Le SQL dans un langage de programmation via PostgreSQL

Support de cours IUT de Calais

Ce support est principalement basé sur des notions présentées par Dalibo, une référence française sur PostgreSQL : https://www.dalibo.com

Kevin GUERRIER

Préambule

Lien vers le support de cours : https://bit.ly/2023-iut-postgresql-kguerrier

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 cité 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.

Au commencement…

PostgreSQL a une origine universitaire, et l'origine du nom PostgreSQL remonte au système de gestion de base de données Ingres, développé 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 effet, à cette époque, les solutions disponibles ne répondaient pas toujours aux besoins de performance et de flexibilité grandissants, et les bases de données relationnelles étaient de plus en plus essentielles pour gérer des ensembles de données complexes.

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

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

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

Si nous voulions résumer quelques principes Fondamentaux de PostgreSQL, voici ce que cela pourrait donner :

- **Sécurité des Données** : PostgreSQL garantit que les données modifiées dans une transaction sont écrites sur le disque après le COMMIT, même en cas de crash.
- **Intégrité des Données** : Le moteur assure le respect des contraintes, évitant par exemple l'insertion de données dépassant les limites définies.
- **Respect des normes SQL** : Bien qu'aucun moteur ne soit totalement compatible, PostgreSQL s'efforce d'adhérer autant que possible à la norme SQL et n'accepte de nouvelles syntaxes que si elles respectent cette norme.
- **Portabilité** : PostgreSQL est conçu pour fonctionner sur une large gamme de systèmes d'exploitation et vise à maintenir cette compatibilité dans le futur.
- **Ajout de Fonctionnalités Raisonnable** : Seules les fonctionnalités considérées essentielles et utiles à la majorité sont intégrées, en raison de la limitation de développeurs disponibles.
- **Performances** : Bien que la sécurité et la fiabilité priment, PostgreSQL offre d'excellentes performances, adaptées à des volumes de données massifs avec une configuration matérielle adéquate.
- **Simplicité du Code** : Le code de PostgreSQL est conçu pour être clair et lisible, facilitant ainsi le débogage futur si nécessaire.
- **Documentation Cruciale** : Ce sujet n'est pas à négliger (jamais !). Sans une bonne documentation, peu de personnes pourront comprendre l'utilité précise de chaque fonction, de chaque opérateur. La documentation garantit la clarté et la compréhension de chaque aspect de PostgreSQL.

→ Lien vers la documentation en français (en fonction des versions) : https://docs.postgresql.fr

L'importance des communautés

Les années 2000 marquent l'émergence de communautés locales, organisées formellement ou de manière informelle, dédiées à PostgreSQL. La communauté japonaise (JPUG), créée en 2000, compte plus de 3000 membres et organise des conférences ainsi que la publication de livres et de magazines.

En 2004, naît l'association française PostgreSQL Fr, qui vise à fournir un cadre légal pour la participation à des événements majeurs tels que Solutions Linux et les RMLL (Les Rencontres mondiales du logiciel libre). Elle soutient également l'organisation d'événements comme le pgDay.fr dans différentes villes de France.

En 2006, le PGDG (PostgreSQL Development Group, entité informelle regroupant l'ensemble des contributeurs) rejoint Software in the Public Interest, Inc. (SPI), une organisation à but non lucratif chargée de collecter et redistribuer des financements. Cette démarche renforce la pérennité financière du projet.

À partir de 2008, des associations d'utilisateurs voient le jour pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. Des événements de grande envergure sont organisés, notamment par PostgreSQL UK à Londres et PostgreSQL Fr à Toulouse. De plus, des "sur-groupes" comme PGUS et PostgreSQL Europe apparaissent pour renforcer les groupes locaux.

Dès 2010, on constate plus d'une conférence par mois entièrement dédiée à PostgreSQL dans le monde, témoignant de l'essor croissant de la communauté.

En 2011, l'association canadienne Postgres Community Association of Canada est créée par des membres de la Core Team. Elle est chargée de gérer les éléments associés au nom déposé PostgreSQL, comme le logo et le nom de domaine sur Internet.

En 2017, les Community Guidelines sont introduites pour établir des règles claires pour les communautés internationales, disponibles sur le site officiel de PostgreSQL.

La communauté PostgreSQL a connu une expansion notable, avec des associations locales et internationales favorisant la promotion, le partage d'informations et l'entraide à l'échelle mondiale.

Il est essentiel pour tout étudiant en informatique de ne pas seulement acquérir des connaissances théoriques, mais aussi de s'immerger dans les communautés actives. PostgreSQL offre une riche communauté d'utilisateurs et de développeurs, prête à partager des idées, des astuces et à répondre à vos questions.

N'hésitez pas à explorer les ressources fournies par les associations locales et internationales. Assister à des conférences et participer à des événements organisés par ces associations peut grandement enrichir votre compréhension de PostgreSQL et élargir votre réseau professionnel. De plus, les forums et les listes de diffusion sont des espaces où vous pouvez poser vos questions et bénéficier de l'expérience collective de la communauté.

N'oubliez pas que vous faites partie de cette communauté dynamique et que votre participation est non seulement encouragée, mais également précieuse.

