=> INSERT INTO capitaines (nom)
 VALUES ('Anne Bonny'), ('Francis Drake');
INSERT 0 2

-- Sélections des capitaines... pirates
=> SELECT * FROM capitaines;
 id |      nom
----+---------------
  1 | Anne Bonny
  2 | Francis Drake
(2 rows)

=> CREATE SCHEMA corsaires;
CREATE SCHEMA
=> SET search_path TO corsaires,pirates,public;
SET
=> CREATE TABLE capitaines (id serial, nom text);
CREATE TABLE
=> INSERT INTO corsaires.capitaines (nom)
 VALUES ('Robert Surcouf'), ('Francis Drake');
INSERT 0 2
-- Sélections des capitaines... français
=> SELECT * FROM capitaines;
```

```
 id |      nom
----+---------------
  1 | Robert Surcouf
  2 | Francis Drake
(2 rows)


-- Quels capitaine portant un nom identique
-- fut à la fois pirate et corsaire ?
=> SELECT pirates.capitaines.nom
  FROM pirates.capitaines,corsaires.capitaines
  WHERE pirates.capitaines.nom=corsaires.capitaines.nom;
      nom
---------------
 Francis Drake
(1 row)
```

2.5.7 VUES

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Améliorations en 9.3 et 9.4
  - vues matérialisées
  - vues automatiquement modifiables

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 en tant que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Exemple:

```
=# SET search_path TO public;
SET

-- 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;
CREATE ROLE
```

```
-- création de la table, et ajout de données
=# ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
ALTER TABLE
=# INSERT INTO capitaines (nom,age,num_cartecredit)
           VALUES ('Robert Surcouf',20,'1234567890123456');
INSERT 0 1

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

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

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

-- 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******
(1 ligne)

=> SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

À partir de la 8.4, 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 pouvait être fastidieux).

Par exemple :

```
=> SET ROLE postgres;
SET
=# 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;
CREATE VIEW
=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
      nom       | age |    num_cc_anon    |            num_md5_cc
```

```
----------------+-----+-----------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
(1 row)
```

La version 9.3 améliore encore les choses en permettant la modification des vues simples, sans ajout de code et de trigger. Ce n'est pas le cas ici étant donné que la vue retourne des colonnes calculées :

```
=# UPDATE capitaines_anon SET nom='Nicolas Surcouf' WHERE nom='Robert Surcouf';
ERROR:  cannot update view "capitaines_anon"
DETAIL:  Views that return columns that are not columns of their base relation
         are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an
       unconditional ON UPDATE DO INSTEAD rule.
```

À partir de la version 9.4, ce type de modification est possible, à condition que seules les colonnes identiques à la table sous-jacente soient mises à jour :

```
=# UPDATE capitaines_anon SET nom='Nicolas Surcouf' WHERE nom='Robert Surcouf';
UPDATE 1

=# SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';
       nom       | age |   num_cc_anon   |            num_md5_cc
-----------------+-----+-----------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
(1 row)

=# UPDATE capitaines_anon SET num_cc_anon='123456789xxxxxx'
  WHERE nom='Nicolas Surcouf';
ERREUR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

La version 9.3 ajoute aussi le support natif des vues matérialisées. Elles ne sont pas mises à jour automatiquement, il faut utiliser une instruction spécifique quand le rafraîchissement s'avère nécessaire. Ce rafraîchissement bloque les écritures et les lectures, et est un rafraîchissement complet de la vue matérialisée. Ces quelques inconvénients sont corrigés en 9.4, qui apporte la possibilité d'effectuer le rafraîchissement de la vue matérialisée de façon concurrente, c'est-à-dire sans bloquer les autres sessions désirant lire le contenu de la vue.

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

-- 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;
SELECT 2

-- 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******
(1 row)

-- 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';
UPDATE 1
=# SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';
 id |      nom       | age | num_cartecredit
----+----------------+-----+------------------
  1 | Robert Surcouf |  20 | 1234567890123456
(1 row)

=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******
(1 row)

-- Après un rafraichissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
=# REFRESH MATERIALIZED VIEW capitaines_anon;
REFRESH MATERIALIZED VIEW
=# SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
      nom       | age |   num_cc_anon
----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******
(1 row)

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions
-- ( >= 9.4 ) :
=# 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 unique pour faire un rafraichissement
```

```
-- sans bloquer les autres sessions.
=# CREATE UNIQUE INDEX ON capitaines_anon(nom);
CREATE INDEX
=# REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
REFRESH MATERIALIZED VIEW
```

Avant la version 9.3, il était possible de simuler une vue matérialisée via une table standard, une procédure stockée de type trigger et un trigger. Pour plus de détails, voir cet article de Jonathan Gardner[42] .

2.5.8 INDEX

Les algorithmes suivants sont supportés :

- `B-tree` (par défaut)
- `GiST` / `SP-GiST`
- `Hash`
- `GIN` (version `8.2`)
- `BRIN` (version `9.5`)

Attention aux index `hash` ! Avant la version 10, leur modification n'est pas enregistrée dans les journaux de transactions, ce qui amène deux problèmes. En cas de crash du serveur, il est fréquemment nécessaire de les reconstruire (`REINDEX`). De plus, ils ne sont pas restaurés avec PITR et donc avec le *Log Shipping* et le *Streaming Replication*. Par ailleurs, toujours avant la version 10, ils ne sont que rarement plus performants que les index B-Tree.

Pour une indexation standard , on utilise en général un index Btree.

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 peuvent être utiles pour les grands volumes de données fortement corrélées par rapport à leur emplacement physique sur les disques.

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

Plus d'informations :

- Article Wikipédia sur les arbres B[43]

[42] http://jonathangardner.net/PostgreSQL/materialized_views/matviews.html
[43] http://fr.wikipedia.org/wiki/Arbre_B

- Article Wikipédia sur les tables de hachage[44]
- Documentation officielle française[45]

2.5.9 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, au delà du type de données.

Elles donnent des informations au planificateur qui lui permettent d'optimiser les requêtes. Par exemple, le planificateur de la version 9.0 sait ne pas prendre en compte une jointure dans certains cas, notamment grâce à l' existence d'une contrainte unique.

Les contraintes d'exclusion ont été ajoutées en 9.0. Elles permettent un test sur plusieurs colonnes avec différents opérateurs (et non pas que l'égalité dans le cas d'une contrainte unique, qui est après tout une contrainte d' exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

2.5.10 DOMAINES

- Types créés par les utilisateurs
- Permettent de créer un nouveau type à partir d'un type de base
- En lui ajoutant des contraintes supplémentaires.

Exemple:

```
=> CREATE DOMAIN code_postal_francais AS text check (value ~ '^\d{5}$');
CREATE DOMAIN
=> ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
ALTER TABLE
=> UPDATE capitaines SET cp='35400' WHERE nom LIKE '%Surcouf';
INSERT 0 1
```

[44] http://fr.wikipedia.org/wiki/Table_de_hachage
[45] http://docs.postgresql.fr/current/textsearch-indexes.html

17.12

```
=> UPDATE capitaines SET cp='1420' WHERE nom LIKE 'Haddock';
ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"
```

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, on peut 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.

2.5.11 ENUMS

- Types créés par les utilisateurs
- Permettent de définir une liste ordonnée de valeurs de type chaîne de caractère pour ce type

Exemple :

```
=> CREATE TYPE jour_semaine
  AS ENUM ('Lundi','Mardi','Mercredi','Jeudi','Vendredi',
  'Samedi','Dimanche');
CREATE TYPE
=> ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;
CREATE TABLE
=> UPDATE capitaines SET jour_sortie='Mardi' WHERE nom LIKE '%Surcouf';
UPDATE 1
=> UPDATE capitaines SET jour_sortie='Samedi' WHERE nom LIKE 'Haddock';
UPDATE 1
=> SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
 id |   nom    | age | num_cartecredit | cp | jour_sortie
----+----------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
(1 rows)
```

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, on aurait pu créer une table `jour_de_la_semaine`, et stocker la clé

associée dans `planning`. On aurait pu tout aussi bien positionner une contrainte `CHECK`, mais on n'aurait plus eu une liste ordonnée.

2.5.12 TRIGGERS

- Opérations: `INSERT`, `COPY`, `UPDATE`, `DELETE`
- 8.4, trigger `TRUNCATE`
- 9.0, trigger pour une colonne, et/ou avec condition
- 9.1, trigger sur vue
- 9.3, trigger DDL
- 10, tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)

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 nous donnent une vision des lignes avant et après modification.

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

Exemple :

```
=> ALTER TABLE capitaines ADD COLUMN salaire integer;
ALTER TABLE

=> CREATE FUNCTION verif_salaire()
  RETURNS trigger AS $verif_salaire$
  BEGIN
  -- On verifie 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 FUNCTION

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

=> UPDATE capitaines SET salaire=2000 WHERE nom='Haddock';
UPDATE 1
=> UPDATE capitaines SET salaire=3000 WHERE nom='Haddock';
UPDATE 1
=> UPDATE capitaines SET salaire=2000 WHERE nom='Haddock';
ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

2.6 SPONSORS & RÉFÉRENCES

- Sponsors
- Références
  - françaises
  - et internationales

Au delà de ses qualités, PostgreSQL suscite toujours les mêmes questions récurrentes :

- qui finance les développements ? (et pourquoi ?)
- qui utilise PostgreSQL ?

2.6.1 SPONSORS

- NTT (Streaming Replication)
- Crunchy Data Solutions (Tom Lane, Stephen Frost, Joe Conway, Greg Smith)
- Microsoft Skype Division (projet skytools)
- EnterpriseDB (Bruce Momjian, Dave Page...)
- 2nd Quadrant (Simon Riggs...)
- VMWare (Heikki Linnakangas)

- Dalibo
- Fujitsu
- Red Hat
- Sun Microsystems (avant le rachat par Oracle)

À partir de juin 2006, le système d'exploitation Unix Solaris 10 embarque PostgreSQL dans sa distribution de base, comme base de données de référence pour ce système d'exploitation. Ce n'est plus le cas avec la sortie en 2011 de Oracle Solaris 11.

Le rachat de MySQL par Sun Microsystems ne constitue pas un danger pour PostgreSQL. Au contraire, Sun a rappelé son attachement et son implication dans le projet.

- News SUN[46]
- Article[47]

Le rachat de Sun Microsystems par Oracle ne constitue pas non plus un danger, bien qu'évidemment les ressources consacrées à PostgreSQL, autant humaines que matérielles, ont toutes été annulées.

NTT finance un groupe de développeurs sur PostgreSQL, ce qui lui a permis de fournir de nombreux patchs pour PostgreSQL, le dernier en date concernant un système de réplication interne au moteur. Ce système a été inclus dans la version de la communauté depuis la 9.0. Plus d'informations[48] .

Ils travaillent à un outil de surveillance de bases PostgreSQL assez poussé qu'ils ont présenté lors de PGCon 2010.

Fujitsu a participé à de nombreux développements aux débuts de PostgreSQL.

Red Hat a longtemps employé Tom Lane à plein temps pour travailler sur PostgreSQL. Il a pu dédier une très grande partie de son temps de travail à ce projet, bien qu'il ait eu d'autres affectations au sein de Red Hat. Il maintient quelques paquets RPM, dont ceux du SGBD PostgreSQL. Il assure une maintenance sur leur anciennes versions pour les distributions Red Hat à grande durée de vie. Tom Lane a travaillé également chez SalesForce, ensuite il a rejoint Crunchy Data Solutions fin 2015.

Skype est apparu il y a plusieurs années maintenant. Ils proposent un certain nombre d'outils très intéressants : PgBouncer (pooler de connexion), Londiste (réplication par trigger), etc. Ce sont des outils qu'ils utilisent en interne et qu'ils publient sous licence BSD comme retour à la communauté. Le rachat par Microsoft n'a pas affecté le développement de ces outils.

[46] http://www.news.com/Sun-backs-open-source-database-PostgreSQL/2100-1014_3-5958850.html
[47] http://www.lemondeinformatique.fr/actualites/lire-sun-encourage-a-essayer-la-version-83-de-postgresql-25253.html
[48] http://wiki.postgresql.org/wiki/Streaming_Replication

17.12

EnterpriseDB est une société anglaise qui a décidé de fournir une version de PostgreSQL propriétaire fournissant une couche de compatibilité avec Oracle. Ils emploient plusieurs codeurs importants du projet PostgreSQL (dont deux font partie de la « Core Team »), et reversent un certain nombre de leurs travaux au sein du moteur communautaire. Ils ont aussi un poids financier qui leur permet de sponsoriser la majorité des grands événements autour de PostgreSQL : PGEast et PGWest aux États-Unis, PGDay en Europe.

Dalibo participe pleinement à la communauté. La société est sponsor platinum du projet PostgreSQL[49] . Elle développe et maintient plusieurs outils plébiscités par la communauté, comme par exemple pgBadger ou Ora2Pg, avec de nombreux autres projets en cours, et une participation active au développement de patchs pour PostgreSQL. Elle sponsorise également des événements comme les PGDay français et européens, ainsi que la communauté francophone. Plus d'informations[50] .

2.6.2 RÉFÉRENCES

- Yahoo
- Météo France
- RATP
- CNAF
- Le Bon Coin
- Instagram
- Zalando
- TripAdvisor

Le DBA de TripAdvisor témoigne de leur utilisation de PostgreSQL dans l'interview suivante[51] .

2.6.3 LE BON COIN

Site de petites annonces :

- Base transactionnelle de 6 To
- 4è site le plus consulté en France (2017)
- 800 000 nouvelles annonces par jour

[49] http://www.postgresql.org/about/sponsors/
[50] http://www.dalibo.org/contributions
[51] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley

- 4 serveurs PostgreSQL en réplication
  - 160 cœurs par serveur
  - 2 To de RAM
  - 10 To de stockage flash

PostgreSQL tient la charge sur de grosses bases de données et des serveurs de grande taille.

Voir les témoignages de ses directeurs technique[52] (témoignage de juin 2012) et infrastructure[53] (juin 2017) pour plus de détails sur la configuration.

2.7 CONCLUSION

- Un projet de grande ampleur
- Un SGBD complet
- Souplesse, extensibilité
- De belles références
- Une solution **stable**, **ouverte**, **performante** et **éprouvée**

Certes, la licence PostgreSQL implique un coût nul (pour l'acquisition de la licence), un code source disponible et aucune contrainte de redistribution. Toutefois, il serait erroné de réduire le succès de PostgreSQL à sa gratuité.

Beaucoup d'acteurs font le choix de leur SGBD sans se soucier de son prix. En l'occurrence, ce sont souvent les qualités intrinsèques de PostgreSQL qui séduisent :

- sécurité des données (reprise en cas de crash et résistance aux bogues applicatifs) ;
- facilité de configuration ;
- montée en puissance et en charge progressive ;
- gestion des gros volumes de données.

2.7.1 BIBLIOGRAPHIE

- Documentation officielle (préface)
- Articles fondateurs de M. Stonebracker
- Présentation du projet PostgreSQL

[52] http://www.postgresqlfr.org/temoignages:le_bon_coin
[53] https://www.kissmyfrogs.com/jean-louis-bergamo-leboncoin-ce-qui-a-ete-fait-maison-est-ultra-performant/

17.12

« Préface : 2. Bref historique de PostgreSQL[54] ». PGDG, 2013

« The POSTGRES™ data model[55] ». Rowe and Stonebraker, 1987

« Présentation du projet PostgreSQL[56] ». Guillaume Lelarge, 2008

Iconographie :

La photo initiale est le logo officiel de PostgreSQL[57] .

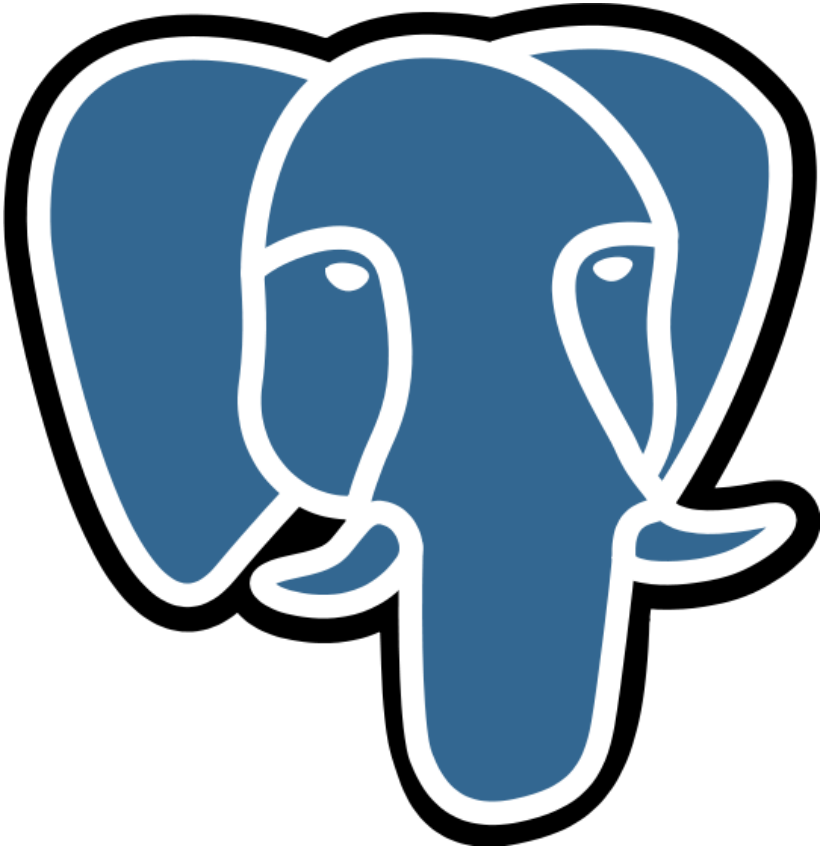
2.7.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[54] http://docs.postgresqlfr.org/9.4/history.html
[55] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[56] http://www.dalibo.org/presentation_du_projet_postgresql
[57] http://wiki.postgresql.org/wiki/Trademark_Policy

3 PL/PGSQL : LES BASES



3.1 PRÉAMBULE

- Vous apprendrez :
  - À choisir si vous voulez écrire du PL
  - À choisir votre langage PL
  - Les principes généraux des langages PL autres que PL/PgSQL
  - Les bases de PL/PgSQL

17.12

Ce module présente la programmation PL/PgSQL. Il commence par décrire les procédures stockées et les différents langages disponibles. Puis il aborde les bases du langage PL/pgsql, autrement dit :

- comment installer PL/PgSQL dans une base PostgreSQL ;
- comment créer un squelette de fonction ;
- comment déclarer des variables ;
- comment utiliser les instructions de base du langage ;
- comment créer et manipuler des structures ;
- comment passer une valeur de retour de la fonction à l'appelant.

3.1.1 AU MENU

- Présentation du PL et des principes
- Présentations de PL/pgSQL et des autres langages PL
- Installation d'un langage PL
- Détails sur PL/pgSQL

3.1.2 OBJECTIFS

- Comprendre les cas d'utilisation d'une fonction PL/PgSQL
- Choisir son langage PL en connaissance de cause
- Comprendre la différence entre PL/PgSQL et les autres langages PL
- Écrire une fonction simple en PL/PgSQL

3.2 INTRODUCTION

3.2.1 INTRODUCTION AUX PL - 1

- PL = Procedural Languages
- 3 langages activés par défaut : C, SQL et PL/PgSQL
  - PL/PgSQL n'était pas activé par défaut avant la 9.0

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

Par défaut, trois langages sont installés et activés : C, SQL et PL/pgsql. Ce dernier n'est activé par défaut que depuis la version 9.0.

3.2.2 INTRODUCTION AUX PL - 2

- Nombreux langages disponibles
- PL/Tcl, PL/Perl, PL/PerlU, PL/python, PL/php, PL/java, PL/mono, etc.

Une quinzaine de langages sont disponibles, ce qui fait que la plupart des langages connus sont couverts. De plus, il est possible d'en ajouter d'autres.

3.2.3 INTRODUCTION AUX PL - 3

- Différence entre les langages de confiance (trusted) et les autres
- Langage de confiance
  - ne permet que l'accès à la base de données
  - donc pas d'accès aux systèmes de fichiers, aux sockets réseaux, etc.
- Trusted : PL/PgSQL, SQL, PL/Perl, PL/Python…
- Untrusted : PL/PerlU, C…

Les langages de confiance ne peuvent qu'accèder à la base de données. Ils ne peuvent pas accéder aux autres bases, aux systèmes de fichiers, au réseau, etc. Ils sont donc confinés, ce qui les rend moins facilement utilisable pour compromettre le système. PL/PgSQL est l'exemple typique. Mais du coup, ils offrent moins de possibilités que les autres langages.

Seuls les superutilisateurs peuvent créer une fonction dans un langage Untrusted. Par contre, ils peuvent ensuite donner les droits d'exécution à ces fonctions aux autres utilisateurs.

3.2.4 LES LANGAGES PL DE POSTGRESQL

Les langages PL fournissent :

- Des fonctionnalités procédurales dans un univers relationnel
- Des fonctionnalités avancées du langage PL choisi

- Des performances de traitement souvent supérieures à celles du même code côté client

Il peut y avoir de nombreuses raisons différentes à l'utilisation d'un langage PL. 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 mise en place d'identifiants pour liens entre ces tables, peut évidemment être écrite côté client. Il est quelquefois plus pratique de l'écrire sous forme de PL. On y gagne :

- La centralisation du code : si plusieurs applications ont potentiellement besoin d'écrire le traitement, cela réduit d'autant les risques de bugs
- Les 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 message réseaux dus à l'exécution de nombreux ordres SQL consécutifs
- La 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. Si vous avez besoin de réaliser du traitement de chaîne puissant, ou de la manipulation de fichiers, PL/Perl ou PL/Python seront probablement des options plus intéressantes, car plus performantes.

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.

Les langages PL permettent donc de rajouter une couche d'abstraction et d'effectuer des traitements avancés directement en base.

3.2.5 INTÉRÊTS DE PL/PGSQL EN PARTICULIER

- Structure inspirée de l'ADA, donc proche du Pascal
- Ajout de structures de contrôle au langage SQL
- Peut effectuer des traitements complexes
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- Est «Trusted»
- Et facile à utiliser

La plupart des gens ont eu l'occasion de faire du Pascal ou de l'ADA, et sont donc familiers avec la syntaxe de PL/PgSQL. Cette syntaxe est d'ailleurs très proche de celle de PLSQL

d'Oracle.

Elle permet d'écrire des requêtes 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 de SQL qui est procédural.

PL/PgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux que vous aurez rajouté. Il peut les manipuler de façon transparente.

PL/PgSQL est trusted. Tous les utilisateurs peuvent donc créer des procédures dans ce langage (par défaut). Vous pouvez toujours soit supprimer le langage, 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, peu de pièges, il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

3.2.6 LES AUTRES LANGAGES PL ONT TOUJOURS LEUR INTÉRÊT

- Avantages des autres langages PL par rapport à PL/PgSQL :
  - Beaucoup plus de possibilités
  - Souvent plus performants pour la résolution de certains problèmes
- Mais un gros défaut :
  - Pas spécialisés dans le traitement de requêtes

Les langages PL «autres», comme PL/Perl et PL/Python (les deux plus utilisés après PL/PgSQL), sont bien plus évolués que PL/PgSQL. Par exemple, ils sont bien plus efficaces en matière de traitement de chaînes de caractères, ils possèdent des structures avancées comme des tables de hachages, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leurs versions untrusted, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un Webservice 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. 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.

Il existe aussi un langage qui est, du moins sur le papier, plus rapide que tous les langages cités précédemment : vous pouvez écrire des procédures stockées en C, directement. Elles seront compilées à l'extérieur de PosgreSQL, en respectant un certain formalisme, puis seront chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour. Attention, toute erreur dans votre 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 ne faire ceci qu'en dernière extrémité.

Le gros défaut est simple et commun à tous ces langages : vous utilisez par exemple PL/Perl. Perl n'est pas spécialement conçu pour s'exécuter en tant que langage de procédures stockées. Ce que vous utilisez quand vous écrivez du PL/Perl est donc du code Perl, avec quelques fonctions supplémentaires (préfixées par spi) pour accéder à la base de données. L'accès aux données est donc rapide, mais assez malaisé au niveau syntaxique, comparé à PL/PgSQL.

Un autre problème des langages PL (autre que C et PL/PgSQL), c'est que ces langages n'ont pas les mêmes types natifs que PostgreSQL, et s'exécutent dans un interpréteur relativement séparé. Les performances sont donc moindres que PL/PgSQL et C pour les traitements dont le plus consommateur est l'accès des données. Souvent, le temps de traitement dans un de ces langages plus évolués est tout de même meilleur grâce au temps gagné par les autres fonctionnalités (la possibilité d'utiliser un cache, ou une table de hachage par exemple).

3.3 INSTALLATION

- PL/pgsql compilé et installé par défaut
- Autres langages à compiler explicitement
- Paquets Debian et RedHat
  - PL/PgSQL par défaut
  - PL/Perl et PL/Python souvent
- L'installeur Windows contient aussi PL/PgSQL

L'installeur Windows contient aussi PL/PgSQL. PL/Perl et PL/Python sont par contre plus compliqués de mise en œuvre. Il faut disposer d'exactement la version qui a été utilisée par le packageur au moment de la compilation de la version windows.

Pour savoir si PL/Perl ou PL/Python a été compilé, on peut à nouveau demander à `pg_config` :