Quelques références…

Si vous n'êtes toujours pas convaincus de l'intérêt de développer vos compétences autours d'un moteur de base de données si "vieux", voici quelques références qui utilisent encore aujourd'hui PostgreSQL (même si dans certains cas, c'est en complément d'autres outils et pour des actions bien spécifiques).

- Météo France[1] utilise PostgreSQL depuis plus d'une décennie pour l'essentiel de ses bases, dont des instances critiques de plusieurs téraoctets :
- L'IGN (Institut Géographique National) [2] utilise PostGIS et PostgreSQL depuis 2006
- La RATP (Régie Autonome des Transports Parisiens)[3] a fait ce choix depuis 2007 également
- La Caisse Nationale d'Allocations Familiales[4] a remplacé ses mainframes par des instances PostgreSQL dès 2010 (4 To et 1 milliard de requêtes par jour).
- Instagram[5] utilise PostgreSQL depuis le début
- Zalando[6] a décrit plusieurs fois son infrastructure PostgreSQL et annonçait en 2018 utiliser pas moins de 300 bases de données en interne et 650 instances dans un cloud AWS. Zalando contribue à la communauté, notamment par son outil de haute disponibilité patroni.
- Le DBA de TripAdvisor[7] témoigne de leur utilisation de PostgreSQL dans l'interview suivante
- Dès 2009, Leroy Merlin[8] migrait vers PostgreSQL des milliers de logiciels de caisse
- La Société Générale[9] a publié son outil de migration d'Oracle à PostgreSQL

De nombreuses autres sociétés participent au Groupe de Travail Inter-Entreprises de PostgreSQLFr : Air France, Carrefour, Leclerc, le CNES, la MSA, la MAIF, PeopleDoc, EDF… et cette liste ne comprends bien évidemment pas celles qui n'ont pas communiqué sur ce sujet ^_^

[1] https://www.postgresql.fr/temoignages/meteo_france
[2] https://www.postgresql.fr/temoignages/ign
[3] https://www.journaldunet.com/solutions/dsi/1013631-la-ratp-integre-postgresql-a-son-systeme-d-information/
[4] https://www.silicon.fr/cnaf-debarrasse-mainframes-149897.html
[5] https://media.postgresql.org/sfpug/instagram_sfpug.pdf
[6] https://www.postgresql.eu/events/pgconfeu2018/schedule/session/2135-highway-to-hell-or-stairway-to-cloud/
[7] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley
[8] https://wiki.postgresql.org/images/6/63/Adeo_PGDay.pdf
[9] https://github.com/societe-generale/code2pg

Enfin, la répartition des parts du marché des SGBD en octobre 2023 :

Rank			DBMS	Database Model	Score		
Oct 2023	Sep 2023	Oct 2022			Oct 2023	Sep 2023	Oct 2022
1.	1.	1.	Oracle	Relational, Multi-model	1261.42	+20.54	+25.05
2.	2.	2.	MySQL	Relational, Multi-model	1133.32	+21.83	-72.06
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	896.88	-5.34	-27.80
4.	4.	4.	PostgreSQL	Relational, Multi-model	638.82	+18.06	+16.10
5.	5.	5.	MongoDB	Document, Multi-model	431.42	-8.00	-54.81
6.	6.	6.	Redis	Key-value, Multi-model	162.96	-0.72	-20.41
7.	7.	7.	Elasticsearch	Search engine, Multi-model	137.15	-1.84	-13.92
8.	8.	8.	IBM Db2	Relational, Multi-model	134.87	-1.85	-14.79
9.	9.	↑ 10.	SQLite	Relational	125.14	-4.06	-12.66
10.	10.	↓ 9.	Microsoft Access	Relational	124.31	-4.25	-13.85
11.	11.	↑ 13.	Snowflake	Relational	123.24	+2.35	+16.51
12.	12.	↓ 11.	Cassandra	Wide column, Multi-model	108.82	-1.24	-9.12
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	99.66	-0.79	-9.65
14.	14.	14.	Splunk	Search engine	92.37	+0.98	-2.28
15.	15.	↑ 16.	Microsoft Azure SQL Database	Relational, Multi-model	80.93	-1.80	-4.03
16.	16.	↓ 15.	Amazon DynamoDB	Multi-model	80.91	+0.00	-7.44
17.	17.	↑ 20.	Databricks	Multi-model	75.82	+0.64	+18.21
18.	18.	↓ 17.	Hive	Relational	69.18	-2.65	-11.42
19.	19.	↓ 18.	Teradata	Relational, Multi-model	58.56	-1.77	-7.51
20.	20.	↑ 22.	Google BigQuery	Relational	56.57	+0.11	+4.12

Source : https://db-engines.com/en/ranking

Les 20 premiers SGBDs restent relativement stables et parmi eux, PostgreSQL est dans le top 5 depuis de nombreuses années !

La boîte à outils PostgreSQL

PostgreSQL n'est qu'un moteur de bases de données. Quand vous l'installez, vous n'avez que ce moteur. Vous disposez bien évidemment des outils en ligne de commande mais aucun outil graphique n'est fourni.

Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

Par choix, nous ne présenterons ici que des logiciels libres et gratuits. Et comme bien souvent dans le monde du Libre, pour chaque problématique, il existe aussi des solutions propriétaires. Ces solutions peuvent parfois apporter des fonctionnalités inédites. Il faut néanmoins considérer que l'offre de la communauté Open-Source répond à la plupart des besoins des utilisateurs de PostgreSQL.

Administration

Au même titre que phpMyAdmin est souvent associé à un moteur MySQL, il existe pgAdmin4 (https://www.pgadmin.org/) qui est un outil d'administration dédié à PostgreSQL, et qui permet aussi de requêter.

Il existe également temBoard (https://labs.dalibo.com/temboard) qui est une console d'administration plus complète. Elle intègre de la supervision, des tableaux de bord, la gestion des sessions en temps réel, du bloat, de la configuration et l'analyse des performances.

Développement

DBeaver (https://dbeaver.io/) est un outil de requêtage courant, utilisable avec de nombreuses bases de données différentes, et adapté à PostgreSQL

Modélisation

Pour la modélisation, pgModeler (https://pgmodeler.io/) est dédié à PostgreSQL. Il permet notamment la modélisation, la rétro ingénierie d'un schéma existant, et la génération de scripts de migration.

Sujet TP - Premiers pas avec PostgreSQL

C'est pas tout ça, mais à un moment, pour apprendre à faire du PostgreSQL, faut faire du PostgreSQL ! Je vous propose donc de passer à un premier atelier pratique autour d'un sujet que vous rencontrez au quotidien : l'université !

L'objectif étant que cette réalisation puisse vous servir et être complétée au fur et à mesure de nos sessions, je vous remercie de porter une attention particulière à leur bonne intégration avant chaque nouveau cours. Si des notions ne sont pas claires, n'hésitez pas à le signaler pour éviter de maintenir des lacunes qui pourraient s'accentuer au fur et à mesure des avancées.

Vous réaliserez ce Projet de manière individuelle et originale (pas de copie, merci ^_^) et me transmettrez les éléments de votre Projet sous forme d'archive sur l'adresse mail : [[guerrier.k @ gmail.com]]. A noter qu'un lien vers l'archive partagée au sein d'un dossier Google Drive est préférable pour éviter toutes les restrictions liées aux sécurités des mails.

/!\ Une dernière chose importante /!\
Merci de respecter le nommage suivant pour vos archives :
2023_PostgreSQL_TP1_NOMPRENOM.zip

Installation du serveur PostgreSQL

Il est possible de télécharger le serveur PostgreSQL à l'adresse suivante : https://www.postgresql.org/download/ en fonction de votre système.

MERCI DE LIRE LES INSTRUCTIONS AVANT DE CLIQUER
SUR TOUS LES LIENS QUE VOUS VOYEZ
OU SUR LES BOUTONS SUIVANT LORS DES INSTALLATIONS !!!

Soyez attentifs notamment aux mots de passe, aux emplacements d'installation, aux outils installés, aux éventuels messages d'erreurs rencontrées, … Pour le moment, pas besoin d'installer d'autres éléments via Stack Builder.

Une fois installé, vous devriez pouvoir saisir la commande suivante dans un terminal pour Connaître la version de PostgreSQL installée :

```
psql --version
```

→ psql (PostgreSQL) 16.0

Si ce n'est pas le cas, pensez à vérifier vos variables d'environnement par exemple.

TP1a - Visite rapide et initialisation

Création de la base de données

Vous utiliserez la première lettre de votre prénom suivi de votre nom et de « db » comme nom de base (ex : kguerrierdb). Par défaut, le nom d'hôte est localhost, l'utilisateur par défaut est postgres, et le mot de passe est celui qui vous a été demandé lors de l'installation.

```
createdb -h <host> -U <login> <nomBase>
```

Ainsi, me concernant, la commande est donc :

```
createdb -h localhost -U postgres kguerrierdb
```

Connexion à votre base de données

Maintenant qu'elle est créée, vous pouvez donc vous y connecter via la commande suivante :

```
psql -h <host> -U <login> <nomBase>
```

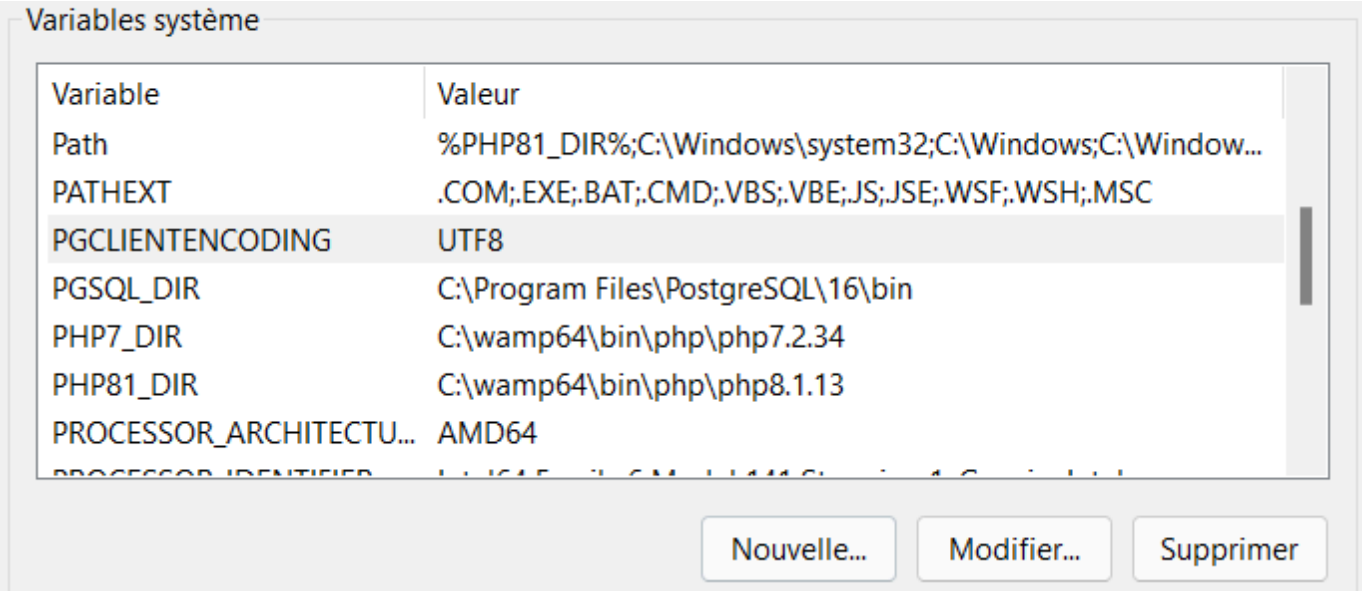
Tout comme pour le moteur MySQL, vous devriez donc arriver sur un prompt vous permettant d'exécuter des commandes :

```
<nomBase>=#
```