```
> pg_config --configure
'--prefix=/usr/local/pgsql-10_icu' '--enable-thread-safety'
'--with-openssl' '--with-libxml' '--enable-nls' '--with-perl' '--enable-debug'
'ICU_CFLAGS=-I/usr/local/include/unicode/'
'ICU_LIBS=-L/usr/local/lib -licui18n -licuuc -licudata' '--with-icu'
```

3.3.1 VÉRIFICATION DE LA DISPONIBILITÉ

Comment vérifier la présence de la bibliothèque :

```
find $(pg_config --libdir) -name "plpgsql.so"
```

```
find $(pg_config --pkglibdir) -name "plpgsql.so"
```

La bibliothèque `plpgsql.so` contient les fonctions qui permettent l'utilisation du langage PL/pgsql. Elle est installée par défaut avec le moteur PostgreSQL.

Elle est chargée par le moteur à la première utilisation d'une procédure utilisant ce langage.

Toutefois, il est encore plus simple de demander à PostgreSQL d'activer le langage. Il sera bien temps ensuite, si cela échoue, de lancer cette commande de diagnostic.

3.3.2 AJOUT DU LANGAGE

- Activé par défaut depuis PostgreSQL 9.0
- Commande similaire pour les autres langages
- Activer :

  ```
  CREATE EXTENSION plpgsql;
  ```

- Désactiver :

  ```
  DROP EXTENSION plpgsql;
  ```

Activer le langage dans la base modèle template1 l'activera aussi pour toutes les bases créées par la suite.

PostgreSQL fournit un outil, appelé `createlang`, pour activer un langage :

```
createlang plpgsql la_base_a_activer
```

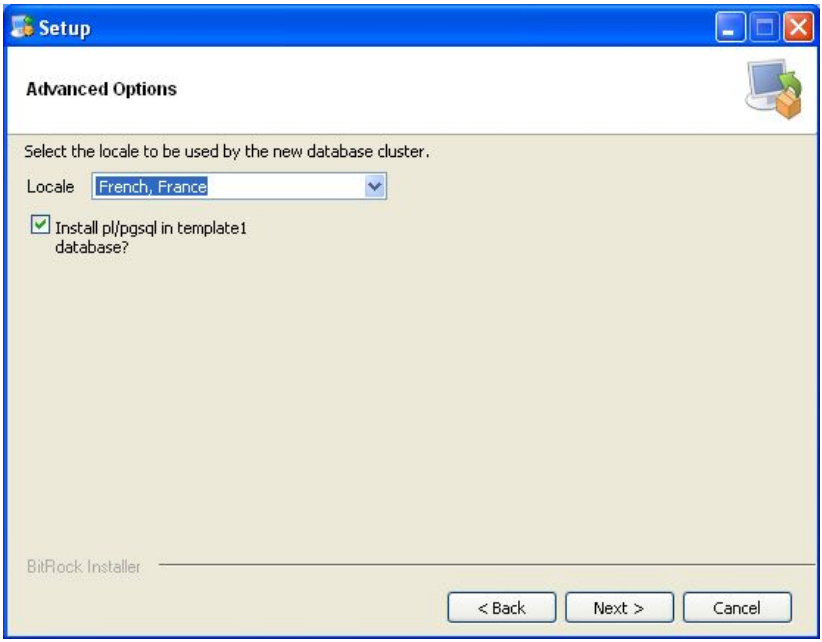
L'outil se connecte à la base indiquée et exécute la commande `CREATE LANGUAGE` pour le langage précisé en argument. Son pendant Windows se nomme `createlang.exe`. Il existe aussi un outil pour désactiver un langage (`droplang`).

Si vous optez pour exécuter vous-même l'ordre SQL, le langage est créé dans la base dans laquelle la commande est lancée.

Pour installer un autre langage, utilisez la même commande tout en remplaçant plpgsql par plperl, plperlu, plpython, pltcl, plsh…

3.3.3 AJOUT DE LANGAGE SOUS WINDOWS

- Cochez la case PL/pgsql pour que l'installeur Windows active le langage dans la base modèle template1



L'installeur Windows a beaucoup changé depuis ses premières versions. Actuellement, plus aucune question n'est posée quant à l'activation du langage PL/pgsql. Des versions antérieures permettaient son activation automatique si l'utilisateur cliquait la case à cocher affichée dans la copie d'écran ci-dessus. Des versions encore plus anciennes proposaient aussi l'installation de plperl et de plpython.

3.3.4 PL/PGSQL DÉJÀ INSTALLÉ ?

- Interroger le catalogue système `pg_language`
- Il contient une ligne par langage installé
- Un langage peut avoir lanpltrusted à false

Voici un exemple d'interrogation de `pg_language` :

```
SELECT lanname, lanpltrusted
FROM pg_language
WHERE lanname='plpgsql';

 lanname | lanpltrusted
---------+--------------
 plpgsql | t
(1 ligne)
```

Un langage est 'trusted' si tous les utilisateurs peuvent créer des procédures dans ce langage. Sinon seuls les super-utilisateurs le peuvent. Il existe par exemple deux variantes de PL/Perl : PL/Perl et PL/PerlU. La seconde est la variante untrusted et est un Perl «complet». La version trusted n'a pas le droit d'ouvrir des fichiers, des sockets, ou autres appels systèmes qui seraient dangereux.

SQL et PL/PgSQL sont trusted, C est untrusted.

3.3.5 UN LANGAGE PL EST DÉJÀ INSTALLÉ ?

- vérification dans psql avec la commande `\dx`
- pgAdmin affiche aussi cette information à partir du nœud « Langages » de la base de données
- phpPgAdmin le permet aussi

Depuis PostgreSQL 9.1, le langage PL/pgsql apparaît comme une extension :

```
base=# \dx
                      Liste des extensions installées
     Nom     | Version |   Schéma   |            Description
-------------+---------+------------+---------------------------------------
 plpgsql     | 1.0     | pg_catalog | PL/pgSQL procedural language
```

C'est évidemment aussi applicable aux autres langages PL.

À partir de la version 1.8 de pgAdmin, il est nécessaire d'autoriser son affichage (menu Fichier/Préférences, onglet Affichage). Vous pourriez avoir besoin de redémarrer pgAdmin pour que la modification soit prise en compte au niveau de l'affichage.

3.4 CRÉATION ET STRUCTURE

- Ordre SQL : `CREATE FUNCTION`
- Pas de procédure

- Le langage est un paramètre comme un autre

Voici la syntaxe complète :

```
Syntax:
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
           [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Et un exemple pour PL/pgsql :

```
CREATE FUNCTION ma_fonction () RETURNS integer
LANGUAGE plpgsql
...
```

Il est à noter que PostgreSQL n'accepte que des fonctions. Si vous voulez créer une procédure, cela revient à créer une fonction qui ne renvoit rien. Il faut utiliser le type `void` dans ce cas.

Attention, c'est un des domaines où il y a le plus de nouvelles fonctionnalités ajoutées : vérifiez bien que votre code est compatible ascendant avec toutes les versions que vous allez devoir supporter

---

3.4.1 ARGUMENTS

- Préciser les arguments :

```
[ [ mode_argument ] [ nom_argument ] type_argument
[ { DEFAULT | = } expr_defaut ] [, ...] ]
```

- mode_argument : en entrée (IN), en sortie (OUT), en entrée/sortie (INOUT) ou à nombre variant (VARIADIC)
- nom_argument : nom (libre et optionnel)
- type_argument : type (parmi tous les types de base et les types utilisateur)
- valeur par défaut : clause DEFAULT

Si le mode de l'argument est omis, `IN` est la valeur implicite.

L'option `VARIADIC` n'est disponible qu'à partir de la version 8.4. Cette option permet de définir une fonction avec un nombre d'arguments libres à condition de respecter le type de l'argument (comme `printf` en C par exemple). Seul un argument `OUT` peut suivre un argument `VARIADIC` : l'argument `VARIADIC` doit être le dernier de la liste des paramètres en entrée puisque tous les paramètres en entrée suivant seront considérées comme faisant partie du tableau variadic. Seuls les arguments `IN` et `VARIADIC` sont utilisables avec une fonction déclarée renvoyant une table (clause `RETURNS TABLE`). S'il y a plusieurs paramètres en `OUT`, un enregistrement composite de tous ces types est renvoyé (c'est donc équivalent sémantiquement à un `RETURNS TABLE`).

La clause `DEFAULT` permet de rendre les paramètres optionnels. Après le premier paramètre ayant une valeur par défaut, tous les paramètres qui suivent doivent avoir une valeur par défaut. Pour rendre le paramètre optionnel, il doit être le dernier argument ou alors les paramètres suivants doivent aussi avoir une valeur par défaut.

3.4.2 CRÉATION D'UNE FONCTION - 2

- Il faut aussi indiquer un type de retour :

  ```
  RETURNS type_ret
  ```

- sauf si un ou plusieurs paramètres sont en mode OUT ou INOUT
- type_ret : type de la valeur en retour (parmi tous les types de base et les types utilisateurs)
- void est un type de retour valide
- Il est aussi possible d'indiquer un type table
- Peut renvoyer plusieurs lignes : clause `SETOF`

Voici comment s'utilise le retour de type table :

```
RETURNS TABLE ( nom_colonne nom_type [, ...] )
```

17.12

On peut aussi indiquer que la fonction ne retourne pas un enregistrement (un scalaire en termes relationnels) mais un jeu d'enregistrements (c'est-à-dire une relation, une table). Il faut utiliser le mot clé `SETOF`

3.4.3 CRÉATION D'UNE FONCTION - 3

- Le langage de la fonction doit être précisé :

  ```
  LANGUAGE nomlang
  ```

- Dans notre cas, nous utiliserons plpgsql.
- Mais il est possible de créer des fonctions en plphp, plruby, voire des langages spécialisés comme plproxy.

3.4.4 CRÉATION D'UNE FONCTION - 4

- Mode de la fonction :

  ```
  IMMUTABLE | STABLE | VOLATILE
  ```

- Ce mode précise la «volatilité» de la fonction.

On peut indiquer à PostgreSQL le niveau de volatilité (ou de stabilité) d'une fonction. Ceci permet d'aider PostgreSQL à optimiser les requêtes utilisant ces fonctions, mais aussi d'interdire leur utilisation dans certains contextes.

- Une fonction est IMMUTABLE (immuable) si son exécution ne dépend que de ses paramètres. Elle ne doit donc dépendre ni du contenu de la base (pas de SELECT, ni de modification de donnée de quelque sorte), ni d'**aucun** autre élément qui ne soit pas un de ses paramètres. Par exemple, `now()` n'est évidemment pas immuable. Une fonction sélectionnant des données d'une table non plus. `to_char()` n'est pas non plus immuable : son comportement dépend des paramètres de session, par exemple `to_char(timestamp with time zone, text)` dépend du paramètre de session timezone...
- Une fonction est STABLE si son exécution donne toujours le même résultat sur toute la durée d'un ordre SQL, pour les mêmes paramètres en entrée. Cela signifie que la fonction ne modifie pas les données de la base. `to_char()` est STABLE.
- Une fonction est VOLATILE dans tous les autres cas. now() est VOLATILE. Une fonction non déclarée comme STABLE ou IMMUTABLE est VOLATILE par défaut.

Quelle importance ?

- Une fonction IMMUTABLE peut être remplacée par son résultat avant même la planification d'une requête l'utilisant. L'exemple le plus simple est une simple opération arithmétique : vous exécutez

```
SELECT * FROM ma_table WHERE mon_champ> abs(-2)
```

et PostgreSQL substitue abs(-2) par 2 et planifie ensuite la requête. Cela fonctionne aussi, bien sûr, avec les opérateurs (comme +), qui ne sont qu'un habillage syntaxique au-dessus d'une fonction.

- Une fonction STABLE peut être remplacée par son résultat pendant l'exécution de la requête. On n'a par contre aucune idée de sa valeur avant de commencer à exécuter la requête parce qu'on ne sait pas encore, à ce moment là, quelles sont les données qui seront visibles en base. Par exemple, avec

```
SELECT * FROM ma_table WHERE mon_timestamp > now()
```

PostgreSQL sait que now() (le timestamp de démarrage de la transaction) va être constant pendant toute la durée de la transaction. Néanmoins, now() n'est pas immutable, il ne va donc pas le remplacer par sa valeur avant d' exécuter la requête. Il n'exécutera par contre now() qu'une seule fois.

- Une fonction volatile doit systématiquement être exécutée à chaque appel.

On comprend donc l'intérêt de se poser la question à l'écriture de chaque fonction.

Une autre importance existe, pour la création d'index sur fonction. Par exemple,

```
CREATE INDEX mon_index ON ma_table ((ma_fonction(ma_colonne))
```

Ceci n'est possible que si la fonction est IMMUTABLE. En effet, si le résultat de la fonction dépend de l'état de la base, la fonction calculée au moment de la création de la clé d'index ne retournera plus le même résultat quand viendra le moment de l'interroger. PostgreSQL n'acceptera donc que les fonctions IMMUTABLE dans la déclaration des index fonctionnels.

3.4.5 CRÉATION D'UNE FONCTION - 5

- Précision sur la façon dont la fonction gère les valeurs NULL :

  `CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT`

- `CALLED ON NULL INPUT` : fonction appelée même si certains arguments sont NULL.

- `RETURNS NULL ON NULL INPUT` ou `STRICT` : la fonction renvoie NULL à chaque fois qu'au moins un argument est NULL.

Si un des arguments est NULL, PostgreSQL n'exécute même pas la fonction et utilise NULL comme résultat.

Dans la logique relationnelle, NULL signifie «la valeur est inconnue». La plupart du temps, il est logique qu'une fonction ayant un paramètre à une valeur inconnue retourne aussi une valeur inconnue, ce qui fait que cette optimisation est très souvent pertinente.

On gagne à la fois en temps d'exécution, mais aussi en simplicité du code (il n'y a pas à gérer les cas NULL pour une fonction dans laquelle NULL ne doit jamais être injecté).

3.4.6 CRÉATION D'UNE FONCTION - 6

- Précision sur la politique de sécurité :

  ```
  [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
  ```

- Permet de déterminer l'utilisateur avec lequel sera exécutée la fonction
- Le «sudo» de la base de données
  - Potentiellement dangereux

Une fonction `SECURITY INVOKER` s'exécute avec les droits de l'appelant. C'est le mode par défaut.

Une fonction `SECURITY DEFINER` s'exécute avec les droits du créateur. Cela permet, au travers d'une fonction, de permettre à un utilisateur d'outrepasser ses droits de façon contrôlée.

Bien sûr, une fonction `SECURITY DEFINER` doit faire l'objet d'encore plus d'attention qu'une fonction normale. Elle peut facilement constituer un trou béant dans la sécurité de votre base.