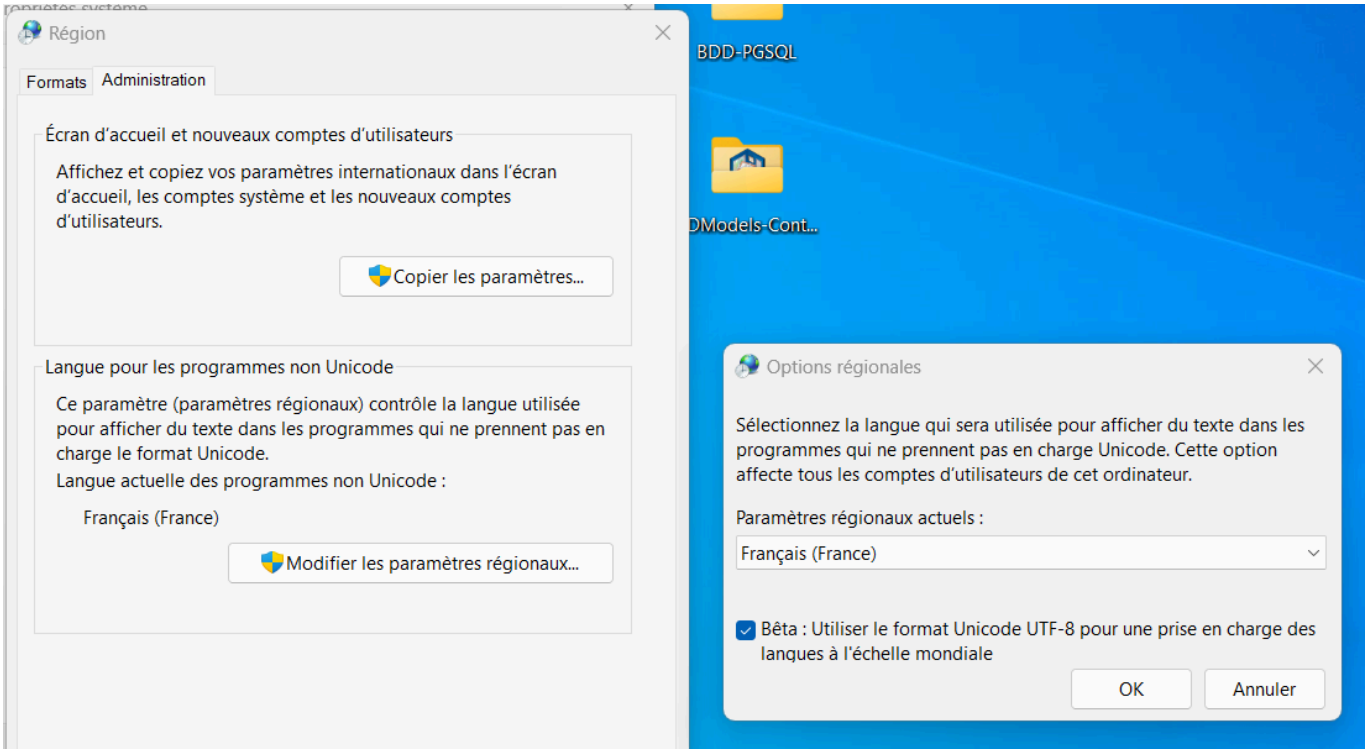
Cas particulier du problème d'encodage

Si un message de différence d'encodage apparaît lors de la connexion, il peut être utile d'adapter votre environnement pour éviter d'avoir des traitements incorrects des caractères, que ce soit à l'affichage ou lors du traitement de fichiers.

Pour cela, la variable d'environnement PGCLIENTENCODING peut être définie.



Pour prendre en charge l'UTF8 sous windows, vous pouvez lancer l'utilitaire intl.cpl, puis dans l'onglet Administration > Modifier les paramètres régionaux, cocher l'utilisation du format unicode UTF-8 pour une prise en charge des langues à l'échelle mondiale.