Des choses importantes sont à noter pour `SECURITY DEFINER` :

- Toute fonction, par défaut, est exécutable par public. La première chose à faire est donc de révoquer ce droit.
- Il faut se protéger des variables de session qui pourraient être utilisées pour modifier le comportement de la fonction, en particulier le **search_path**. Il doit donc **impérativement** être positionné en dur dans cette fonction (soit d'emblée, avec un SET dans la fonction, soit en positionnant un SET dans le CREATE FUNCTION).

Le mot clé EXTERNAL est facultatif, et n'est là que pour être en conformité avec la norme SQL : en effet, dans PostgreSQL, on peut modifier le security definer pour toutes les fonctions, qu'elles soient externes ou pas.

3.4.7 CRÉATION D'UNE FONCTION - 7

- Précision du code à exécuter :

  ```
  AS 'definition' | AS 'fichier_obj', 'symbole_lien'
  ```

- Premier cas : chaîne « definition » contenant le code réel de la fonction
- Deuxième cas : `fichier_obj` est le nom de la bibliothèque, `symbole_lien` est le nom de la fonction dans le code source C

`symbole_lien` n'est à utiliser que quand le nom de la fonction diffère du nom de la fonction C qui l'implémente.

3.4.8 CRÉATION D'UNE FONCTION - 8

- Paramètre obsolète :

  ```
  WITH ( attribut [, ...] )
  ```

- `isStrict`, équivalent à `STRICT` ou `RETURNS NULL ON NULL INPUT`
- `isCachable`, équivalent à `IMMUTABLE`
- À remplacer par les syntaxes présentées précédemment

La clause `WITH` est présentée ici pour être complet mais il est fortement déconseillé de l'utiliser car obsolète. Elle pourrait disparaître rapidement.

3.4.9 CRÉATION D'UNE FONCTION - 9

- `COST cout_execution`
  - coût estimé pour l'exécution de la fonction
- `ROWS nb_lignes_resultat`
  - nombre estimé de lignes que la fonction renvoie

COST est représenté en unité de `cpu_operator_cost` (100 par défaut).

ROWS vaut par défaut 1000 pour les fonctions `SETOF`. Pour les autres fonctions, la valeur de ce paramètre est ignorée et remplacée par 1.

Ces deux paramètres ne modifient pas le comportement de la fonction. Ils ne servent que pour aider l'optimiseur de requête à estimer le coût d'appel à la fonction, afin de savoir, si plusieurs plans sont possibles, lequel est le moins coûteux par rapport au nombre d'appels de la fonction et au nombre d'enregistrements qu'elle retourne.

3.4.10 CRÉATION D'UNE FONCTION - 10

- `PARALLEL [UNSAFE | RESTRICTED | SAFE]`
  - la fonction peut-elle être excutée en mode parallèle

`PARALLEL UNSAFE` indique que la fonction ne peut pas être exécutée dans le mode parallèle. La présence d'une fonction de ce type dans une requête SQL force un plan d'exécution en série. C'est la valeur par défaut.

Une fonction est non parallélisable si elle modifie l'état d'une base ou si elle fait des changements sur la transaction.

`PARALLEL RESTRICTED` indique que la fonction peut être exécutée en mode parallèle mais l'exécution est restreinte au processus principal d'exécution.

Une fonction peut être déclarée comme restreinte si elle accède aux tables temporaires, à l'état de connexion des clients, aux curseurs, aux requêtes préparées.

`PARALLEL SAFE` indique que la fonction s'exécute correctement dans le mode parallèle sans restriction.

En général, si une fonction est marquée sûre ou restreinte à la parallélisation alors qu'elle ne l'est pas, elle pourrait renvoyer des erreurs ou fournir de mauvaises réponses lorsqu'elle est utilisée dans une requête parallèle.

En cas de doute, les fonctions doivent être marqiées comme UNSAFE, ce qui correspond à la valeur par défaut.

3.4.11 EXEMPLE DE FONCTION PL/PERL

- Permet d'insérer une facture associée à un client
- Si le client n'existe pas, une entrée est créée

Voici l'exemple de la fonction :

```
CREATE OR REPLACE FUNCTION
    public.demo_insert_perl(nom_client text, titre_facture text)
 RETURNS integer
 LANGUAGE plperl
 STRICT
AS $function$
  use strict;
  my ($nom_client, $titre_facture)=@_;
  my $rv;
  my $id_facture;
  my $id_client;

  # Le client existe t'il ?
  $rv = spi_exec_query('SELECT id_client FROM mes_clients WHERE nom_client = '
    . quote_literal($nom_client)
  );
  # Sinon on le crée :
  if ($rv->{processed} == 0)
  {
    $rv = spi_exec_query('INSERT INTO mes_clients (nom_client) VALUES ('
        . quote_literal($nom_client) . ') RETURNING id_client'
    );
  }
  # Dans les deux cas, l'id client est dans $rv :
  $id_client=$rv->{rows}[0]->{id_client};

  # Insérons maintenant la facture
  $rv = spi_exec_query(
    'INSERT INTO mes_factures (titre_facture, id_client) VALUES ('
    . quote_literal($titre_facture) . ", $id_client ) RETURNING id_facture"
  );

  $id_facture = $rv->{rows}[0]->{id_facture};

  return $id_facture;
$function$ ;
```

Cette fonction n'est pas parfaite, elle ne protège pas de tout. Il est tout à fait possible d'avoir une insertion concurrente entre le `SELECT` et le `INSERT` par exemple.

Il est clair que l'accès aux données est malaisé en PL/Perl, comme dans la plupart des langages, puisqu'ils ne sont pas prévus spécifiquement pour cette tâche. Par contre, on dispose de toute la puissance de Perl pour les traitements de chaîne, les appels système…

PL/Perl, c'est :

- Perl, moins les fonctions pouvant accéder à autre chose qu'à PostgreSQL (PL/PerlU pour cela)
- Un bloc de code anonyme appelé par PostgreSQL
- Des fonctions d'accès à la base, `spi_*`

3.4.12 EXEMPLE DE FONCTION PL/PGSQL

- Même fonction en PL/PgSQL
- L'accès aux données est simple et naturel
- Les types de données SQL sont natifs
- La capacité de traitement est limitée par le langage
- **Attention** au nommage des variables et paramètres

Pour éviter les conflits avec les objets de la base, il est conseillé de préfixer les variables.

```
CREATE OR REPLACE FUNCTION
public.demo_insert_plpgsql(p_nom_client text, p_titre_facture text)
 RETURNS integer
 LANGUAGE plpgsql
 STRICT
AS $function$
DECLARE
  v_id_facture int;
  v_id_client int;
BEGIN
  -- Le client existe t'il ?
  SELECT id_client
  INTO v_id_client
  FROM mes_clients
  WHERE nom_client = p_nom_client;
  -- Sinon on le crée :
  IF (NOT FOUND) THEN
    INSERT INTO mes_clients (nom_client)
    VALUES (p_nom_client)
    RETURNING id_client INTO v_id_client;
  END IF;
  -- Dans les deux cas, l'id client est dans v_id_client

  -- Insérons maintenant la facture
  INSERT INTO mes_factures (titre_facture, id_client)
  VALUES (p_titre_facture, v_id_client)
  RETURNING id_facture INTO v_id_facture;
```

```
  return v_id_facture;
END;
$function$ ;
```

3.4.13 STRUCTURE D'UNE FONCTION PL/PGSQL - 1

```
CREATE FUNCTION addition(entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2;
  RETURN resultat;
END';
```

Le langage PL/PgSQL n'est pas sensible à la casse, tout comme SQL (sauf les noms de colonnes, si vous les mettez entre des guillemets doubles).

L'opérateur de comparaison est «=», l'opérateur d'affectation «:=»

3.4.14 STRUCTURE D'UNE FONCTION - 2

- Le code de la fonction est structuré comme suit :
  - DECLARE, pour la déclaration des variables locales
  - BEGIN, pour indiquer le début du code de la fonction
  - END, pour en indiquer la fin
  - Instructions séparées par des points-virgules
  - Commentaires commençent par -- ou compris entre /* et */

3.4.15 STRUCTURE D'UNE FONCTION - 3

- Labels de bloc possibles
- Plusieurs blocs d'exception possibles dans une fonction
- Permet de préfixer des variables avec le label du bloc

- De donner un label à une boucle itérative et de préciser de quelle boucle on veut sortir, quand plusieurs d'entre elles sont imbriquées

Indiquer le nom d'un label ainsi :

```
<<mon_label>>
-- le code (blocs DECLARE, BEGIN-END, et EXCEPTION)
```

ou bien (pour une boucle)

```
[ <<mon_label>> ]
LOOP
    ordres …
END LOOP [ mon_label ];
```

Il est aussi bien sûr possible d'utiliser des labels pour des boucles `FOR`, `WHILE`, `FOREACH`.

On sort d'un bloc ou d'une boucle avec la commande `EXIT`, on peut aussi utiliser `CONTINUE` pour passer à l'exécution suivante d'une boucle sans terminer l'itération courante.

Par exemple :

```
EXIT [mon_label] WHEN compteur > 1;
```

3.4.16 MODIFICATION D'UNE FONCTION

- Dans le cas d'une modification, `CREATE OR REPLACE FUNCTION`
- Une fonction est définie par son nom et ses arguments
- Si type de retour différent, la fonction doit d'abord être supprimée puis recréée

Une fonction est surchargeable. La seule façon de les différencier est de prendre en compte les arguments (nombre et type). Deux fonctions identiques aux arguments près (on parle de prototype) ne sont pas identiques, mais bien deux fonctions distinctes.

3.4.17 SUPPRESSION D'UNE FONCTION

- Ordre SQL : `DROP FUNCTION`
- Arguments (en entrée) nécessaires à l'identification de la fonction à supprimer :

```
DROP FUNCTION addition(integer, integer);
DROP FUNCTION public.addition(integer, integer);
```

Les types des arguments en entrée doivent être indiqués. Par contre, leur nom n'a aucune importance. Vous pouvez toutefois les passer quand même mais, comme pour un `CREATE FUNCTION`, ils sont simplement ignorés.

3.4.18 UTILISATION DES GUILLEMETS

- L'utilisation des guillemets devient très rapidement complexe.
- Surtout lorsque le source se place directement entre guillemets
  - doublage de tous les guillemets du code source.
- Utilisation de `$$` à la place des guillemets qui entourent les sources.
- Ou de n'importe quel autre marqueur

Exemple :

Syntaxe utilisant uniquement des guillemets :

```
requete := requete || '' AND vin LIKE ''''bordeaux%'''' AND xyz''
```

Simplification grâce aux dollars :

```
requete := requete || $sql$ AND vin LIKE 'bordeaux%' AND xyz$sql$
```

Si vous avez besoin de mettre entre guillemets du texte qui inclut `$$`, vous pouvez utiliser `$Q$`, et ainsi de suite. Le plus simple étant de définir un marqueur de fin de fonction plus complexe…

3.5 DÉCLARATIONS

- Types natifs de PostgreSQL intégralement supportés
- Quelques types spécifiques à PL/PgSQL

En dehors des types natifs de PostgreSQL, PL/pgsql y ajoute des types spécifiques pour faciliter l'écriture des fonctions.

3.5.1 DÉCLARATION DE PARAMÈTRES

- Déclarés en dehors du code de la fonction
- Possible d'associer directement un nom au type

- Les arguments sans nom peuvent être nommés avec l'argument `ALIAS FOR` dans la partie `DECLARE`

Avant la 8.0, il n'était pas possible d'indiquer un nom aux paramètres. Le premier argument avait pour nom `$1`, le deuxième `$2`, etc.

3.5.2 DÉCLARATION DE VARIABLES

- Variables déclarées dans le source, dans la partie `DECLARE` :

```
DECLARE
nombre integer;
contenu text;
```

- Les variables peuvent se voir associées une valeur initiale :

```
nombre integer := 5;
```

3.5.3 DÉCLARATION DE CONSTANTES

- Clause supplémentaire `CONSTANT` :

```
DECLARE
valeur_fixe CONSTANT integer := 12;
version_fonction CONSTANT text := '1.12';
```

L'otpion CONSTANT permet de définir une variable pour laquelle il sera alors impossible d'assigner une valeur dans le reste de la fonction.

3.5.4 RÉCUPÉRATION D'UN TYPE

- Possible de récupérer le type d'une autre variable avec `%TYPE` :

```
quantite integer;
total quantite%TYPE
```

- Possible de récupérer le type de la colonne d'une table :

```
quantite ma_table.ma_colonne%TYPE
```

Cela permet d'écrire des fonctions plus génériques.

3.5.5 TYPE ROW - 1

- But :
  - utilisation de structures,
  - renvoi de plusieurs valeurs à partir d'une fonction
- Utiliser un type composite :

```
CREATE TYPE ma_structure AS (un_entier integer,
une_chaine text,
...);
CREATE FUNCTION ma_fonction ()
RETURNS ma_structure...;
```

3.5.6 TYPE ROW - 2

- Possible d'utiliser le type composite défini par la ligne d'une table :

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS '
DECLARE
ligne ma_table%ROWTYPE;
...
```

L'utilisation de « %ROWTYPE » permet de définir une variable qui contient la structure d'un enregistrement de la table spécifiée. « %ROWTYPE » n'est pas obligatoire, il est néanmoins préférable d'utiliser cette forme, bien plus portable. En effet, dans PostgreSQL, toute création de table créé un type associé de même nom, le nom de la table seul est donc suffisant.

3.5.7 TYPE RECORD - 1

- Identique au type ROW
  - sauf que son type n'est connu que lors de son affectation
- Une variable de type RECORD peut changer de type au cours de l'exécution de la fonction, suivant les affectations réalisées

17.12

RECORD est beaucoup utilisé pour manipuler des curseurs : cela évite de devoir se préoccuper de déclarer un type correspondant exactement aux colonnes de la requête associée à chaque curseur.

3.5.8 TYPE RECORD - 2

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS '
DECLARE
  ligne RECORD;
  ...
BEGIN
  SELECT INTO ligne * FROM ma_premiere_table;
  -- traitement de la ligne
  FOR ligne IN SELECT * FROM ma_deuxieme_table LOOP
  -- traitement de cette nouvelle ligne
  ...
```

3.5.9 MODE D'UN PARAMÈTRE : IN, OUT, INOUT

- Avant la version 8.1, paramètres en entrée uniquement
- À partir de la 8.1, trois modes de paramètres
  - `IN` : en entrée
  - `OUT` : en sortie
  - `INOUT` : en entrée et en sortie
- Fonction à plusieurs paramètres OUT : identique à une fonction qui renvoie un ROWTYPE pour un type composite créé préalablement
- Pas besoin de l'expression `RETURN` ou `RETURN NEXT` dans une fonction avec paramètre(s) `OUT`

`RETURN` est inutile avec des paramètres `OUT` parce que c'est la valeur des paramètres `OUT` à la fin de la fonction qui est retournée.

Dans le cas d'un `RETURN NEXT`, cela signifie que la fonction retourne un `SETOF` d'enregistrements. Chaque appel à `RETURN NEXT` retourne donc un enregistrement composé d'une copie de toutes les variables, au moment de l'appel à `RETURN NEXT`.

3.6 INSTRUCTIONS

- Concernent les opérations sur la base de données
  - extraction ou modification

3.6.1 AFFECTATION D'UNE VALEUR À UNE VARIABLE

- SELECT INTO :

```
SELECT INTO un_entier 5;
```

- Opérateur `:=` :

```
un_entier := 5;
un_entier := une_colonne FROM ma_table WHERE id = 5;
```

Privilégiez la seconde écriture…

3.6.2 EXÉCUTION D'UNE REQUÊTE SANS RÉSULTAT

- Exécution de la requête en direct
- Utilisation de `PERFORM` :

```
PERFORM * FROM ma_table WHERE une_colonne>0;
```

- Affectation de la variable `FOUND` si une ligne est affectée par l'instruction
- Permet aussi d'appeler une autre fonction sans en récupérer de résultat

On peut déterminer qu'aucune ligne n'a été trouvé par la requête en utilisant la variable `FOUND` :

```
PERFORM * FROM ma_table WHERE une_colonne>0;
IF NOT FOUND THEN
...
END IF;
```

Pour appeler une fonction, il suffit d'utiliser `PERFORM` de la manière suivante :

```
PERFORM mafonction(argument1);
```

3.6.3 EXÉCUTION D'UNE REQUÊTE

- Affectation de la ligne renvoyée dans une variable de type RECORD ou ROW :

  ```
  SELECT * INTO ma_variable_ligne FROM ma_table...;
  ```

- Si plusieurs enregistrements renvoyés, seul le premier est récupéré
- Pour contrôler qu'un seul enregistrement est renvoyé, remplacer INTO par INTO STRICT
- Pour récupérer plus d'un enregistrement, écrire une boucle
- L'ordre est statique : on ne peut pas faire varier les colonnes retournées, la clause WHERE, les tables...

Dans le cas du type ROW, la définition de la ligne doit correspondre parfaitement à la définition de la ligne renvoyée. Utiliser un type RECORD permet d'éviter ce type de problème. La variable obtient directement le type ROW de la ligne renvoyée.

3.6.4 FONCTION RENVOYANT UN ENSEMBLE

- Doit renvoyer un ensemble d'un type SETOF
- Chaque ligne sera récupérée par l'instruction RETURN NEXT

3.6.5 EXEMPLE D'UNE FONCTION SETOF

Exemple :

```
CREATE FUNCTION liste_entier (limite integer)
RETURNS SETOF integer
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    RETURN NEXT i;
  END LOOP;
END
$$ LANGUAGE plpgsql;
```

3.6.6 EXÉCUTION D'UNE TELLE FONCTION

Utilisation de cette requête :

```
ma_base=# SELECT * FROM liste_entier(5);

liste_entier
--------------
1
2
3
4
5
(5 lignes)
```

3.6.7 EXÉCUTION D'UNE REQUÊTE - PERFORM

- `PERFORM <query>`
- Permet l'exécution
  - d'un `INSERT`, `UPDATE`, `DELETE` (si la clause RETURNING n'est pas utilisée)
  - ou même `SELECT`, si le résultat importe peu
- Pour obtenir le nombre de lignes affectées :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

Il est à noter que `ROW_COUNT` s'applique à l'ordre SQL précédent, quel qu'il soit :

- PERFORM ;
- EXECUTE ;
- Ou même à un ordre statique directement dans le code PL/PgSQL.

3.6.8 EXÉCUTION D'UNE REQUÊTE - EXECUTE - 1

- Instruction :

```
EXECUTE '<chaine>' [INTO [STRICT] cible];
```

- Exécute la requête comprise dans la variable chaîne
- La variable chaine peut être construite à partir d'autres variables

- Cible contient le résultat de l'exécution de la requête dans le cas d'un résultat sur une seule ligne
- Mot clé USING supplémentaire depuis PostgreSQL 8.4.

3.6.9 EXÉCUTION D'UNE REQUÊTE - EXECUTE - 2

- Sans `STRICT`, cible contient la première ligne d'un résultat multi-lignes ou NULL s'il n'y a pas de résultat.
- Avec `STRICT`, une exception est levée si le résultat ne contient aucune ligne (`NO_DATA_FOUND`) ou en contient plusieurs (`TOO_MANY_ROWS`).
- GET DIAGNOSTICS integer_var = ROW_COUNT

Nous verrons comment traiter les exceptions plus loin.

3.6.10 POUR CONSTRUIRE UNE REQUÊTE

- Fonction `quote_ident` pour mettre entre guillemets un identifiant d'un objet PostgreSQL (table, colonne, etc.)
- Fonction `quote_literal` pour mettre entre guillemets une valeur (chaîne de caractères)
- Fonction `quote_nullable` pour mettre entre guillemets une valeur (chaîne de caractères), sauf NULL qui sera alors renvoyé sans les guillemets
- L'opérateur de concaténation || est à utiliser pour concaténer tous les morceaux de la requête.
- Ou utiliser la fonction `format(...)`, équivalent de sprintf et disponible depuis 9.1

La fonction `format` est l'équivalent de la fonction `sprintf` : elle formate une chaine en fonction d'un patron et de valeurs à appliquer à ses paramètres et la retourne. Les type de paramètre reconnus par format sont :

- `%I` : est remplacé par un identifiant d'objet. C'est l'équivalent de la fonction `quote_ident`. L'objet en question est entouré en double-guillemet si nécessaire ;
- `%L` : est remplacé par une valeur littérale. C'est l'équivalent de la fonction `quote_literal`. Des simple-guillemet sont ajoutés à la valeur et celle-ci est correctement échappée si nécessaire ;
- `%s` : est remplacé par la valeur donnée sans autre forme de transformation ;
- `%%` : est remplacé par un simple `%`.

Voici un exemple d'utilisation de cette fonction, utilisant des paramètre positionnels :

```
select format(
    'SELECT %I FROM %I WHERE %1$I=%3$L',
    'MaColonne',
    'ma_table',
    $$l'été$$
);
                            format
-----------------------------------------------------------
 SELECT "MaColonne" FROM ma_table WHERE "MaColonne"='l''été'
```

3.6.11 EXÉCUTION D'UNE REQUÊTE - EXECUTE - 3

- EXECUTE command-string [INTO [STRICT] target] [USING expression [, ...]];
- Permet de créer une requête dynamique avec des variables de substitution
- Beaucoup plus lisible que des `quote_nullable` :

```
EXECUTE 'SELECT count(*) FROM mytable
WHERE inserted_by = $1 AND inserted <= $2'
INTO c
USING checked_user, checked_date;
```

- Le nombre de paramètres de la requête doit être fixe, ainsi que leur type
- Ne concerne pas les identifiants !

3.7 STRUCTURES DE CONTRÔLES

- Pourquoi du PL?
- Le but du PL est de pouvoir effectuer des traitements procéduraux.
- Nous allons donc maintenant aborder les structures de contrôle.

3.7.1 TESTS IF/THEN/ELSE/END IF - 1

```
IF condition THEN
  instructions
[ELSEIF condition THEN
 instructions]
```

```
[ELSEIF condition THEN
  instructions]
[ELSE
  instructions]
END IF
```

Ce dernier est l'équivalent d'un CASE en C pour une vérification de plusieurs alternatives.

3.7.2 TESTS IF/THEN/ELSE/END IF - 2

Exemple :

```
IF nombre = 0 THEN
  resultat := 'zero';
ELSEIF nombre > 0 THEN
   resultat := 'positif';
ELSEIF nombre < 0 THEN
   resultat := 'négatif';
ELSE
   resultat := 'indéterminé';
END IF;
```

3.7.3 TESTS CASE

Deux possibilités :

- 1ère :

  ```
  CASE variable
  WHEN expression THEN instructions
  ELSE instructions
  END CASE
  ```

- 2nde :

  ```
  CASE
  WHEN expression-booléene THEN instructions
  ELSE instructions
  END CASE
  ```

Quelques exemples :

```
CASE x
  WHEN 1, 2 THEN
    msg := 'un ou deux';
```

```
  ELSE
    msg := 'autre valeur que un ou deux';
END CASE;

CASE
  WHEN x BETWEEN 0 AND 10 THEN
    msg := 'la valeur est entre 0 et 10';
  WHEN x BETWEEN 11 AND 20 THEN
  msg := 'la valeur est entre 11 et 20';
END CASE;
```

3.7.4 BOUCLE LOOP/EXIT/CONTINUE - 1

- Créer une boucle (label possible)
  - `LOOP` / `END LOOP` :
- Sortir de la boucle
  - `EXIT [label] [WHEN expression_booléenne]`
- Commencer une nouvelle itération de la boucle
  - `CONTINUE [label] [WHEN expression_booléenne]`

3.7.5 BOUCLE LOOP/EXIT/CONTINUE - 2

Exemple :

```
LOOP
  resultat := resultat + 1;
  EXIT WHEN resultat > 100;
  CONTINUE WHEN resultat < 50;
  resultat := resultat + 1;
END LOOP;
```

Cette boucle incrémente le resultat de 1 à chaque itération tant que la valeur de resultat est inférieur à 50. Ensuite, resultat est incrémenté de 1 deux fois. Arrivé à 100, la procédure sort de la boucle.

3.7.6 BOUCLE WHILE

- Instruction :

```
WHILE condition LOOP instructions END LOOP;
```

- Boucle jusqu'à ce que la condition soit fausse
- Label possible

3.7.7 BOUCLE FOR - 1

- Synopsys :

```
FOR variable in [REVERSE] entier1..entier2 [BY incrément]
LOOP
instructions
END LOOP;
```

- `variable` va obtenir les différentes valeurs entre entier1 et entier2
- Label possible.

3.7.8 BOUCLE FOR - 2

- L'option `BY` permet d'augmenter l'incrémentation :

```
FOR variable in 1..10 BY 5...
```

- L'option `REVERSE` permet de faire défiler les valeurs en ordre inverse :

```
FOR variable in REVERSE 10..1 ...
```

3.7.9 BOUCLE FOR... IN... LOOP

- Permet de boucler dans les lignes résultats d'une requête
- Exemple :

```
FOR ligne IN SELECT * FROM ma_table LOOP
  instructions
END LOOP;
```

- Label possible
- `ligne` de type RECORD, ROW ou liste de variables séparées par des virgules
- Utilise un curseur en interne

Exemple :

```
FOR a, b, c, d IN SELECT col_a, col_b, col_c, col_d FROM ma_table
LOOP
  -- instructions
END LOOP;
```

3.7.10 BOUCLE FOREACH

- Permet de boucler sur les éléments d'un tableau
- Syntaxe :

```
FOREACH variable [SLICE n] IN ARRAY expression LOOP
    instructions
END LOOP
```

- `variable` va obtenir les différentes valeurs du tableau retourné par `expression`
- `SLICE` permet de jouer sur le nombre de dimensions du tableau à passer à la variable
- label possible

Voici deux exemples permettant d'illustrer l'utilité de `SLICE` :

- sans `SLICE` :

```
do $$
declare a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int;
begin
  foreach b in array a loop
  raise info 'var: %', b;
end loop;
end $$;
INFO:  var: 1
INFO:  var: 2
INFO:  var: 3
INFO:  var: 4
INFO:  var: 5
INFO:  var: 6
```

- Avec `SLICE` :

```
do $$
declare a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int[];
begin
```

```
  foreach b slice 1 in array a loop
  raise info 'var: %', b;
end loop;
end $$;
INFO:  var: {1,2}
INFO:  var: {3,4}
INFO:  var: {5,6}
```

3.8 RETOUR D'UNE FONCTION

- `RETURN [expression]`
- Renvoie cette expression à la requête appelante
- `expression` optionnelle si argument(s) déclarés OUT
  - `RETURN` lui-même optionnel si argument(s) déclarés OUT

3.8.1 RETURN NEXT

- Fonction `SETOF`, aussi appelé fonction SRF (`Set Returning Function`)
- Fonctionne avec des types scalaires (normaux) et des types composites
- `RETURN NEXT` renvoie une ligne du `SETOF`
- Cette fonction s'appelle de cette façon :

  ```
  SELECT * FROM ma_fonction();
  ```

- `expression` de renvoi optionnelle si argument de mode OUT

Tout est conservé en mémoire jusqu'à la fin de la fonction. Donc, si beaucoup de données sont renvoyées, cela pourrait occasionner quelques lenteurs.

Par ailleurs, il est possible d'appeler une SRF par

```
SELECT ma_fonction();
```

Dans ce cas, on récupère un résultat d'une seule colonne, de type composite.

3.8.2 RETURN QUERY

- Fonctionne comme `RETURN NEXT`

- `RETURN QUERY la_requete`
- `RETURN QUERY EXECUTE chaine_requete`

Par ce mécanisme, on peut très simplement produire une fonction retournant le résultat d'une requête complexe fabriquée à partir de quelques paramètres.

3.9 CONCLUSION

- Ajoute un grand nombre de structure de contrôle (test, boucle, etc.)
- Facile à utiliser et à comprendre
- Attention à la compatibilité ascendante

3.9.1 POUR ALLER PLUS LOIN

- Documentation officielle
  - « Chapitre 40. PL/pgsql - Langage de procédures SQL »

La documentation officielle sur le langage PL/pgsql peut être consultée en français à cette adresse[58] .

3.9.2 QUESTIONS

N'hésitez pas, c'est le moment !

3.10 TRAVAUX PRATIQUES

3.10.1 ÉNONCÉS

TP1.1 Écrire une fonction `hello` qui renvoie la chaîne de caractère « Hello World! » en SQL.

Écrire une fonction `hello_pl` qui renvoie la chaîne de caractère « Hello World! » en PL/PgSQL.

[58] http://docs.postgresql.fr/9.4/plpgsql.html

17.12

Comparer les coûts des deux plans d'exécutions de ces requêtes. Expliquer les coûts.

TP1.2 Écrire une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre flottant.

Écrire cette même fonction en SQL.

Comment corriger le problème de la division par zéro ? Écrivez cette nouvelle fonction dans les deux languages.

Conseil : dans ce genre de calcul impossible, il est possible d'utiliser avec PostgreSQL la constante `NaN` (*Not A Number*).

TP1.3 Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre.

Par exemple, appeler la fonction avec comme arguments les chaînes « deux » et « trois » doit renvoyer 6.

Essayez de multiplier « deux » par 4. Qu'obtenez-vous ? Pourquoi ?

Écrire une fonction permettant de factoriser cet exemple.

TP1.4 Écrire une fonction qui prend en argument le nom de l'utilisateur puis lui dit « Bonjour » ou « Bonsoir » suivant l'heure actuelle. Pour cela, aidez-vous de la fonction `to_char()`.

Écrire la même fonction avec un paramètre `OUT`.

Trouvez une meilleure méthode, plus performante, que l'utilisation de `to_char()` pour calculer l'heure courante.

Ré-écrivez la fonction en SQL.

TP1.5

Problème

Écrire une fonction qui calcule la date de Pâques.

Écrire une fonction qui calcule la date de la Pentecôte.

Enfin, écrire une fonction qui renvoie tous les jours fériés d'une année (libellé et date).

Quelques conseils

Pour le calcul de Pâques :

Soit `m` l'année. On calcule successivement :

- le reste de `m/19` : c'est la valeur de `a`.
- le reste de `m/4` : c'est la valeur de `b`.
- le reste de `m/7` : c'est la valeur de `c`.
- le reste de `(19a + p)/30` : c'est la valeur de `d`.
- le reste de `(2b + 4c + 6d + q)/7` : c'est la valeur de `e`.

Les valeurs de `p` et de `q` varient de 100 ans en 100 ans. De 2000 à 2100, `p` vaut 24, `q` vaut 5.

La date de Pâques est le `(22 + d + e)` mars ou le `(d + e - 9)` avril.

En ce qui concerne l'Ascension, cette fête a lieu le jeudi de la sixième semaine après Pâques (soit trente-neuf jours après Pâques).

TP1.6 Écrire une fonction qui inverse une chaîne (si « toto » en entrée, « otot » en sortie). (note : une fonction `reverse()` existe déjà dans PostgreSQL, utiliser un autre nom pour cette fonction, par exemple `inverser()`).

Utiliser la fonction `\timing` de psql pour tester la rapidité de cette fonction. Quelque fois, PL/pgSQL n'est pas un langage assez rapide.

TP1.7 Utilisation de la base `cave` pour la suite du TP1.

Créer une fonction `nb_bouteilles` qui renvoie le nombre de bouteilles en stock suivant une année et un type de vin (donc passés en paramètre).

Utiliser la fonction `generate_series()` pour extraire le nombre de bouteilles en stock sur plusieurs années, de 1990 à 1999.

TP1.8 Créer une fonction qui renvoie le même résultat que la requête précédente mais sans utiliser `generate_series`. Elle a donc trois arguments : l'année de début, l'année de fin et le type de fin. Elle renvoie pour chaque année, l'année elle-même et le nombre de bouteilles pour cette année. Elle peut faire appel à `nb_bouteilles`.

Pour aller plus loin - Écrire la même fonction que la question précédente, mais avec des paramètres `OUT`. - Écrire la même fonction que la question précédente, mais en utilisant une fonction variadic pour les années.

3.10.2 SOLUTIONS

TP1.1 Solution :

```
CREATE OR REPLACE FUNCTION hello()
RETURNS text
AS $BODY$
  SELECT 'hello world !'::text;
$BODY$
LANGUAGE SQL;

CREATE OR REPLACE FUNCTION hello_pl()
RETURNS text
AS $BODY$
  BEGIN
    RETURN 'hello world !';
  END
$BODY$
LANGUAGE plpgsql;
```

Requêtage :

```
cave=# explain SELECT hello();
                QUERY PLAN
------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=0)
(1 ligne)

cave=# explain SELECT hello_pl();
                QUERY PLAN
------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=0)
(1 ligne)
```

Par défaut, si on ne précise pas le coût (`COST`) d'une fonction, cette dernière a un coût par défaut de 100. Ce coût est à multiplier par la valeur de `cpu_operator_cost`, par défaut à 0.0025. Le coût total d'appel de la fonction `hello_pl` est donc par défaut de :

```
100*cpu_operator_cost + cpu_tuple_cost
```

TP1.2 Solution :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS float4
AS $BODY$
  BEGIN
```

```
    RETURN arg1::float4/arg2::float4;
  END
$BODY$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION division_sql(integer, integer)
RETURNS float4
AS $BODY$
    SELECT $1::float4/$2::float4;
$BODY$
LANGUAGE SQL;
```

NB : Avant la version 9.2 de PostgreSQL, les paramètres nommés ne sont pas supportés dans les fonctions SQL.

Requêtage :

```
cave=# SELECT division(4,2);
 division
----------
        2
(1 ligne)

cave=# SELECT division(1,5);
 division
----------
      0.2
(1 ligne)

cave=# SELECT division(1,0);
ERREUR:  division par zéro
CONTEXTE : PL/pgSQL function "division" line 2 at return
```

Solution 2 :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS float4
AS $BODY$
  BEGIN
    IF arg2 = 0 THEN
      RETURN 'NaN';
    ELSE
      RETURN arg1::float4/arg2::float4;
    END IF;
  END $BODY$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION division_sql(arg1 integer, arg2 integer)
RETURNS float4
```

```
AS $BODY$
  SELECT CASE $2
    WHEN 0 THEN 'NaN'
    ELSE $1::float4/$2::float4
  END;
$BODY$
LANGUAGE SQL;
```

Requêtage 2 :

```
cave=# SELECT division(1,5);
 division
----------
      0.2
(1 ligne)

cave=# SELECT division(1,0);
 division
----------
      NaN
(1 ligne)


cave=# SELECT division_sql(2,0);
 division_sql
--------------
          NaN
(1 ligne)

cave=# SELECT division_sql(2,6);
 division_sql
--------------
     0.333333
(1 ligne)
```

TP1.3 Solution :

```
CREATE OR REPLACE FUNCTION multiplication(arg1 text, arg2 text)
RETURNS integer
AS $BODY$
  DECLARE
    a1 integer;
    a2 integer;
  BEGIN
    IF arg1 = 'zéro' THEN
      a1 := 0;
```

```
    ELSEIF arg1 = 'un' THEN
      a1 := 1;
    ELSEIF arg1 = 'deux' THEN
      a1 := 2;
    ELSEIF arg1 = 'trois' THEN
      a1 := 3;
    ELSEIF arg1 = 'quatre' THEN
      a1 := 4;
    ELSEIF arg1 = 'cinq' THEN
      a1 := 5;
    ELSEIF arg1 = 'six' THEN
      a1 := 6;
    ELSEIF arg1 = 'sept' THEN
      a1 := 7;
    ELSEIF arg1 = 'huit' THEN
      a1 := 8;
    ELSEIF arg1 = 'neuf' THEN
      a1 := 9;
    END IF;

    IF arg2 = 'zéro' THEN
      a2 := 0;
    ELSEIF arg2 = 'un' THEN
      a2 := 1;
    ELSEIF arg2 = 'deux' THEN
      a2 := 2;
    ELSEIF arg2 = 'trois' THEN
      a2 := 3;
    ELSEIF arg2 = 'quatre' THEN
      a2 := 4;
    ELSEIF arg2 = 'cinq' THEN
      a2 := 5;
    ELSEIF arg2 = 'six' THEN
      a2 := 6;
    ELSEIF arg2 = 'sept' THEN
      a2 := 7;
    ELSEIF arg2 = 'huit' THEN
      a2 := 8;
    ELSEIF arg2 = 'neuf' THEN
      a2 := 9;
    END IF;

    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

17.12

Requêtage:

```
cave=# SELECT multiplication('deux', 'trois');
 multiplication
----------------
              6
(1 ligne)

cave=# SELECT multiplication('deux', 'quatre');
 multiplication
----------------
              8
(1 ligne)

cave=# SELECT multiplication('deux', 4::text);
 multiplication
----------------


(1 ligne)
```

Solution 2 :

```
CREATE OR REPLACE FUNCTION texte_vers_entier(arg text)
RETURNS integer AS $BODY$
  DECLARE
    ret integer;
  BEGIN
    IF arg = 'zéro' THEN
      ret := 0;
    ELSEIF arg = 'un' THEN
      ret := 1;
    ELSEIF arg = 'deux' THEN
      ret := 2;
    ELSEIF arg = 'trois' THEN
      ret := 3;
    ELSEIF arg = 'quatre' THEN
      ret := 4;
    ELSEIF arg = 'cinq' THEN
      ret := 5;
    ELSEIF arg = 'six' THEN
      ret := 6;
    ELSEIF arg = 'sept' THEN
      ret := 7;
    ELSEIF arg = 'huit' THEN
      ret := 8;
    ELSEIF arg = 'neuf' THEN
      ret := 9;
    ELSE
```

```
      RAISE NOTICE 'argument "%" invalide', arg;
      ret := NULL;
    END IF;

    RETURN ret;
  END
$BODY$
LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION multiplication(arg1 text, arg2 text)
RETURNS integer
AS $BODY$
  DECLARE
    a1 integer;
    a2 integer;
  BEGIN
    a1 := texte_vers_entier(arg1);
    a2 := texte_vers_entier(arg2);
    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

Requêtage 2 :

```
cave=# SELECT multiplication('deux', 4::text);
NOTICE:  argument "4" invalide
CONTEXTE : PL/pgSQL function "multiplication" line 6 at assignment
 multiplication
----------------

(1 ligne)

cave=# SELECT multiplication('deux', 'quatre');
 multiplication
----------------
              8
(1 ligne)
```

TP1.4 Solution :

```
CREATE OR REPLACE FUNCTION salutation(utilisateur text)
RETURNS text
AS $BODY$
  DECLARE
    heure integer;
```

```
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    RETURN libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Requêtage :

```
cave=# SELECT salutation ('Guillaume');
     salutation
--------------------
 Bonsoir Guillaume !
(1 ligne)
```

Solution 2 :

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Requêtage 2 :

```
cave=# SELECT salutation ('Guillaume');
     salutation
--------------------
```

```
 Bonsoir Guillaume !
(1 ligne)
```

Solution 3 :

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    SELECT INTO heure extract(hour from now())::int;
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Fonction en SQL :

```
CREATE OR REPLACE FUNCTION salutation_sql(nom text)
RETURNS text
AS $$
  SELECT CASE extract(hour from now()) > 12
    WHEN 't' THEN 'Bonsoir '||$1
    ELSE 'Bonjour '||$1
  END::text;
$$ LANGUAGE SQL;
```

TP1.5 Solution :

```
CREATE OR REPLACE FUNCTION paques(annee integer)
RETURNS timestamp
AS $$
  DECLARE
    a integer;
    b integer;
    r date;
  BEGIN
    a := (19*(annee % 19) + 24) % 30;
    b := (2*(annee % 4) + 4*(annee % 7) + 6*a + 5) % 7;
```

```
    SELECT (annee::text||'-03-31')::date + (a+b-9) INTO r;
    RETURN r;
  END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION ascension(annee integer)
RETURNS timestamp
AS $$
  DECLARE
    r    date;
  BEGIN
    SELECT paques(annee)::date + 40 INTO r;
    SELECT r + (4 - extract(dow from r))::integer INTO r;
    RETURN r;
  END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION vacances(annee integer, alsace_moselle boolean)
RETURNS SETOF record
AS $$
  DECLARE
    f integer;
    r record;
  BEGIN
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO r;
    RETURN NEXT r;
    SELECT 'Ascension'::text, ascension(annee)::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date INTO r;
    RETURN NEXT r;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date INTO r;
    RETURN NEXT r;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date INTO r;
    RETURN NEXT r;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO r;
    RETURN NEXT r;
```