Après un redémarrage, votre système et le moteur PostgreSQL devraient ainsi fonctionner avec le même encodage.

Quelques commandes PostgreSQL

Voici quelques commandes qui vous seront utiles lors de la manipulation de votre base PostgreSQL. A noter que la tabulation permet l'autocomplétion ;-)

`\?`	Affiche l'aide
`\h [commande]`	Affiche l'aide pour une commande.
`\l`	Affiche la liste des bases de données
`\d`	Affiche la liste des tables de la base de données sur laquelle vous êtes connectée
`\d [NOM OBJET]`	Affiche la description de l'objet
`\o [fichier]`	Redirige les résultats des requêtes dans le fichier. Si le paramètre est manquant, les résultats s'affichent dans le terminal
`\i [fichier]`	Exécuter les commandes du fichier
`\t`	Modification du formatage de sortie (colonnes alignées ou non).
`\H`	Activer/désactiver le formatage de sortie type HTML
`\q`	Quitter PostgreSQL

Pour ces exercices, vous pouvez télécharger une archive via le lien suivant :
https://bit.ly/2023-iut-postgresql-tp1-scripts-kguerrier

Nous allons maintenant créer les tables et les remplir. Pour ce faire, exécuter les commandes contenues dans les fichiers “creation_tables.sql” et “insertion_donnees.sql”. Attention, ne faîtes pas de copier/coller (postgreSQL n'aime pas ça) mais utilisez la commande `\i [fichier]`.

Exercice 1 :

Afficher la liste des bases de données présentes sur votre serveur à l’aide des commandes PostgreSQL adapter et rediriger le résultat dans un fichier NOMPRENOM_tp1_liste_bds.txt pour la version tabulaire et NOMPRENOM_tp1_liste_bds.html pour la version formatée en HTML

Exercice 2 :

Afficher la liste des tables de votre base de données à l’aide de commandes PostgreSQL et les rediriger dans un fichier NOMPRENOM_tp1_liste_tables.txt pour la version tabulaire et NOMPRENOM_tp1_liste_tables.html pour la version formatée en HTML

Exercice 3 :

Pour chaque table, afficher sa description à l’aide de commandes PostgreSQL et rediriger le resultat dans un fichier NOMPRENOM_tp1_description_NOMTABLE.txt pour la version tabulaire et NOMPRENOM_tp1_description_NOMTABLE.html pour la version formatée en HTML.

TP1b - Blocs PL/pgSQL

Plusieurs langages sont disponibles au sein de PostgreSQL, certains officiellement supportés, d’autres portés par des communautés.

Les langages officiellement supportés par le projet sont :

- PL/pgSQL
- PL/Perl12
- PL/Python13 (version 2 et 3)
- PL/Tcl

Pour qu’un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL.

Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l’extension installée dans la base, par exemple :

```
CREATE EXTENSION plperl ;
```

```
CREATE EXTENSION plpython3u ;
```

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n’est pas performant quand il s’agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d’utiliser PL/Perl, voire PL/Python.

Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

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

Dans le cadre de ce cours, nous utiliserons PL/pgSQL.

Fonctions & procédures

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

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

Avant de réaliser votre première fonction, exécutez la commande suivante pour créer une table nommée temp contenant les champs :

- no (INT)
- chiffre (INT)

```
CREATE TABLE temp (no INT, chiffre INT);
```

Exercice 4 :

Pour cette table temp, afficher sa description à l'aide de commandes PostgreSQL et rediriger le resultat dans un fichier NOMPRENOM_tp1_description_NOMTABLE.txt pour la version tabulaire et NOMPRENOM_tp1_description_NOMTABLE.html pour la version formatée en HTML.

Exercice 5 :

Créer une procédure appelée mon_insert qui a pour objet d'alimenter notre table temps.

```
kguerrierdb=# CREATE PROCEDURE mon_insert()
kguerrierdb-# LANGUAGE plpgsql
kguerrierdb-# as $$
kguerrierdb$# DECLARE
kguerrierdb$# x integer := 100;
kguerrierdb$# BEGIN
kguerrierdb$# FOR i IN 1..10 LOOP
kguerrierdb$# INSERT INTO temp VALUES (i,x);
kguerrierdb$# x:=x+100;
kguerrierdb$# END LOOP;
kguerrierdb$# END;
kguerrierdb$# $$;
```

Si tout s'est correctement déroulé, vous devriez voir le message de confirmation CREATE PROCEDURE.

Vous pouvez alors vérifier que votre procédure est bien présente dans la liste des fonctions visibles grâce à la commande \df

`\df`	Affiche la liste des fonctions de la base de données sur laquelle vous êtes connectée

Afficher la liste des fonctions de votre base de données à l'aide de commandes PostgreSQL et la rediriger dans un fichier NOMPRENOM_tp1_liste_fonctions.txt pour la version tabulaire et NOMPRENOM_tp1_liste_fonctions.html pour la version formatée en HTML.