```
    -- insertion des deux jours supplémentaires dans le cas
    -- de l'Alsace et la Moselle
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO r;
      RETURN NEXT r;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date INTO r;
      RETURN NEXT r;
    END IF;

    RETURN;
  END;
$$
LANGUAGE plpgsql;
```

Requêtage :

```
cave=# SELECT * FROM vacances(2007, true) AS (libelle text, jour date);
       ORDER BY jour;
      libelle       |    jour
--------------------+------------
 Jour de l'an       | 2017-01-01
 Vendredi saint     | 2017-04-14
 Pâques             | 2017-04-17
 Fête du travail    | 2017-05-01
 Victoire 1945      | 2017-05-08
 Ascension          | 2017-05-25
 Fête nationale     | 2017-07-14
 Assomption         | 2017-08-15
 La toussaint       | 2017-11-01
 Armistice 1918     | 2017-11-11
 Noël               | 2017-12-25
 Lendemain de Noël  | 2017-12-26
(12 rows)
```

Cette solution nécessite d'utiliser un alias de la fonction lors de son appel, sinon PostgreSQL ne sait pas déterminer la définition des colonnes retournées par la fonction.

Solution 2 :

Une autre forme d'écriture possible consiste à indiquer les deux colonnes de retour comme des paramètres `OUT` :

```
CREATE OR REPLACE FUNCTION
public.vacances(annee integer, alsace_moselle boolean DEFAULT false,
                OUT libelle text, OUT jour date)
 RETURNS SETOF record
 LANGUAGE plpgsql
```

17.12

```
AS $function$
  DECLARE
    f integer;
    r record;
  BEGIN
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Ascension'::text, ascension(annee)::date INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date INTO libelle, jour;
    RETURN NEXT;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO libelle, jour;
    RETURN NEXT;

    -- insertion des deux jours supplémentaires dans le cas
    -- de l'Alsace et la Moselle
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO libelle, jour;
      RETURN NEXT;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date
        INTO libelle, jour;
      RETURN NEXT;
    END IF;

    RETURN;
  END;
$function$;
```

La fonction s'utilise alors de façon simple :

```
cave=# SELECT * FROM vacances(2015, false);
     libelle      |    jour
------------------+------------
 Jour de l'an     | 2017-01-01
 Pâques           | 2017-04-17
 Ascension        | 2017-05-25
 Fête du travail  | 2017-05-01
 Victoire 1945    | 2017-05-08
 Fête nationale   | 2017-07-14
 Assomption       | 2017-08-15
 La toussaint     | 2017-11-01
 Armistice 1918   | 2017-11-11
 Noël             | 2017-12-25
(10 rows)
```

À noter l'ajout d'une valeur par défaut pour le paramètre `alsace_moselle`. Cela permet d'appeler la fonction `vacances` sans spécifier le seconde argument :

```
cave=# SELECT * FROM vacances(2015);
     libelle      |    jour
------------------+------------
 Jour de l'an     | 2017-01-01
 Pâques           | 2017-04-17
 Ascension        | 2017-05-25
 Fête du travail  | 2017-05-01
 Victoire 1945    | 2017-05-08
 Fête nationale   | 2017-07-14
 Assomption       | 2017-08-15
 La toussaint     | 2017-11-01
 Armistice 1918   | 2017-11-11
 Noël             | 2017-12-25
(10 rows)
```

TP1.6 Solution :

```
CREATE OR REPLACE FUNCTION inverser(varchar)
RETURNS varchar
AS $PROC$
  DECLARE
    str_in ALIAS FOR $1;
    str_out varchar;
    str_temp varchar;
    position integer;
  BEGIN
    -- Initialisation de str_out, sinon sa valeur reste à NULL
    str_out := '';
```

```
    -- Suppression des espaces en début et fin de chaîne
    str_temp := trim(both ' ' from str_in);
    -- position initialisée a la longueur de la chaîne
    -- la chaîne est traitée a l?envers
    position := char_length(str_temp);
    -- Boucle: Inverse l'ordre des caractères d'une chaîne de caractères
    WHILE position > 0 LOOP
    -- la chaîne donnée en argument est parcourue
    -- à l'envers,
    -- et les caractères sont extraits individuellement au
    -- moyen de la fonction interne substring
      str_out := str_out || substring(str_temp, position, 1);
      position := position - 1;
    END LOOP;
    RETURN str_out;
  END;
$PROC$
LANGUAGE plpgsql;
```

TP1.7 Solution :

```
CREATE OR REPLACE FUNCTION nb_bouteilles(v_annee integer, v_typevin text)
RETURNS integer
AS $BODY$
  DECLARE
    nb integer;
  BEGIN
    SELECT INTO nb  count(*) FROM vin v
      JOIN stock s ON v.id=s.vin_id
      JOIN type_vin t ON v.type_vin_id=t.id
    WHERE
      s.annee=v_annee
      AND t.libelle=v_typevin;
    RETURN nb;
  END
$BODY$
LANGUAGE 'plpgsql';
```

Requêtage :

- Sur une seule année :

```
SELECT nb_bouteilles(2000, 'rouge');
```

- Sur une période :

```
SELECT a, nb_bouteilles(a, 'rose')
FROM generate_series(1990, 1999) AS a
```

```
ORDER BY a;
```

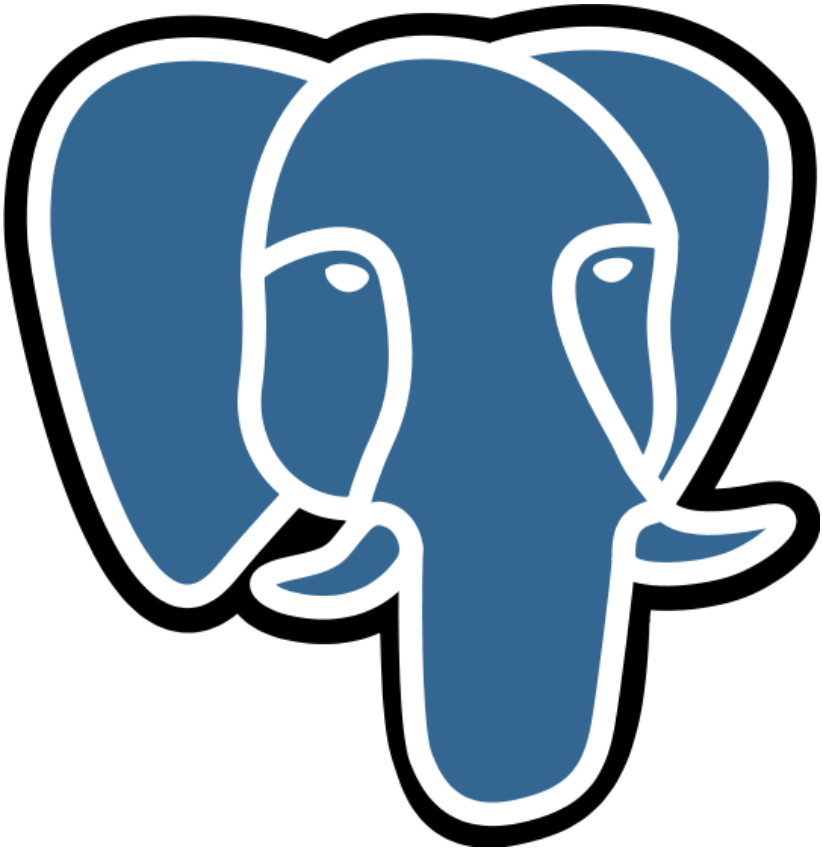
TP1.8 Solution :

```
CREATE OR REPLACE FUNCTION
nb_bouteilles(v_anneedeb integer, v_anneefin integer, v_typevin text)
RETURNS SETOF record
AS $BODY$
  DECLARE
    resultat record;
    i integer;
  BEGIN
    FOR i in v_anneedeb..v_anneefin
    LOOP
      SELECT INTO resultat i, nb_bouteilles(i, v_typevin);
      RETURN NEXT resultat;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE 'plpgsql';
```

Requêtage :

```
SELECT * FROM nb_bouteilles(1990, 2000, 'blanc')
  AS (annee integer, nb integer);
```

4 PL/PGSQL AVANCÉ



4.1 PRÉAMBULE

4.1.1 AU MENU

- Fonctions « variadic » et polymorphes

- Procédures trigger
- Curseurs
- Récupérer les erreurs
- Messages d'erreur dans les logs
- Sécurité
- Optimisation
- Problèmes fréquents

4.1.2 OBJECTIFS

Objectifs :

- Connaître la majorité des possibilités de PL/PgSQL
- Les utiliser pour étendre les fonctionnalités de la base
- Écrire du code robuste
- Éviter les pièges de sécurité
- Savoir optimiser une fonction

4.2 FONCTIONS VARIADIC

4.2.1 FONCTIONS « VARIADIC » : INTRODUCTION

- Permet de créer des fonctions avec un nombre d'arguments variables
- ... mais du même type

L'utilisation du mot clé `VARIADIC` dans la déclaration des fonctions permet d'utiliser un nombre variable d'arguments dans la mesure où tous les arguments optionnels sont du même type de données. Ces arguments sont passés à la fonction sous forme de tableau d'arguments du même type.

```
VARIADIC tableau text[]
```

Il n'est pas possible d 'utiliser d'autres arguments en entrée à la suite d'un paramètre `VARIADIC`.

4.2.2 FONCTIONS « VARIADIC » : EXEMPLE

Récupérer le minimum d'une liste :

```
CREATE FUNCTION pluspetit(VARIADIC numeric[])
RETURNS numeric AS $$
SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT pluspetit(10, -1, 5, 4.4);
 pluspetit
-----------
        -1
(1 row)
```

Quelques explications sur cette fonction :

- SQL est un langage de procédures stockées.
  - Une fonction SQL ne contient que des ordres SQL exécutés séquentiellements
  - Le résultat de la fonction est le résultat du dernier ordre
- generate_subscript() prend un tableau en premier paramètre et la dimension de ce tableau (un tableau peut avoir plusieurs dimensions), et elle retourne une série d'entiers allant du premier au dernier indice du tableau dans cette dimension
- g(i) est un alias : generate_subscripts est une SRF (`set-returning function`, retourne un `SETOF`), g est donc le nom de l'alias de table, et i le nom de l'alias de colonne.

4.2.3 FONCTIONS « VARIADIC » : EXEMPLE PLPGSQL

- En PL/PgSQL, cette fois-ci
- Démonstration de FOREACH xxx IN ARRAY aaa LOOP
- Précédemment, obligé de convertir le tableau en relation pour boucler (unnest)

En `PL/PgSQL`, il est possible d'utiliser une boucle `FOREACH` pour parcourir directement le tableau des arguments optionnels.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
BEGIN
```

```
FOREACH courant IN ARRAY liste LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

auparavant il fallait développer le tableau avec la fonction `unnest()` pour réaliser la même opération.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
BEGIN
FOR courant IN SELECT unnest(liste) LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

4.3 FONCTIONS POLYMORPHES

4.3.1 FONCTIONS POLYMORPHES : INTRODUCTION

- Typer les variables oblige à dupliquer les fonctions communes à plusieurs types
- PostgreSQL propose des types polymorphes
- Le typage se fait à l'exécution

Pour pouvoir utiliser la même fonction en utilisant des types différents, il est nécessaire de la redéfinir avec les différents types autorisés en entrée. Par exemple, pour autoriser l'utilisation de données de type `integer` ou `float` en entrée et retournés par une même fonction, il faut la dupliquer.

17.12

```
CREATE OR REPLACE FUNCTION
  addition(var1 integer, var2 integer)
RETURNS integer
AS $$
DECLARE
  somme integer;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION
  addition(var1 float, var2 float)
RETURNS float
AS $$
DECLARE
  somme float;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

L' utilisation de types polymorphes permet d'éviter ce genre de duplications fastidieuses.

4.3.2 FONCTIONS POLYMORPHES : « ANYELEMENT »

- Remplace tout type de données simples et composites
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Tous les paramètres et type de retour de type « anyelement » se voient attribués le même type
- Donc un seul type pour tous les anyelement autorisés
- Paramètre spécial $0 : du type attribué aux éléments « anyelement »

4.3.3 FONCTIONS POLYMORPHES : « ANYARRAY »

- « anyarray » remplace tout tableau de type de données simples et composites
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Le typage se fait à l'exécution
- Tous les paramètres de type « anyarray » se voient attribués le même type

4.3.4 FONCTIONS POLYMORPHES : EXEMPLE

L'addition est un exemple fréquent :

```
CREATE OR REPLACE FUNCTION
  addition(var1 anyelement, var2 anyelement)
RETURNS anyelement
AS $$
DECLARE
  somme ALIAS FOR $0;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

4.3.5 FONCTIONS POLYMORPHES : TESTS

```
# SELECT addition(1, 3);
 addition
----------
        4
(1 row)

# SELECT addition(1.3, 3.5);
 addition
----------
      4.8
(1 row)
```

L'opérateur + étant défini pour les entiers comme pour les numeric, la fonction ne pose aucun problème pour ces deux types de données, et retourne une donnée du même type que les données d'entrée.

4.3.6 FONCTIONS POLYMORPHES : PROBLÈME

- Attention lors de l'utilisation de type polymorphe...

17.12

```
# select addition('un'::text, 'mot'::text);
ERREUR:  L'opérateur n'existe pas : text + text
LIGNE 1 : SELECT   $1  +  $2
^
ASTUCE : Aucun opérateur correspond au nom donné et aux types d'arguments.
    Vous devez ajouter des conversions explicites de type.
REQUÊTE : SELECT   $1  +  $2
CONTEXTE : PL/pgSQL function "addition" line 4 at assignment
```

Le typage n'étant connu qu'à l'exécution, c'est aussi à ce moment là que se déclenchent les erreurs.

De même, l'affectation du type unique pour tous les éléments se fait sur la base du premier élément, ainsi :

```
# SELECT addition(1, 3.5);
ERROR:  function addition(integer, numeric) does not exist
LIGNE 1 : SELECT addition(1, 3.5);
                 ^
ASTUCE : No function matches the given name and argument types.
         You might need to add explicit type casts.
```

génère une erreur car du premier argument est déduit le type `integer`, ce qui n'est évidemment pas le cas du deuxième. Il peut donc être nécessaire d'utiliser un cast explicite pour résoudre ce genre de problématique.

```
# SELECT addition(1::numeric, 3.5);
 addition
----------
      4.5
(1 row)
```

4.4 FONCTIONS TRIGGER

4.4.1 PROCÉDURES TRIGGER : INTRODUCTION

- Procédure stockée
- Action déclenchée par INSERT (incluant COPY), UPDATE, DELETE, TRUNCATE
- Mode **par ligne** ou **par instruction**

- Exécution d'une procédure stockée codée à partir de tout langage de procédure activée dans la base de données

Un déclencheur est une spécification précisant que la base de données doit exécuter une fonction particulière quand un certain type d'opération est traité. Les fonctions déclencheurs peuvent être définies pour s'exécuter avant ou après une commande `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE`.

La fonction déclencheur doit être définie avant que le déclencheur lui-même puisse être créé. La fonction déclencheur doit être déclarée comme une fonction ne prenant aucun argument et retournant un type trigger.

Une fois qu'une fonction déclencheur est créée, le déclencheur est créé avec CREATE TRIGGER. La même fonction déclencheur est utilisable par plusieurs déclencheurs.

Un trigger `TRUNCATE` ne peut utiliser que le mode par instruction, contrairement aux autres triggers pour lesquels vous avez le choix entre « par ligne » et « par instruction ».

Enfin, l'instruction `COPY` est traitée comme s'il s'agissait d'une commande `INSERT`.

4.4.2 PROCÉDURES TRIGGER : VARIABLES (1/5)

- `OLD` :
  - type de données `RECORD` correspondant à la ligne avant modification
  - valable pour un `DELETE` et un `UPDATE`
- `NEW` :
  - type de données `RECORD` correspondant à la ligne après modification
  - valable pour un `INSERT` et un `UPDATE`

4.4.3 PROCÉDURES TRIGGER : VARIABLES (2/5)

- Ces deux variables sont valables uniquement pour les triggers en mode ligne
  - pour les triggers en mode instruction, la version 10 propose les tables de transition
- Accès aux champs par la notation pointée
  - `NEW.champ1` pour accéder à la nouvelle valeur de `champ1`

4.4.4 PROCÉDURES TRIGGER : VARIABLES (3/5)

- `TG_NAME` : nom du trigger qui a déclenché l'appel de la fonction
- `TG_WHEN` : chaîne valant `BEFORE`, `AFTER` ou `INSTEAD OF` suivant le type du trigger
- `TG_LEVEL` : chaîne valant `ROW` ou `STATEMENT` suivant le mode du tigger
- `TG_OP` : chaîne valant `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` suivant l'opération qui a déclenché le trigger

---

4.4.5 PROCÉDURES TRIGGER : VARIABLES (4/5)

- `TG_RELID` : `OID` de la table qui a déclenché le trigger
- `TG_TABLE_NAME` : nom de la table qui a déclenché le trigger
- `TG_TABLE_SCHEMA` : nom du schéma contenant la table qui a déclenché le trigger

Vous pourriez aussi rencontrer dans du code `TG_RELNAME`. C'est aussi le nom de la table qui a déclenché le trigger. Attention, cette variable est obsolète, il est préférable d'utiliser maintenant TG_TABLE_NAME

4.4.6 PROCÉDURES TRIGGER : VARIABLES (5/5)

- `TG_NARGS` : nombre d'arguments donnés à la fonction trigger
- `TG_ARGV` : les arguments donnés à la fonction trigger (le tableau commence à 0)

La fonction trigger est déclarée sans arguments mais il est possible de lui en passer dans la déclaration du trigger. Dans ce cas, il faut utiliser les deux variables ci-dessus pour y accéder. Attention, tous les arguments sont convertis en texte. Il faut donc se cantonner à des informations simples, sous peine de compliquer le code.