Avant de l'exécuter, vérifier le contenu de votre table temp en exécutant la commande SQL :

```
SELECT * FROM temp;
```

Le résultat devrait bien être vide ^_^

Appelez ensuite votre procédure afin qu'elle "nourrisse" votre table :

```
CALL mon_insert();
```

Et vérifiez le nouveau contenu de votre table temp qui devrait alors avoir à présent 10 lignes :

```
SELECT * FROM temp;
```

Afficher le résultat de cette requête et le rediriger dans un fichier NOMPRENOM_tp1_data_temp.txt pour la version tabulaire et NOMPRENOM_tp1_data_temp.html pour la version formatée en HTML.

TP1c - Le schéma de la base IUT

A présent, vous allez travailler sur la base que vous avez créée grâce aux fichiers importés. Toutes les requêtes vont donc faire référence aux tables auxquelles vous avez déjà accès :

- etudiants,
- enseignants,
- matieres,
- modules,
- faire_cours,
- epreuves
- avoir_note.

Description des tables créées

Retrouvez la structure de ces différentes tables à l'aide de la commande « \d ».

Cette observation de structure effectuée, réalisez un schéma expliquant le fonctionnement de cette base (schéma montrant les tables, clés primaires et clés étrangères et les liens entre les tables) puis déduisez en le MCD. Vous réaliserez ce schéma sur le logiciel libre analyseSI (https://launchpad.net/analysesi). Une fois le fichier jar téléchargé, vous pouvez l'exécuter directement en lançant la commande java -jar analyseSI-0.80.jar

Il sera régulièrement utilisé en tant que référence dans la suite des cours donc prêtez-y toute l'attention nécessaire.

Une fois terminé, vous enregistrerez une version png de votre schéma dans un fichier NOMPRENOM_tp1_schema.png

Et voilà ! Vous avez fait vos premiers pas en PostgreSQL ! Félicitations !!!

Réalisez une archive nommée 2023_PostgreSQL_TP1_NOMPRENOM.zip et contenant les fichiers générés lors des exercices ci-dessus. Ces fichiers ne devraient pas être bloqués par les moteurs de mails donc vous devriez pouvoir les joindre directement en pièce jointe. En cas d'erreur, stockez là dans un dossier partagé (par exemple sur Google Drive) et transmettez-moi le lien de téléchargement par mail ;-)

Nettoyage :

Vous pouvez supprimer la procédure que vous avez créé pour ce TP en exécutant la commande suivante : .

```
DROP PROCEDURE mon_insert() ;
```

Tout comme vous pouvez également supprimer la table temps créée pour ce TP en exécutant la commande suivante : .

```
DROP TABLE temp ;
```

Gestion des auto incréments sur les tables

Afin d'activer les autoincréments sur les tables créées par les fichiers SQL, il est nécessaire d'executer les commandes suivantes :

```
CREATE SEQUENCE etu_id_seq OWNED BY etudiants.numetu;
ALTER TABLE etudiants ALTER COLUMN numetu SET DEFAULT nextval('etu_id_seq');
UPDATE etudiants SET numetu = nextval('etu_id_seq');

CREATE SEQUENCE ens_id_seq OWNED BY enseignants.numens;
ALTER TABLE enseignants ALTER COLUMN numens SET DEFAULT nextval('ens_id_seq');
UPDATE enseignants SET numens = nextval('ens_id_seq');

CREATE SEQUENCE mod_id_seq OWNED BY modules.nummod;
ALTER TABLE modules ALTER COLUMN nummod SET DEFAULT nextval('mod_id_seq');
UPDATE modules SET nummod = nextval('mod_id_seq');

CREATE SEQUENCE mat_id_seq OWNED BY matieres.nummat;
ALTER TABLE matieres ALTER COLUMN nummat SET DEFAULT nextval('mat_id_seq');
UPDATE matieres SET nummat = nextval('mat_id_seq');

CREATE SEQUENCE epr_id_seq OWNED BY epreuves.numepr;
ALTER TABLE epreuves ALTER COLUMN numepr SET DEFAULT nextval('epr_id_seq');
UPDATE epreuves SET numepr = nextval('epr_id_seq');
```

Retour sur les fonctions et opérateurs sous PostgreSQL

Cette partie est issue de la documentation officielle de postgreSQL :
https://docs.postgresql.fr/16/functions.html

Les opérateurs

Les opérateurs logiques habituels sont disponibles :

- `boolean AND boolean → boolean`
- `boolean OR boolean → boolean`
- `NOT boolean → boolean`

Le SQL utilise un système logique en trois valeurs, avec true, false et null, qui représente une valeur « inconnue ».

Voici les tables de vérités :

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT** *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Les opérateurs AND et OR sont commutatifs, c'est-à-dire que vous pouvez intervertir les opérandes gauche et droit sans changer le résultat. (Néanmoins, il n'est pas garanti que l'opérande gauche soit évaluée avant l'opérande droit !).

Fonctions et opérateurs mathématiques

Les opérateurs mathématiques sont fournis pour de nombreux types de données au sein de PostgreSQL.