```
CREATE OR REPLACE FUNCTION verifier_somme()
RETURNS trigger AS $$
DECLARE
    fact_limit integer;
    arg_color varchar;
BEGIN
    fact_limit   := TG_ARGV[0];

    IF NEW.somme > fact_limit THEN
       RAISE NOTICE 'La facture % necessite une verification. '
                    'La somme % depasse la limite autorisee de %.',
                    NEW.idfact, NEW.somme, fact_limit;
```

```
    END IF;

    NEW.datecreate := current_timestamp;

    return NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER trig_verifier_debit
   BEFORE INSERT OR UPDATE ON test
   FOR EACH ROW
   EXECUTE PROCEDURE verifier_somme(400);

CREATE TRIGGER trig_verifier_credit
   BEFORE INSERT OR UPDATE ON test
   FOR EACH ROW
   EXECUTE PROCEDURE verifier_somme(800);
```

4.4.7 PROCÉDURES TRIGGER : RETOUR

- Une fonction trigger a un type de retour spécial, `trigger`
- Trigger `ROW`, `BEFORE` :
  - Si retour NULL, annulation de l'opération, sans déclencher d'erreur
  - Sinon, poursuite de l'opération avec cette valeur de ligne
- Trigger `ROW`, `AFTER` : valeur de retour ignorée
- Trigger `STATEMENT` : valeur de retour ignorée
- Pour ces deux derniers cas, annulation possible dans le cas d'une erreur à l'exécution de la fonction (que vous pouvez déclencher dans le code du trigger)

Une fonction de trigger retourne le type spécial `trigger`, pour cette raison ces fonctions ne peuvent être utilisées que dans le contexte d'un ou plusieurs triggers.

Il est possible d'annuler l'action d'un trigger de type ligne avant l'opération en retournant `NULL`. Ceci annule purement et simplement le trigger sans déclencher d'erreur.

Pour les triggers de type ligne intervenant après l'opération, comme pour les triggers à l'instruction, une valeur de retour est inutile. Elle est ignorée.

Il est possible d'annuler l'action d'un trigger de type ligne intervenant après l'opération ou d'un trigger à l'instruction, en remontant une erreur à l'exécution de la fonction.

4.4.8 PROCÉDURES TRIGGER : EXEMPLE - 1

- Horodater une opération sur une ligne :

```
CREATE TABLE ma_table (
id serial,
-- un certain nombre de champs informatifs
date_ajout timestamp,
date_modif timestamp);
```

4.4.9 PROCÉDURES TRIGGER : EXEMPLE - 2

```
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;
```

4.4.10 OPTIONS DE CREATE TRIGGER

CREATE TRIGGER permet quelques variantes :

- CREATE TRIGGER name WHEN (condition)
- CREATE TRIGGER name BEFORE UPDATE OF colx ON my_table
- CREATE CONSTRAINT TRIGGER : exécuté qu'au moment de la validation de la transaction
- CREATE TRIGGER view_insert INSTEAD OF INSERT ON my_view

- On peut ne déclencher un trigger que si une condition est vérifiée. Cela simplifie souvent le code du trigger, et gagne en performances : plus du tout besoin pour le moteur d'aller exécuter la fonction.
- On peut ne déclencher un trigger que si une colonne spécifique a été modifiée. Il ne s'agit donc que de triggers sur update. Encore un moyen de simplifier le code et de gagner en performances en évitant les déclenchements inutiles.

- On peut créer un trigger en le déclarant comme étant un trigger de contrainte. Il peut alors être 'deferrable', 'deferred', comme tout autre contrainte, c'est à dire n'être exécutée qu'au moment de la validation de la transaction, ce qui permet de ne vérifier les contraintes implémentées par le trigger qu'au moment de la validation finale.
- On peut créer un trigger sur une vue. C'est un trigger INSTEAD OF, qui permet de programmer de façon efficace les INSERT/UPDATE/DELETE/TRUNCATE sur les vues. Auparavant, il fallait passer par le système de règles (RULES), complexe et sujet à erreurs.

4.4.11 TABLES DE TRANSITION

- Pour les triggers de type AFTER et de niveau STATEMENT
- Possibilité de stocker les lignes avant et/ou après modification
  - `REFERENCING OLD TABLE`
  - `REFERENCING NEW TABLE`
- Par exemple

```
CREATE TRIGGER tr1
AFTER DELETE ON t1
REFERENCING OLD TABLE AS oldtable
FOR EACH STATEMENT
EXECUTE PROCEDURE log_delete();
```

Dans le cas d'un trigger en mode instruction, il n'est pas possible d'utiliser les variables `OLD` et `NEW` car elles ciblent une seule ligne. Pour cela, le standard SQL parle de tables de transition.

La version 10 de PostgreSQL permet donc de rattraper le retard à ce sujet par rapport au standard SQL et SQL Server.

Voici un exemple de leur utilisation.

Nous allons créer une table t1 qui aura le trigger et une table archives qui a pour but de récupérer les enregistrements supprimés de la table t1.

```
postgres=# CREATE TABLE t1 (c1 integer, c2 text);
CREATE TABLE

postgres=# CREATE TABLE archives (id integer GENERATED ALWAYS AS IDENTITY,
      dlog timestamp DEFAULT now(),
```

```
      t1_c1 integer, t1_c2 text);
CREATE TABLE
```

Maintenant, il faut créer le code de la procédure stockée :

```
postgres=# CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
        RETURN null;
      END
      $$;
CREATE FUNCTION
```

Et ajouter le trigger sur la table t1 :

```
postgres=# CREATE TRIGGER tr1
      AFTER DELETE ON t1
      REFERENCING OLD TABLE AS oldtable
      FOR EACH STATEMENT
      EXECUTE PROCEDURE log_delete();
CREATE TRIGGER
```

Maintenant, insérons un million de ligne dans t1 et supprimons-les :

```
postgres=# INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000

postgres=# DELETE FROM t1;
DELETE 1000000
Time: 2141.871 ms (00:02.142)
```

La suppression avec le trigger prend 2 secondes. Il est possible de connaître le temps à supprimer les lignes et le temps à exécuter le trigger en utilisant l'ordre `EXPLAIN ANALYZE` :

```
postgres=# TRUNCATE archives;
TRUNCATE TABLE

postgres=# INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000

postgres=# EXPLAIN (ANALYZE) DELETE FROM t1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=781.612..781.612 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.113..104.328 rows=1000000 loops=1)
 Planning time: 0.079 ms
```

```
 Trigger tr1: time=1501.688 calls=1
 Execution time: 2287.907 ms
(5 rows)
```

Donc la suppression des lignes met 0,7 seconde alors que l'exécution du trigger met 1,5 seconde.

Pour comparer, voici l'ancienne façon de faire (configuration d'un trigger en mode ligne) :

```
postgres=# CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) VALUES (old.c1, old.c2);
        RETURN null;
      END
      $$;
CREATE FUNCTION

postgres=# DROP TRIGGER tr1 ON t1;
DROP TRIGGER

postgres=# CREATE TRIGGER tr1
      AFTER DELETE ON t1
      FOR EACH ROW
      EXECUTE PROCEDURE log_delete();
CREATE TRIGGER

postgres=# TRUNCATE archives;
TRUNCATE TABLE

postgres=# TRUNCATE t1;
TRUNCATE TABLE

postgres=# INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000

postgres=# DELETE FROM t1;
DELETE 1000000
Time: 8445.697 ms (00:08.446)

postgres=# TRUNCATE archives;
TRUNCATE TABLE

postgres=# INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000

postgres=# EXPLAIN (ANALYZE) DELETE FROM t1;
                                QUERY PLAN
```

```
------------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=1049.420..1049.420 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.061..121.701 rows=1000000 loops=1)
 Planning time: 0.096 ms
 Trigger tr1: time=7709.725 calls=1000000
 Execution time: 8825.958 ms
(5 rows)
```

Donc avec un trigger en mode ligne, la suppression du million de lignes met presque 9 secondes à s'exécuter, dont 7,7 pour l'exécution du trigger. Sur le trigger en mode instruction, il faut compter 2,2 secondes, dont 1,5 sur le trigger. Les tables de transition nous permettent de gagner en performance.

Le gros intérêt des tables de transition est le gain en performance que cela apporte.

4.4.12 TABLES DE TRANSITION

- À partir de la version 10
- `REFERENCING OLD TABLE`
- `REFERENCING NEW TABLE`
- Par exemple

```
CREATE TRIGGER tr1
  AFTER DELETE ON t1
  REFERENCING OLD TABLE AS oldtable
  FOR EACH STATEMENT
  EXECUTE PROCEDURE log_delete();
```

Dans le cas d'un trigger en mode instruction, il n'est pas possible d'utiliser les variables `OLD` et `NEW` car elles ciblent une seule ligne. Pour cela, le standard SQL parle de tables de transition. Ces dernières ont été implémentées dans PostgreSQL pour la version 10. Voici un exemple de leur utilisation.

Nous allons créer une table `t1` et une table `poubelle` qui a pour but de récupérer les enregistrements supprimés de la table `t1` grâce à un trigger.

```
CREATE TABLE t1 (c1 integer, c2 text);

CREATE TABLE poubelle (id integer GENERATED ALWAYS AS IDENTITY,
  dlog timestamp DEFAULT now(),
```

```
  t1_c1 integer,
  t1_c2 text) ;
```

Maintenant, il faut créer le code de la procédure stockée :

```
CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  INSERT INTO poubelle (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
  RETURN null;
END
$$;
```

Et ajouter le trigger sur la table `t1` :

```
CREATE TRIGGER tr1
AFTER DELETE ON t1
REFERENCING OLD TABLE AS oldtable
FOR EACH STATEMENT
EXECUTE PROCEDURE log_delete();
```

Maintenant, insérons un million de lignes dans `t1` et supprimons-les :

```
$ INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
$ \timing on
$ DELETE FROM t1;
DELETE 1000000
Time: 6753.294 ms (00:06.753)
```

La suppression avec le trigger prend 6,7 secondes. Il est possible de connaître le temps à supprimer les lignes et le temps à exécuter le trigger en utilisant l'ordre `EXPLAIN ANALYZE` :

```
$ TRUNCATE poubelle;
TRUNCATE TABLE
Time: 545.840 ms
$ INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
Time: 4259.430 ms (00:04.259)
$ EXPLAIN (ANALYZE) DELETE FROM t1;
                        QUERY PLAN
-----------------------------------------------------------------------
 Delete on t1  (cost=0.00..17880.90 rows=1160690 width=6)
               (actual time=2552.210..2552.210 rows=0 loops=1)
```

```
   ->  Seq Scan on t1  (cost=0.00..17880.90 rows=1160690 width=6)
                         (actual time=0.071..183.026 rows=1000000 loops=1)
 Planning time: 0.094 ms
 Trigger tr1: time=4424.651 calls=1
 Execution time: 6982.290 ms
(5 rows)
```

Donc la suppression des lignes dure 2,5 secondes et l'exécution du trigger ajoute 4,4 secondes.

Pour comparer, voici l'ancienne façon de faire (configuration d'un trigger en mode *row*) :

```
$ CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO poubelle (t1_c1, t1_c2) VALUES (old.c1, old.c2);
    RETURN OLD;
END
$$;
CREATE FUNCTION
$ CREATE TRIGGER tr1
  BEFORE DELETE ON t1
  FOR EACH ROW
  EXECUTE PROCEDURE log_delete();
CREATE TRIGGER
$ TRUNCATE poubelle;
TRUNCATE TABLE
$ TRUNCATE t1;
TRUNCATE TABLE
$ INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
$ DELETE FROM t1;
DELETE 1000000
Time: 19021.459 ms (00:19.021)
$ TRUNCATE poubelle;
TRUNCATE TABLE
Time: 51.709 ms
$ INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
Time: 2382.365 ms (00:02.382)
$ EXPLAIN (ANALYZE) DELETE FROM t1;
                        QUERY PLAN
```

```
------------------------------------------------------------------------
 Delete on t1  (cost=0.00..10650.00 rows=1 width=6)
               (actual time=20819.006..20819.006 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..10650.00 rows=1 width=6)
                         (actual time=0.057..268.418 rows=1000000 loops=1)
 Planning time: 0.153 ms
 Trigger tr1: time=16140.528 calls=1000000
 Execution time: 20819.097 ms
(5 rows)
```

Donc avec un trigger en mode ligne, la suppression du million de lignes met 20 secondes à s'exécuter, dont 16 pour l'exécution du trigger. Sur le trigger en mode instruction, il faut compter 6,7 secondes, donc 4,4 pour le trigger.

Le gros intérêt des tables de transition est donc le gain en performance que cela apporte. Sur la suppression d'un million de lignes, il est deux fois plus rapide de passer par un trigger en mode instruction que par un trigger en mode ligne.

4.5 CURSEURS

4.5.1 CURSEURS : INTRODUCTION

- Exécuter une requête en une fois peut ramener beaucoup de résultats
- Tout ce résultat est en mémoire
  - risque de dépassement mémoire
- La solution : les curseurs
- Un curseur permet d'exécuter la requête sur le serveur mais de ne récupérer les résultats que petit bout par petit bout
- Dans une transaction ou une fonction

4.5.2 CURSEURS : DÉCLARATION D'UN CURSEUR

- Avec le type refcursor :

```
curseur refcursor;
```

- Avec la pseudo-instruction `CURSOR FOR` :

```
curseur CURSOR FOR SELECT * FROM ma_table;
```

- Avec une requête paramétrée :

```
curseur CURSOR (param integer) IS
SELECT * FROM ma_table WHERE un_champ=param;
```

La première forme permet la création d'un curseur non lié à une requête.

4.5.3 CURSEURS : OUVERTURE D'UN CURSEUR

- Lier une requête à un curseur :

```
OPEN curseur FOR requete
```

- Plan de la requête mis en cache
- Lier une requête dynamique à un curseur

```
OPEN curseur FOR EXECUTE chaine_requete
```

Voici un exemple de lien entre une requête et un curseur :

```
OPEN curseur FOR SELECT * FROM ma_table;
```

Et voici un exemple d' utilisation d'une requête dynamique :

```
OPEN curseur FOR EXECUTE 'SELECT * FROM ' || quote_ident(TG_TABLE_NAME);
```

4.5.4 CURSEURS : OUVERTURE D'UN CURSEUR LIÉ

- Instruction SQL : `OPEN curseur(arguments)`
- Permet d'ouvrir un curseur déjà lié à une requête
- Impossible d'ouvrir deux fois le même curseur
- Plan de la requête mise en cache
- Exemple :

```
curseur CURSOR FOR SELECT * FROM ma_table;
...
OPEN curseur;
```

4.5.5 CURSEURS : RÉCUPÉRATION DES DONNÉES

- Instruction SQL :

```
FETCH [ direction { FROM | IN } ] curseur INTO cible
```

- Récupère la prochaine ligne
- FOUND indique si cette nouvelle ligne a été récupérée
- Cible est :
  - une variable RECORD
  - une variable ROW
  - un ensemble de variables séparées par des virgules

4.5.6 CURSEURS : RÉCUPÉRATION DES DONNÉES

- `direction` du `FETCH` :
  - NEXT, PRIOR
  - FIRST, LAST
  - ABSOLUTE nombre, RELATIVE nombre
  - nombre
  - ALL
  - FORWARD, FORWARD nombre, FORWARD ALL
  - BACKWARD, BACKWARD nombre, BACKWARD ALL

4.5.7 CURSEURS : MODIFICATION DES DONNÉES

- Mise à jour d'une ligne d'un curseur :

```
UPDATE une_table SET ... WHERE CURRENT OF curseur
```

- Suppression d'une ligne d'un curseur :

```
DELETE FROM une_table WHERE CURRENT OF curseur
```

4.5.8 CURSEURS : FERMETURE D'UN CURSEUR

- Instruction SQL : `CLOSE curseur`

- Ferme le curseur
- Permet de récupérer de la mémoire
- Permet aussi de réouvrir le curseur

4.5.9 CURSEURS : RENVOI D'UN CURSEUR

- Fonction renvoyant une valeur de type `refcursor`
- Permet donc de renvoyer plusieurs valeurs

Voici un exemple d'utilisation d'une référence de curseur retournée par une fonction :

```
CREATE FUNCTION consult_all_stock(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM stock;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM consult_all_stock('cursor_a');

FETCH ALL FROM cursor_a;
COMMIT;
```

4.6 GESTION DES ERREURS

4.6.1 GESTION DES ERREURS : INTRODUCTION

Sans exceptions :

- Toute erreur provoque un arrêt de la fonction
- Toute modification suite à une instruction SQL (INSERT, UPDATE, DELETE) est annulée
- D'où l'ajout d'une gestion personnalisée des erreurs avec le concept des exceptions

---

4.6.2 GESTION DES ERREURS : UNE EXCEPTION

- La fonction comporte un bloc supplémentaire, EXCEPTION :

```
DECLARE
  -- déclaration des variables locales
BEGIN
  -- instructions de la fonction
EXCEPTION
WHEN condition THEN
  -- instructions traitant cette erreur
WHEN condition THEN
  -- autres instructions traitant cette autre erreur
  -- etc.
END
```

4.6.3 GESTION DES ERREURS : FLOT DANS UNE FONCTION

- L'exécution de la fonction commence après le `BEGIN`
- Si aucune erreur ne survient, le bloc EXCEPTION est ignoré
- Si une erreur se produit
  - tout ce qui a été modifié dans la base dans le bloc est annulé
  - les variables gardent par contre leur état
  - l'exécution passe directement dans le bloc de gestion de l'exception

---

4.6.4 GESTION DES ERREURS : FLOT DANS UNE EXCEPTION

- Recherche d'une condition satisfaisante
- Si cette condition est trouvée
  - exécution des instructions correspondantes
- Si aucune condition n'est compatible
  - sortie du bloc BEGIN/END comme si le bloc d'exception n'existait pas
  - passage de l'exception au bloc BEGIN/END contenant (après annulation de ce que ce bloc a modifié en base)
- Dans un bloc d'exception, les instructions `INSERT`, `UPDATE`, `DELETE` de la fonction ont été annulées
- Dans un bloc d'exception, les variables locales de la fonction ont gardé leur ancienne valeur

4.6.5 GESTION DES ERREURS : CODES D'ERREURS

- `SQLSTATE` : code d'erreur
- `SQLERRM` : message d'erreur
- par exemple :
  - `Data Exception` : division par zéro, overflow, argument invalide pour certaines fonctions, etc.
  - `Integrity Constraint Violation` : unicité, CHECK, clé étrangère, etc.
  - `Syntax Error`
  - `PL/pgsql Error` : `RAISE EXCEPTION`, pas de données, trop de lignes, etc.
- Les erreurs sont contenues dans des classes d'erreurs plus génériques, qui peuvent aussi être utilisées

Toutes les erreurs sont référencées dans la documentation[59]

Attention, des codes d'erreurs nouveaux apparaissent à chaque version.

La classe data_exception contient de nombreuses erreurs, comme datetime_field_overflow, invalid_escape_character, invalid_binary_representation... On peut donc, dans la déclaration de l'exception, intercepter toutes les erreurs de type data_exception d'un coup, ou une par une.

L'instruction `GET STACKED DIAGNOSTICS` permet d'avoir une vision plus précise de l'erreur récupéré par le bloc de traitement des exceptions. La liste de toutes les informations que l'on peut collecter est disponible dans la documentation[60] .

La démonstration ci-dessous montre comment elle peut être utilisée.

```
# CREATE TABLE t5(c1 integer PRIMARY KEY);
CREATE TABLE
# INSERT INTO t5 VALUES (1);
INSERT 0 1
# CREATE OR REPLACE FUNCTION test(INT4) RETURNS void AS $$
DECLARE
    v_state   TEXT;
    v_msg     TEXT;
    v_detail  TEXT;
    v_hint    TEXT;
    v_context TEXT;
BEGIN
    BEGIN
```

[59] http://docs.postgresql.fr/current/errcodes-appendix.html
[60] http://docs.postgresql.fr/current/plpgsql-control-structures.html#plpgsql-exception-diagnostics-values

```
        INSERT INTO t5 (c1) VALUES ($1);
    EXCEPTION WHEN others THEN
        GET STACKED DIAGNOSTICS
            v_state   = RETURNED_SQLSTATE,
            v_msg     = MESSAGE_TEXT,
            v_detail  = PG_EXCEPTION_DETAIL,
            v_hint    = PG_EXCEPTION_HINT,
            v_context = PG_EXCEPTION_CONTEXT;
        raise notice E'Et une exception :
            state  : %
            message: %
            detail : %
            hint   : %
            context: %', v_state, v_msg, v_detail, v_hint, v_context;
    END;
    RETURN;
END;
$$ LANGUAGE plpgsql;
# SELECT test(2);
 test
------

(1 row)

# SELECT test(2);
NOTICE:  Et une exception :
            state  : 23505
            message: duplicate key value violates unique constraint "t5_pkey"
            detail : Key (c1)=(2) already exists.
            hint   :
            context: SQL statement "INSERT INTO t5 (c1) VALUES ($1)"
PL/pgSQL function test(integer) line 10 at SQL statement
 test
------

(1 row)
```

4.6.6 MESSAGES D'ERREURS : RAISE - 1

- Envoyer une trace dans les journaux et/ou vers le client
  - `RAISE niveau message`
- Niveau correspond au niveau d'importance du message
  - `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `EXCEPTION`

- Message est la trace à enregistrer
- Message dynamique... tout signe % est remplacé par la valeur indiquée après le message
- Champs DETAIL et HINT disponibles à partir de la version 8.4

4.6.7 MESSAGES D'ERREURS : RAISE - 2

Exemples :

```
RAISE WARNING 'valeur % interdite', valeur;

RAISE WARNING 'valeur % ambigue',
              valeur
              USING HINT = 'Controlez la valeur saisie en amont';
```

Les autres niveaux pour `RAISE` ne sont que des messages, sans déclenchement d'exception.

4.6.8 MESSAGES D'ERREURS : CONFIGURATION DES LOGS

- Deux paramètres importants pour les traces
- `log_min_messages`
  - niveau minimum pour que la trace soit enregistrée dans les journaux
- `client_min_messages`
  - niveau minimum pour que la trace soit envoyée au client
- Dans le cas d'un `RAISE NOTICE message`, il faut avoir soit `log_min_messages`, soit `client_min_messages`, soit les deux à la valeur `NOTICE` au minimum.

4.6.9 MESSAGES D'ERREURS : RAISE EXCEPTION - 1

- Annule le bloc en cours d'exécution : `RAISE EXCEPTION message`
- Sauf en cas de présence d'un bloc `EXCEPTION` gérant la condition `RAISE_EXCEPTION`
- message est la trace à enregistrer, et est dynamique... tout signe % est remplacé par la valeur indiquée après le message

---

4.6.10 MESSAGES D'ERREURS : RAISE EXCEPTION - 2

Exemple :

```
RAISE EXCEPTION 'erreur interne';
-- La chose à ne pas faire !
```

Le rôle d'une exception est d'intercepter une erreur pour exécuter un traitement permettant soit de corriger l'erreur, soit de remonter une erreur pertinente. Intercepter un problème pour retourner «erreur interne» n'est pas une bonne idée.

4.6.11 FLUX DES ERREURS DANS DU CODE PL

Les exceptions non traitées «remontent»

- de bloc BEGIN/END imbriqués vers les blocs parents (fonctions appelantes comprises)
- Jusqu'à ce que personne ne puisse les traiter
- Voir note pour démonstration. Commençons par une fonction sans exception.

Démonstration en plusieurs étapes :

```
# CREATE TABLE ma_table (
    id integer unique
);
CREATE TABLE

# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
  INSERT INTO ma_table VALUES (1);
END
$function$;
CREATE FUNCTION


# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
```

```
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 6 at SQL statement
```

Une exception a été remontée. Avec un message explicite.