Pour les voir en détails (et pour voir le comportement de ces opérateurs lorsqu'ils sont utilisés avec des types sans conventions mathématiques standards comme par exemple les types dates/heures), reportez-vous à la documentation officielle :

https://docs.postgresql.fr/16/functions-math.html

Fonctions et opérateurs sur des chaînes de caractères

De la même manière, les opérateurs permettant de manipuler les chaînes de caractères sont fournis et vous pouvez retrouver l'ensemble de leurs documentation à cette adresse :

https://docs.postgresql.fr/16//functions-string.html

Fonctions de formatage de type de données

Les fonctions de formatage de PostgreSQL forment un ensemble puissant d'outils pour convertir différents types de données (date/heure, entier, nombre à virgule flottante, numérique) vers des chaînes formatées et pour convertir de chaînes formatées vers des types de données spécifiques. Pour plus d'informations, reportez-vous à la documentation officielle :

https://docs.postgresql.fr/16/functions-formatting.html

Fonctions et opérateurs pour date/heure

La manipulation de données date/heure peut également être faite directement par le biais d'opérateurs et de fonctions en PostgreSQL. Reportez-vous à cette documentation pour voir précisément leur fonctionnement :

https://docs.postgresql.fr/16/functions-datetime.html

Fonctions et opérateurs de géométrie

Même si leur utilisation est plus spécifique, les types géométriques (point, box, lseg, line, path, polygon et circle) disposent d'un ensemble de fonctions et d'opérateurs de support natifs. et vous pouvez retrouver l'ensemble de leurs documentation à cette adresse :

https://docs.postgresql.fr/16/functions-geometry.html

Fonctions et opérateurs tableau

Plus fréquents, les types tableau disposent également d'un ensemble de fonctions et d'opérateurs.

Les opérateurs de comparaison comparent le contenu des tableaux éléments par éléments, en utilisant la fonction de comparaison B-tree par défaut pour le type de données de l'élément, et trient en se basant sur la première différence rencontrée. Dans les tableaux multi-dimensionnels, les éléments sont visités dans l'ordre des lignes. Si les contenus de deux tableaux sont identiques mais que leur dimension est différente, la première différence dans l'information de dimension détermine l'ordre de tri.

Vous pouvez retrouver l'ensemble de leurs documentation à cette adresse :

https://docs.postgresql.fr/16/functions-array.html

Bonnes pratiques : Utilisation des Index dans PostgreSQL

Un index est une structure de données essentielle utilisée et maintenue par le système de gestion de base de données (SGBD). Sa principale fonction est d'accélérer la recherche de données dans une table, ce qui en fait un outil précieux pour optimiser les performances des requêtes.

Les points clés

- Accélération des Opérations
  - L'index permet d'accélérer diverses opérations effectuées par le SGBD telles que la recherche, le tri, les jointures et l'agrégation de données. En créant un index sur les colonnes les plus fréquemment utilisées dans les requêtes, vous pouvez significativement améliorer la réactivité de votre système.

- Construction de l'Index
  - Un index est construit en parallèle d'une table en se basant sur la valeur d'un ou plusieurs champs. Cela signifie que lors de la création de l'index, le SGBD organise les données en fonction des valeurs dans ces champs spécifiques, facilitant ainsi la recherche ultérieure.

- Impact sur les Performances d'Insertion et de Mise à Jour
  - Il est important de noter que bien que les index améliorent les performances des requêtes, ils peuvent ralentir les opérations d'insertion et de mise à jour. Cela est dû au fait que chaque fois qu'une entrée est ajoutée ou mise à jour dans la table, l'index doit être reconstruit pour refléter les nouvelles données.

- Gestion des Index en Cas d'Introductions Massives de Données
  - En cas d'introduction massive de données, il peut être judicieux de supprimer temporairement l'index, de procéder aux opérations d'insertion ou de mise à jour, puis de recréer l'index par la suite. Cette stratégie permet d'optimiser le processus et d'éviter des ralentissements excessifs.

PostgreSQL offre une variété d'algorithmes d'indexation pour optimiser les performances des requêtes. Le choix de l'index dépend du type de données et des requêtes que vous exécutez.

Index Btree

L'index Btree est souvent utilisé pour l'indexation standard en raison de ses performances élevées et de sa polyvalence. Il convient bien à la plupart des situations et offre de nombreuses possibilités pour l'optimisation des requêtes.

Exemple : Supposons que nous ayons une table "utilisateurs" avec une colonne "nom" et que nous voulions rechercher un utilisateur par son nom. Un index Btree sur la colonne "nom" permettrait d'accélérer considérablement cette recherche.

```
CREATE INDEX idx_nom ON utilisateurs (nom);
```

Index Hash

Les index hash sont utilisés pour les comparaisons d'égalité, en particulier avec de grandes chaînes de caractères. Cependant, avant la version 10, leur utilisation était déconseillée en raison de l'absence de journalisation, ce qui pouvait entraîner des problèmes en cas de crash ou de restauration.

Exemple : Si nous avons une table de hachages contenant des mots de passe pour l'authentification, un index hash sur la colonne des mots de passe (nommée “h_passwords”) pourrait accélérer la vérification de l'authenticité d'un utilisateur.

```
CREATE INDEX idx_hpswd ON utilisateurs USING hash (h_passwords);
```

Index GIN et GIST

Les index GIN (Generalized Inverted Index) et GIST (Generalized Search Tree) sont conçus pour gérer de grands volumes de données complexes et multidimensionnelles. Ils sont utilisés pour des types de données spécifiques tels que les indexations textuelles, géométriques, géographiques ou de tableaux.