```
# SELECT * FROM ma_table ;
 a
---
(0 row)
```

La fonction a bien été annulée

4.6.12 FLUX DES ERREURS DANS DU CODE PL - 2

- Les erreurs remontent
- Cette fois-ci on rajoute un bloc PL pour intercepter l'erreur.

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
  INSERT INTO ma_table VALUES (1);
EXCEPTION WHEN unique_violation THEN
  RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
  RAISE NOTICE 'erreur: %',sqlerrm;
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------


(1 row)
```

L'erreur est bien devenue un message de niveau NOTICE.

```
# SELECT * FROM ma_table ;
 a
```

```
---
(0 row)
```

La table n'en reste pas moins vide pour autant puisque le bloc a été annulé.

4.6.13 FLUX DES ERREURS DANS DU CODE PL - 3

- Cette fois-ci, on rajoute un bloc PL indépendant pour gérer le second insert.

Voici une nouvelle version de la fonction :

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
  -- Va déclencher une erreur de violation de contrainte d'unicité
    INSERT INTO ma_table VALUES (1);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------


(1 row)
```

En apparence, le résultat est identique.

```
# SELECT * FROM ma_table ;
 a
---
```

```
 1
(1 row)
```

Mais cette fois-ci, le bloc BEGIN parent n'a pas eu d'exception, il s'est donc bien terminé.

4.6.14 FLUX DES ERREURS DANS DU CODE PL - 4

- Illustrons maintenant la remontée d'erreurs.
- Nous avons deux blocs imbriqués.
- Une erreur non prévue va se produire dans le bloc intérieur.

On commence par ajouter une contrainte sur la colonne pour empêcher les valeurs supérieures ou égales à 10 :

```
# ALTER TABLE ma_table ADD CHECK (id < 10 ) ;
ALTER TABLE
```

Puis, on recrée la fonction de façon à ce qu' elle déclenche cette erreur dans le bloc le plus bas, et la gère uniquement dans le bloc parent :

```
CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
    -- Va déclencher une erreur de violation de check (col < 10)
    INSERT INTO ma_table VALUES (100);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
EXCEPTION WHEN check_violation THEN
  RAISE NOTICE 'violation de contrainte check';
  RAISE EXCEPTION 'mais on va remonter une exception à l''appelant, '
                  'juste pour le montrer';
END
$function$;
```

Exécutons la fonction :

```
# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 4 at SQL statement
```

C'est normal, nous avons toujours l' enregistrement à 1 du test précédent. L'exception se déclenche donc dans le bloc parent, sans espoir d'interception: nous n'avons pas d'exception pour lui.

Nettoyons donc la table, pour reprendre le test :

```
# TRUNCATE ma_table ;
TRUNCATE TABLE
# SELECT demo_exception();
NOTICE:  violation de contrainte check
ERREUR:  mais on va remonter une exception à l'appelant, juste pour le montrer
CONTEXT:  PL/pgSQL function demo_exception() line 17 at RAISE
```

Le gestionnaire d'exception qui intercepte l'erreur est bien ici celui de l'appelant. Par ailleurs, comme nous retournons nous-même une exception, la requête ne retourne pas de résultat, mais une erreur: il n'y a plus personne pour récupérer l'exception, c'est donc PostgreSQL lui-même qui s'en charge.

4.7 SÉCURITÉ

4.7.1 SÉCURITÉ : DROITS

- L'exécution de la fonction dépend du droit `EXECUTE`
- Par défaut, ce droit est donné à la création de la fonction :
  - au propriétaire de la fonction
  - au groupe spécial PUBLIC

4.7.2 SÉCURITÉ : AJOUT

- Ce droit peut être donné avec l'instruction SQL `GRANT` :

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION function_name [ ( [ [ argmode ] [ arg_name ]
                                                                    arg_type [, ...] ] ) ]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

4.7.3 SÉCURITÉ : SUPPRESSION

- Un droit peut être révoqué avec l'instruction SQL REVOKE :

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION function_name [ ( [ [ argmode ] [ arg_name ]
                                        arg_type [, ...] ] ) ] [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

4.7.4 SÉCURITÉ : SECURITY INVOKER/DEFINER

- SECURITY INVOKER : la fonction s'exécute avec les droits de l'utilisateur qui l'exécute
- SECURITY DEFINER : la fonction s'exécute avec les droits de l'utilisateur qui en est le propriétaire
  - Équivalent du sudo Unix
- Il faut impérativement sécuriser les variables d'environnement (surtout le search_path) en SECURITY DEFINER

Exemple d'une fonction en SECURITY DEFINER avec un search_path sécurisé :

```
CREATE OR REPLACE FUNCTION instance_is_in_backup ( )
RETURNS BOOLEAN AS $$
DECLARE is_exists BOOLEAN;
BEGIN
        -- Set a secure search_path: trusted schemas, then 'pg_temp'.
        PERFORM pg_catalog.set_config('search_path', 'pg_temp', true);
        SELECT ((pg_stat_file('backup_label')).modification IS NOT NULL)
        INTO is_exists;
        RETURN is_exists;
EXCEPTION
WHEN undefined_file THEN
  RETURN false;
```

```
END
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

4.7.5 SÉCURITÉ : LEAKPROOF

- `LEAKPROOF` : indique au planificateur que la fonction ne peut pas faire fuiter d'information de contexte
  - réservé aux super-utilisateurs
  - si on la déclare telle, s'assurer que la fonction est véritablement sûre !
- option utile lorsque l'on utilise des vues avec l'option `security_barrier`

Certains utilisateurs créent des vues pour filtrer des lignes, afin de restreindre la visibilité sur certaines données. Or, cela peut se révéler dangereux si un utilisateur malintentionné a la possibilité de créer une fonction car il peut facilement contourner cette sécurité si cette option n'est pas utilisée, notamment en jouant sur des paramètres de fonction comme `COST`, qui permet d'indiquer au planificateur un coût estimé pour la fonction.

En indiquant un coût extrêmement faible, le planificateur aura tendance à réécrire la requête, et à déplacer l'exécution de la fonction dans le code même de la vue, avant l'application des filtres restreignant l'accès aux données : la fonction a donc accès a tout le contenu de la table, et peut faire fuiter des données normalement inaccessibles, par exemple à travers l'utilisation de la commande `RAISE`.

L'option `security_barrier` des vues dans PostgreSQL bloque ce comportement du planificateur, mais en conséquence empêche le choix de plans d'exécutions potentiellement plus performants. Déclarer une fonction avec l'option `LEAKPROOF` permet d'indiquer à PostgreSQL que celle-ci ne peut pas occasionner de fuite d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut en optimiser l'exécution. Cette option n'est accessible qu'aux super-utilisateurs.

4.7.6 SÉCURITÉ : VISIBILITÉ DES SOURCES - 1

- Le code d'une fonction est visible par tout le monde
- Y compris ceux qui n'ont pas le droit d'exécuter la fonction
- Vous devez donc écrire un code robuste
  - pas espérer que, comme personne n'en a le code, personne ne trouvera de faille
- **Surtout pour les fonctions SECURITY DEFINER**

4.7.7 SÉCURITÉ : VISIBILITÉ DES SOURCES - 2

```
# SELECT proargnames, prosrc
FROM pg_proc WHERE proname='addition';

-[ RECORD 1 ]-------------------------
proargnames | {var1,var2}
prosrc      |
            :   DECLARE
            :     somme ALIAS FOR $0;
            :   BEGIN
            :     somme := var1 + var2;
            :     RETURN somme;
            :   END;
            :
```

La méta-commande psql `\df+ public.addition` permet également d'obtenir cette information.

4.7.8 SÉCURITÉ : INJECTIONS SQL

- Les paramètres d'une fonction doivent être considérés comme hostiles :
  - Ils contiennent des données non validées (qui appelle la fonction ?)
  - Ils peuvent, si l'utilisateur est imaginatif, être utilisés pour exécuter du code
- Utiliser `quote_ident`, `quote_literal` et `quote_nullable`

Voici un exemple simple :

```
CREATE TABLE ma_table_secrete1 (b integer, a integer);
INSERT INTO ma_table_secrete1 SELECT i,i from generate_series(1,20) i;

CREATE OR REPLACE FUNCTION demo_injection ( param1 text, value1 text )
 RETURNS SETOF ma_table_secrete1
 LANGUAGE plpgsql
 SECURITY DEFINER
AS $function$
-- Cette fonction prend un nom de colonne variable
-- et l'utilise dans une clause WHERE
-- Il faut donc une requête dynamique
-- Par contre, mon utilisateur 'normal' qui appelle
-- n'a droit qu'aux enregistrements où a<10
DECLARE
```

```
  ma_requete text;
  ma_ligne record;
BEGIN
  ma_requete := 'SELECT * FROM ma_table_secrete1 WHERE ' || param1 || ' = ' ||
                value1 || ' AND a < 10';
    RETURN QUERY EXECUTE ma_requete;
END
$function$;
```

```
# SELECT * from demo_injection ('b','2');
 a | b
---+---
 2 | 2
(1 row)


# SELECT * from demo_injection ('a','20');
 a | b
---+---
(0 row)
```

Tout va bien, elle effectue ce qui est demandé.

Par contre, elle effectue aussi ce qui n'est pas prévu :

```
# SELECT * from demo_injection ('1=1 --','');
  a  |  b
-----+-----
   1 |   1
   2 |   2
   3 |   3
   4 |   4
   5 |   5
   6 |   6
   7 |   7
   8 |   8
   9 |   9
  10 |  10
  11 |  11
  12 |  12
  13 |  13
  14 |  14
  15 |  15
  16 |  16
  17 |  17
  18 |  18
  19 |  19
  20 |  20
```

```
(20 lignes)
```

Cet exemple est évidemment simplifié.

Une règle demeure : ne jamais faire confiance aux paramètres d'une fonction. Au minimum, un `quote_ident` pour param1 et un `quote_literal` pour val1 étaient obligatoires, pour se protéger de ce genre de problèmes.

4.8 OPTIMISATION

4.8.1 FONCTIONS IMMUTABLE, STABLE OU VOLATILE - 1

- Par défaut, PostgreSQL considère que les fonctions sont `VOLATILE`
- `volatile` : Fonction dont l'exécution ne peut ni ne doit être évitée

Les fonctions de ce type sont susceptibles de renvoyer un résultat différent à chaque appel, comme par exemple `random()` ou `setval()`.

Toute fonction ayant des effets de bords doit être qualifiée `volatile` dans le but d'éviter que PostgreSQL utilise un résultat intermédiaire déjà calculé et évite ainsi d'exécuter le code de la fonction.

4.8.2 FONCTIONS "IMMUTABLE", "STABLE" OU "VOLATILE" - 2

- `immutable` : Fonctions déterministes, dont le résultat peut être précalculé avant de planifier la requête.

Certaines fonctions que l'on écrit sont déterministes. C'est à dire qu'à paramètre(s) identique(s), le résultat est identique.

Le résultat de telles fonctions est alors remplaçable par son résultat avant même de commencer à planifier la requête.

Voici un exemple qui utilise cette particularité :

```
create function factorielle (a integer) returns bigint as
$$
declare
  result bigint;
begin
```

```
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
$$
language plpgsql immutable;
```

```
# CREATE TABLE test (a bigint UNIQUE);
CREATE TABLE
# INSERT INTO test SELECT generate_series(1,1000000);
INSERT 0 1000000
# ANALYZE test;
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..16925.00 rows=1000000 width=8)
                   (actual time=0.032..130.921 rows=1000000 loops=1)
   Filter: (a < '479001600'::bigint)
 Planning time: 896.039 ms
 Execution time: 169.954 ms
(4 rows)
```

La fonction est exécutée une fois, remplacée par sa constante, et la requête est ensuite planifiée.

Si on déclare la fonction comme STABLE :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                        QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using test_a_key on test
        (cost=0.68..28480.67 rows=1000000 width=8)
              (actual time=0.137..115.592 rows=1000000 loops=1)
   Index Cond: (a < factorielle(12))
   Heap Fetches: 0
 Planning time: 4.682 ms
 Execution time: 153.762 ms
(5 rows)
```

La requête est planifiée sans connaître factorielle(12), donc avec une hypothèse très approximative sur la cardinalité. factorielle(12) est calculé, et la requête est exécutée. Grâce au `Index Only Scan`, le requête s'effectue rapidement.

Si on déclare la fonction comme VOLATILE :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
```

```
------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..266925.00 rows=333333 width=8)
                   (actual time=1.005..57519.702 rows=1000000 loops=1)
   Filter: (a < factorielle(12))
 Planning time: 0.388 ms
 Execution time: 57573.508 ms
(4 rows)
```

La requête est planifiée, et factorielle(12) est calculé pour chaque enregistrement de la table, car on ne sait pas si elle retourne toujours le même résultat.

4.8.3 FONCTIONS IMMUTABLE, STABLE OU VOLATILE - 3

- `stable` : Fonction ayant un comportement **stable** au sein d'un même ordre SQL.

Ces fonctions retournent la même valeur pour la même requête SQL, mais peuvent retourner une valeur différente dans la prochaine instruction.

Il s'agit typiquement de fonctions dont le traitement dépend d'autres valeurs dans la base de données, ou bien de réglages de configuration. Les fonctions de la famille `current_timestamp` sont `stable` car leurs valeurs n'évoluent pas au sein d'une même transaction.

PostgreSQL refusera de déclarer comme STABLE toute fonction modifiant la base : elle ne peut pas être stable si elle modifie la base.

4.8.4 OPTIMISATION : RIGUEUR

- Fonction `STRICT`
- La fonction renvoit NULL si au moins un des arguments est NULL

Les fonctions définies comme `STRICT` ou `RETURNS NULL ON NULL INPUT` annule l'exécution de la requête si l'un des paramètres passés est `NULL`. Dans ce cas la fonction est considérée comme ayant renvoyé `NULL`.

Si l'on reprend l'exemple de la fonction `factorielle()` :

```
create or replace function factorielle (a integer) returns bigint as
$$
declare
  result bigint;
begin
```

```
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
$$
language plpgsql immutable STRICT;
```

on obtient le résultat suivant si elle est exécutée avec la valeur NULL passée en paramètre :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(NULL);
                  QUERY PLAN
-----------------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=8)
         (actual time=0.002..0.002 rows=0 loops=1)
   One-Time Filter: false
 Planning time: 0.100 ms
 Execution time: 0.039 ms
(4 rows)
```

4.8.5 OPTIMISATION : EXCEPTION

- Un bloc contenant une clause EXCEPTION est plus coûteuse en entrée/sortie qu'un bloc sans :
  - un SAVEPOINT est créé à chaque fois pour pouvoir annuler le bloc uniquement.
- À utiliser avec parcimonie
- Un bloc BEGIN imbriqué a un coût aussi : un SAVEPOINT est créé à chaque fois.

4.8.6 REQUÊTE STATIQUE OU DYNAMIQUE ?

Les requêtes statiques :

- Sont écrites «en dur» dans le code PL/PgSQL
- Donc pas d'EXECUTE ou PERFORM
- Sont préparées une fois par session, à leur première exécution
- Peuvent avoir un plan générique lorsque c'est jugé utile par le planificateur

17.12

Avant la version 9.2, un plan générique (indépendant des paramètres de l'ordre SQL) était systématiquement généré et utilisé. Ce système permet de gagner du temps d'exécution si la requête est réutilisée plusieurs fois, et qu'elle est coûteuse à planifier.

Toutefois, un plan générique n'est pas forcément idéal dans toutes les situations, et peut conduire à des mauvaises performances.

Par exemple :

```
SELECT * FROM ma_table WHERE col_pk = param_function
```

est un excellent candidat à être écrit statiquement : le plan sera toujours le même : on attaque l'index de la primary key pour trouver l'enregistrement.

```
SELECT * FROM ma_table WHERE col_timestamp > param_function
```

est un moins bon candidat : le plan, idéalement, dépend de param _function : on ne parcourt pas la même fraction de la table suivant la valeur de param_function.

Par défaut, un plan générique ne sera utilisé dès la première exécution d'une requête statique que si celle-ci ne dépend d'aucun paramètre. Dans le cas contraire, cela ne se produira qu'au bout de plusieurs exécutions de la requête, et seulement si le planificateur détermine que les plans spécifiques utilisés n'apportent pas d'avantage par rapport au plan générique.

4.8.7 REQUÊTE STATIQUE OU DYNAMIQUE ? - 2

Les requêtes dynamiques :

- Sont écrites avec un EXECUTE, PERFORM…
- Sont préparées à chaque exécution
- Ont un plan optimisé
- Sont donc plus coûteuses en planification
- Mais potentiellement plus rapides à l'exécution

L'écriture d'une requête dynamique est par contre un peu plus pénible, puisqu'il faut fabriquer un ordre SQL, puis le passer en paramètre à EXECUTE, avec tous les quote_* que cela implique pour en protéger les paramètres.

Pour se faciliter la vie, on peut utiliser EXECUTE query USING param1, param2 …, qui est même quelquefois plus lisible que la syntaxe en dur : les paramètres de la requête sont clairement identifiés dans cette syntaxe.

Par contre, la syntaxe USING n'est utilisable que si le nombre de paramètres est fixe.

4.8.8 REQUÊTE STATIQUE OU DYNAMIQUE ? -3

Alors, statique ou dynamique ?

- Si la requête est simple, statique
  - peu de WHERE
  - peu ou pas de jointure
- Sinon dynamique

La limite est difficile à placer, il s'agit de faire un compromis entre temps de planification d'une requête (quelques dizaines de microsecondes pour une requête basique à potentiellement plusieurs secondes si on dépasse la dizaine de jointures) et le temps d'exécution.

Dans le doute, réalisez un test de performance de la fonction sur un jeu de données représentatif.

4.9 OUTILS

- Trois outils disponibles
  - un debugger
  - un pseudo profiler

Tous les outils d'administration PostgreSQL permettent d'écrire des procédures stockées en PL/pgsql, la plupart avec les fonctionnalités habituelles (comme le surlignage des mots clés, l'indentation automatique, etc.).

Par contre, pour aller plus loin, l'offre est restreinte. Il existe tout de même un debugger qui fonctionne avec pgAdmin III, sous la forme d'une extension.

4.9.1 PLDEBUGGER

- License Artistic 2.0
- À partir de PostgreSQL 8.2
- Installé par défaut avec le one-click installer
  - Mais non activé
- Compilation nécessaire pour les autres systèmes

pldebugger est un outil initialement créé par Dave Page et Korry Douglas au sein d'EnterpriseDB, repris par la communauté. Il est proposé sous license libre (Artistic 2.0).

Il fonctionne grâce à des hooks implémentés dans la version 8.2 de PostgreSQL. Du coup, cet outil ne fonctionne qu'à partir de PostgreSQL 8.2.

Il est assez peu connu, ce qui explique que peu l'utilisent. Seul l'outil d'installation « one-click installer » l'installe par défaut. Pour tous les autres systèmes, cela réclame une compilation supplémentaire. Cette compilation est d'ailleurs peu aisée étant donné qu'il n'utilise pas le système pgxs.

4.9.2 PLDEBUGGER - COMPILATION

- Récupérer le source avec git
- Copier le répertoire dans le répertoire contrib des sources de PostgreSQL
- Et les suivre étapes standards
  - make
  - make install

Voici les étapes à réaliser pour compiler pldebugger en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/usr/src/postgresql-10` et qu'ils ont été préconfigurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /usr/src/postgresql-10/contrib
```

- Cloner le dépôt git :

```
$ git clone git://git.postgresql.org/git/pldebugger.git
Cloning into 'pldebugger'...
remote: Counting objects: 441, done.
remote: Compressing objects: 100% (337/337), done.
remote: Total 441 (delta 282), reused 171 (delta 104)
Receiving objects: 100% (441/441), 170.24 KiB, done.
Resolving deltas: 100% (282/282), done.
```

- Se placer dans le nouveau répertoire `pldebugger` :

```
$ cd pldebugger
```

- Compiler pldebugger :

```
$ make
```

- Installer pldebugger :

```
$ make install
```

L'installation copie le fichier `plugin_debugger.so` dans le répertoire des bibliothèques partagées de PostgreSQL. L'installation copie ensuite les fichiers SQL et de contrôle de l'extension `pldbgapi` dans le répertoire `extension` du répertoire `share` de PostgreSQL.

4.9.3 PLDEBUGGER - ACTIVATION

- Configurer `shared_preload_libraries`
  - `shared_preload_libraries = 'plugin_debugger'`
- Redémarrer PostgreSQL
- Installer l'extension `pldbgapi` :

```
mabase# CREATE EXTENSION pldbgapi;
```

La configuration du paramètre `shared_preload_libraries` permet au démarrage de PostgreSQL de laisser la bibliothèque `plugin_debugger` s'accrocher aux hooks de l'interpréteur PL/pgsql. Du coup, pour que la modification de ce paramètre soit prise en compte, il faut redémarrer PostgreSQL.

L'interaction avec pldebugger se fait par l'intermédiaire de procédures stockées. Il faut donc au préalable créer ces procédures stockées dans la base contenant les procédures PL/pgsql à débugguer. Cela se fait en créant l'extension :

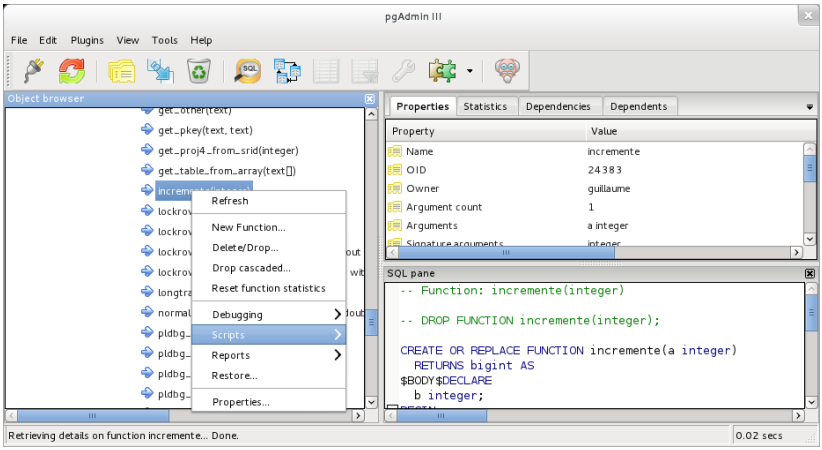
```
$ psql
psql (10)
Type "help" for help.

postgres# create extension pldbgapi;
CREATE EXTENSION
```

4.9.4 PLDEBUGGER - UTILISATION

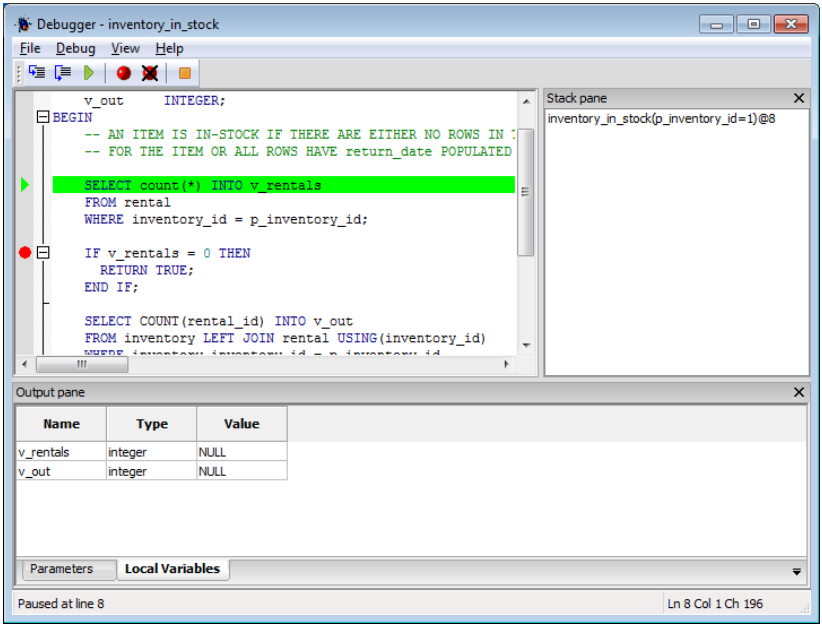
- Via pgAdmin (version 1.8 au minimum)

Le menu contextuel pour accéder au débuggage d'une fonction :

17.12



La fenêtre du débugger :



4.9.5 LOG_FUNCTIONS

- Créé par Dalibo
- License BSD
- Compilation nécessaire

log_functions est un outil créé par Guillaume Lelarge au sein de Dalibo. Il est proposé sous license libre (BSD).

4.9.6 LOG_FUNCTIONS - COMPILATION

- Récupérer l'archive sur PGXN.org
- Décompresser l'archive puis : `make USE_PGXS=1 && make USE_PGXS=1 install`

Voici les étapes à réaliser pour compiler pldebugger en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/home/guillaume/postgresql-9.1.4` et qu'ils ont été préconfigurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /home/guillaume/postgresql-9.1.4/contrib
```

- Récupérer le dépôt git de log_functions :

```
$ git://github.com/gleu/log_functions.git
Cloning into 'log_functions'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 24 (delta 8), reused 24 (delta 8)
Receiving objects: 100% (24/24), 11.71 KiB, done.
Resolving deltas: 100% (8/8), done.
```

- Se placer dans le nouveau répertoire `log_functions` :

```
$ cd log_functions
```

- Compiler log_functions :

```
$ make
```

- Installer log_functions :

```
$ make install
```

L'installation copie le fichier `log_functions.o` dans le répertoire des bibliothèques partagées de PostgreSQL.

17.12

Si la version de PostgreSQL est supérieure ou égale à la 9.2 alors l' installation est plus simple et les sources de PostgreSQL ne sont plus nécessaires.

Téléchargement de log_functions :

```
wget http://api.pgxn.org/dist/log_functions/1.0.0/log_functions-1.0.0.zip
```

puis décompression et installation de l'extension :

```
unzip log_functions-1.0.0.zip
cd log_functions-1.0.0/
make USE_PGXS=1 && make USE_PGXS=1 install
```

L'installation copie aussi le fichier `log_functions.so` dans le répertoire des bibliothèques partagées de PostgreSQL.

4.9.7 LOG_FUNCTIONS - ACTIVATION

- Permanente
  - `shared_preload_libraries = 'log_functions'`
  - Redémarrage de PostgreSQL
- Au cas par cas
  - LOAD 'log_functions'

Le module log_functions est activable de deux façons.

La première consiste à demander à PostgreSQL de le charger au démarrage. Pour cela, il faut configurer la variable `shared_preload_libraries`, puis redémarrer PostgreSQL pour que le changement soit pris en compte.

La deuxième manière de l'activer est de l'activer seulement au moment où son utilisation s'avère nécessaire. Il faut utiliser pour cela la commande `LOAD` en précisant le module à charger.

La première méthode a un coût en terme de performances car le module s'exécute à chaque exécution d'une procédure stockée écrite en PL/pgsql. La deuxième méthode rend l'utilisation du profiler un peu plus complexe. Le choix est donc laissé à l'administrateur.

4.9.8 LOG_FUNCTIONS - CONFIGURATION

- 5 paramètres en tout

- À configurer
  - dans Postgresql.conf
  - ou avec SET
- Ne pas oublier `custom_variable_classes`

Les informations de profilage récupérées par log_functions sont envoyées dans les traces de PostgreSQL. Comme cela va générer plus d'écriture, et donc plus de lenteurs, il est possible de configurer chaque trace.

La configuration se fait soit dans le fichier `postgresql.conf` soit avec l'instruction `SET`. Cependant, n'oubliez pas de configurer le paramètre `custom_variable_classes` à cette valeur :

```
custom_variable_classes = 'log_functions'
```

Voici la liste des paramètres et leur utilité :

- `log_functions.log_declare`, à mettre à true pour tracer le moment où PL/pgsql exécute la partie `DECLARE` d 'une procédure stockée ;
- `log_functions.log_function_begin`, à mettre à true pour tracer le moment où PL/pgsql exécute la partie `BEGIN` d 'une procédure stockée ;
- `log_functions.log_function_end`, à mettre à true pour tracer le moment où PL/pgsql exécute la partie `END` d 'une procédure stockée ;
- `log_functions.log_statement_begin`, à mettre à true pour tracer le moment où PL/pgsql commence l'exécution d 'une instruction dans une procédure stockée ;
- `log_functions.log_statement_end`, à mettre à true pour tracer le moment où PL/pgsql termine l'exécution d' une instruction dans une procédure stockée.

Par défaut, seuls `log_statement_begin` et `log_statement_end` sont à false pour éviter la génération de traces trop importantes.

4.9.9 LOG_FUNCTIONS - UTILISATION

- Exécuter des procédures stockées en PL/pgsql
- Lire les journaux applicatifs
  - `grep` très utile

Voici un exemple d'utilisation de cet outil :

```
b2# SELECT incremente(4);
 incremente
------------
          5
```

```
(1 row)

b2# LOAD 'log_functions';
LOAD
b2# SET client_min_messages TO log;
LOG:  duration: 0.136 ms  statement: set client_min_messages to log;
SET
b2# SELECT incremente(4);
LOG:  log_functions, DECLARE, incremente
LOG:  log_functions, BEGIN, incremente
CONTEXT:  PL/pgSQL function "incremente" during function entry
LOG:  valeur de b : 5
LOG:  log_functions, END, incremente
CONTEXT:  PL/pgSQL function "incremente" during function exit
LOG:  duration: 118.332 ms  statement: select incremente(4);
 incremente
------------
          5
(1 row)
```

4.10 PROBLÈMES CONNUS

4.10.1 RELATION INEXISTANTE

« relation with OID XXXX does not exist »

- PL/pgsql met en cache les fonctions et leurs requêtes
- Si une fonction accède à une table temporaire, l'OID de la table est mise en cache
- Si la table est supprimée entre temps, le prochain appel à la fonction fera toujours référence à l'ancien OID
- Solution : utiliser EXECUTE, pour lequel la requête est analysée à chaque exécution
- Plus d'actualité depuis PostgreSQL 8.3.
- Attention avant PostgreSQL 9.3 si deux objets de même nom résident dans des schémas différents

Lors de leur première exécution les fonctions et leurs requêtes sont mise en cache pour accélérer les appels suivants. Si une fonction accède à une table temporaire, l'identifiant de cette relation est lui aussi stocké en cache. Du coup si la table temporaire est supprimée entre deux appels à la fonction, celle-ci fera toujours référence à l'identifiant gardé en

cache, d'où l'erreur « relation with OID XXXX does not exist ». Pour palier à ce problème il est nécessaire d'utiliser une requête dynamique et de faire appel à `EXECUTE`. Depuis la version 8.3 de PostgreSQL `PL/PgSQL` détecte bien que la table n'existe plus et renvoi l'erreur correspondante : `ERROR: relation "..." does not exist`.

Autre problème, l'objet utilisé par la première exécution de la fonction est celui ciblé par le paramètre `search_path` au moment de cette première exécution. L'objet utilisé par la seconde exécution est celui de la première exécution quelle que soit la valeur du paramètre `search_path` ensuite. Ce comportement a été corrigé dans la version 9.3 de PostgreSQL.

4.11 CONCLUSION

- PL/PgSQL est un langage puissant
- Seul inconvénient : sa lenteur par rapport à d'autres PL comme PL/perl ou C
  - PL/perl est très efficace pour les traitements de chaîne uniquement
- Permet néanmoins de traiter la plupart des cas, de façon simple et efficace.

4.11.1 POUR ALLER PLUS LOIN

- Documentation officielle
  - « Chapitre 40. PL/pgsql - Langage de procédures SQL »

Quelques liens utiles dans la documentation de PostgreSQL :

- Chapitre 40. PL/pgsql - Langage de procédures SQL[61]
- Annexe A. Codes d'erreurs de PostgreSQL[62]

4.11.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[61] http://docs.postgresql.fr/current/plpgsql.html
[62] http://docs.postgresql.fr/current/errcodes-appendix.html

4.12 TRAVAUX PRATIQUES

4.12.1 ÉNONCÉS

TP2.1

Ré-écrire la fonction de division pour tracer le problème de division par zéro (vous pouvez aussi utiliser les exceptions).

TP2.2

Tracer dans une table toutes les modifications du champ `nombre` dans `stock`. On veut conserver l'ancienne et la nouvelle valeur. On veut aussi savoir qui a fait la modification et quand.

Interdire la suppression des lignes dans stock. Afficher un message dans les logs dans ce cas.

Afficher aussi un message `NOTICE` quand nombre devient inférieur à 5, et `WARNING` quand il vaut 0.

TP2.3

Interdire à tout le monde, sauf un compte admin, l'accès à la table des logs précédemment créée .

En conséquence, le trigger fonctionne-t-il ? Le cas échéant, le modifier pour qu'il fonctionne.

TP2.4

Lire toute la table `stock` avec un curseur.

Afficher dans les journaux applicatifs toutes les paires `(vin_id, contenant_id)` pour chaque nombre supérieur à l'argument de la fonction.

TP2.5

Ré-écrire la fonction `nb_bouteilles` du TP précédent de façon à ce qu'elle prenne désormais en paramètre d'entrée une liste variable d'années à traiter.

4.12.2 SOLUTIONS

TP2.1 Solution :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS float4 AS
$BODY$
  BEGIN
    RETURN arg1::float4/arg2::float4;
    EXCEPTION WHEN OTHERS THEN
      -- attention, division par zéro
      RAISE LOG 'attention, [%]: %', SQLSTATE, SQLERRM;
      RETURN 'NaN';
  END $BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

```
  cave=# SET client_min_messages TO log;
  SET
  cave=# SELECT division(1,5);
   division
  ----------
        0.2
  (1 ligne)

  cave=# SELECT division(1,0);
  LOG:  attention, [22012]: division par zéro
   division
  ----------
       NaN
  (1 ligne)
```

TP2.2 Solution :

1.

La table de log :

```
  CREATE TABLE log_stock (
  id serial,
  utilisateur text,
  dateheure timestamp,
  operation char(1),
  vin_id integer,
  contenant_id integer,
```

```
  annee integer,
  anciennevaleur integer,
  nouvellevaleur integer);
```

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    -- ce test a pour but de vérifier que le contenu de nombre a bien changé
    -- c'est forcément le cas dans une insertion et dans une suppression
    -- mais il faut tester dans le cas d'une mise à jour en se méfiant
    -- des valeurs NULL
    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'DELETE'
    THEN
      -- cas de la suppression
      v_atracer := true;
```

```
      v_vinid := OLD.vin_id;
      v_contenantid := OLD.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NULL;
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur);
    END IF;

    RETURN NEW;

  END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
  CREATE TRIGGER log_stock_nombre_trig
  AFTER INSERT OR UPDATE OR DELETE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE log_stock_nombre();
```

2.

On commence par supprimer le trigger :

```
  DROP TRIGGER log_stock_nombre_trig ON stock;
```

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
```

```
    v_atracer boolean := false;
  BEGIN

    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_anciennevaleur, v_nouvellevaleur);
    END IF;

    RETURN NEW;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
  CREATE TRIGGER trace_nombre_de_stock
  AFTER INSERT OR UPDATE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE log_stock_nombre();
```

La deuxième fonction trigger :

```
CREATE OR REPLACE FUNCTION empeche_suppr_stock()
  RETURNS TRIGGER AS
$BODY$
  BEGIN

    IF TG_OP = 'DELETE'
    THEN
      RAISE WARNING 'Tentative de suppression du stock (%, %, %)',
                    OLD.vin_id, OLD.contenant_id, OLD.annee;
      RETURN NULL;
    ELSE
      RETURN NEW;
    END IF;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Le deuxième trigger :

```
  CREATE TRIGGER empeche_suppr_stock_trig
  BEFORE DELETE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE empeche_suppr_stock();
```

3.

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
```

```
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    END IF;

    IF v_nouvellevaleur < 1
    THEN
      RAISE WARNING 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
                    v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
    ELSEIF v_nouvellevaleur < 5
    THEN
      RAISE LOG 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
                v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur);
    END IF;

    RETURN NEW;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

Faire des INSERT, DELETE, UPDATE pour jouer avec.

TP2.3 Solution :

```
 CREATE ROLE admin;
 ALTER TABLE log_stock OWNER TO admin;
 ALTER TABLE log_stock_id_seq OWNER TO admin;
 REVOKE ALL ON TABLE log_stock FROM public;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
ERROR:  permission denied for relation log_stock
CONTEXT:  SQL statement "INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur)"
PL/pgSQL function log_stock_nombre() line 45 at SQL statement

 ALTER FUNCTION log_stock_nombre() OWNER TO admin;
 ALTER FUNCTION log_stock_nombre() SECURITY DEFINER;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
  INSERT 0 1
```

Que constatez-vous dans `log_stock` ? (un petit indice : regardez l'utilisateur)

TP2.4 Solution :

```
CREATE OR REPLACE FUNCTION verif_nombre(maxnombre integer)
  RETURNS integer AS
$BODY$
  DECLARE
    v_curseur refcursor;
    v_resultat stock%ROWTYPE;
    v_index integer;
  BEGIN

    v_index := 0;
    OPEN v_curseur FOR SELECT * FROM stock WHERE nombre > maxnombre;
    LOOP
      FETCH v_curseur INTO v_resultat;
      IF NOT FOUND THEN
        EXIT;
      END IF;
      v_index := v_index + 1;
```

```
      RAISE NOTICE 'nombre de (%, %) : % (supérieur à %)',
        v_resultat.vin_id, v_resultat.contenant_id, v_resultat.nombre, maxnombre;
    END LOOP;

  RETURN v_index;

END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage:

```
  SELECT verif_nombre(16);
  INFO:  nombre de (6535, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6538, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6541, 3) : 17 (supérieur à 16)
  [...]
  INFO:  nombre de (6692, 3) : 18 (supérieur à 16)
  INFO:  nombre de (6699, 3) : 17 (supérieur à 16)
   verif_nombre
  --------------
         107935
  (1 ligne)
```

TP2.5

```
CREATE OR REPLACE FUNCTION
  nb_bouteilles(v_typevin text, VARIADIC v_annees integer[])
RETURNS SETOF record
AS $BODY$
  DECLARE
    resultat record;
    i integer;
  BEGIN
    FOREACH i IN ARRAY v_annees
    LOOP
      SELECT INTO resultat i, nb_bouteilles(v_typevin, i);
      RETURN NEXT resultat;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE plpgsql;
```

Exécution:

```
-- ancienne fonction
cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1995)
```

```
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1990 | 5608
  1991 | 5642
  1992 | 5621
  1993 | 5581
  1994 | 5614
  1995 | 5599
(6 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1992, 1994)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1990 | 5608
  1992 | 5621
  1994 | 5614
(3 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1993, 1991)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1993 | 5581
  1991 | 5642
(2 lignes)
```