Exemple : Dans une application de recherche de texte, un index GIN sur le contenu des mots clés enregistrés pour les articles permettrait des recherches rapides et efficaces.

```
CREATE INDEX idx_contents ON articles USING gin (keywords);
```

Exemple : Dans une application de recherche de points d’intérêts à proximité de coordonnées à partir d’une base géographique contenant les latitude / longitude (champ coordonnées de type “point”), un index GIST sur ce champ permettrait des recherches rapides et efficaces.

```
CREATE INDEX idx_coords ON points_repere USING gist (coordonnees);
```

Autres types d’index

Index BRIN

Les index BRIN (Block Range INdexes) sont particulièrement adaptés aux grandes tables où les données sont fortement corrélées en termes d'emplacement physique sur les disques. Ils sont très compacts et offrent d'excellentes performances dans ces scénarios.

Exemple : Si nous avons une grande table d'enregistrements de capteurs avec des horodatages, un index BRIN sur la colonne des horodatages pourrait être très efficace.

Index Bloom

Les index Bloom sont utilisés pour indexer de nombreuses colonnes interrogées simultanément. Ils nécessitent l'ajout d'une extension nommée "bloom".

Exemple : Dans une base de données de catalogues de produits en ligne, un index Bloom sur les attributs comme la couleur, la taille et le prix permettrait des recherches rapides lors de la navigation des utilisateurs.

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

Exemple : Si nous avons une table de noms d'utilisateur et que nous voulons effectuer des recherches partielles basées sur des expressions régulières, le module pg_trgm pourrait être utilisé pour accélérer ces recherches.

Bonnes pratiques : Utilisation des Contraintes

En plus de maintenir la qualité des données, les contraintes fournissent des informations précieuses au planificateur de requêtes. Par exemple, une contrainte d'unicité permet au planificateur de savoir qu'il peut ignorer certaines jointures dans certains cas, améliorant ainsi les performances des requêtes.

Les principales types de contraintes sont les suivantes :

- La présence obligatoire : NOT NULL
  - → id_client NOT NULL
- L'unicité : UNIQUE
  - → id_client UNIQUE
- Les clés primaires (equiv. UNIQUE && NOT NULL)
  - → PRIMARY KEY (id_client)
- Les clés étrangères qui renvoie à une clés d'une autre table
  - → produit_id REFERENCES produits(id_produit)

Sujet TP - Index, Contraintes et Manipulation de fonctions natives

Retournons à présent à notre atelier pratique autour du sujet de : l'université.

Petit rappel : Vous réaliserez ce Projet de manière individuelle et originale (pas de copie, merci ^_^) et me transmettrez les éléments de votre Projet sous forme d'archive sur l'adresse mail : [[guerrier.k @ gmail.com]]. A noter qu'un lien vers l'archive partagée au sein d'un dossier Google Drive est préférable pour éviter toutes les restrictions liées aux sécurités des mails.

/!\ Une dernière chose importante /!\
Merci de respecter le nommage suivant pour vos archives :
2023_PostgreSQL_TP2_NOMPRENOM.zip

Exercice 1 :

Analysez la base IUT pour identifier et lister les contraintes déjà présentes dans chaque table.

Vous créerez un fichier texte NOMPRENOM_tp2_liste_contraintes.txt dans lequel vous indiquerez pour chaque table les contraintes présentes et le ou les champs concernés .

Complément : Si vous avez d'autres contraintes que vous trouvez pertinentes, listez les et précisez les commandes qui permettrait de les ajouter.

Exercice 2 :

Analysez la base IUT pour identifier et lister les indexs déjà présents dans chaque table.

Vous créerez un fichier texte NOMPRENOM_tp2_liste_indexs.txt dans lequel vous indiquerez pour chaque table les indexes présents et le ou les champs concernés .

Complément : Si vous avez d'autres indexes que vous trouvez pertinents, listez les et précisez les commandes qui permettrait de les ajouter.

Exercice 3 :

A partir du schéma de la base IUT, réalisez les requêtes suivantes sur votre base de données :

- Lister les nom, prénom et âge des étudiants.
- Lister les nom, prénom des étudiants (dans une seule colonne avec des chaînes concaténées) ainsi que le mois de leur naissance.
- Lister les nom, prénom et notes des étudiants ainsi que les épreuves concernées.
- Lister les nom, prénom des enseignants ainsi que l'ensemble des matières qu'ils proposent (en utilisant la fonction d'agrégat de chaîne de caractères)
- Lister les nom, coef des modules ainsi que l'ensemble des matières qui les composent (en utilisant la fonction d'agrégat de chaîne de caractères)
-

Vous noterez l'ensemble des requêtes dans un fichier texte NOMPRENOM_tp2_liste_requetes.txt.

Vous placerez ces fichiers dans un répertoire NOMPRENOM_tp2/indexsFonctions/ qui sera à placer dans l'archive finale.

Et dans un tout autre contexte, vous avez en charge de résoudre cet escape game : Mystery Crimes - Enquête

[Cet exercice est repris en totalité du cours de Bénédicte TALON ;-)]

Pour cet exercice, vous créerez une base de données nommée « _mysterycrimesdb » prefixé par vos initiales comme nom de base (ex : kg_mysterycrimesdb).

Trouverez-vous qui a fait le coup ? L'inspecteur Talon vous laisse les dossiers sur une enquête qu'elle n'a su résoudre... Mais elle n'est pas très ordonnée dans ses dossiers ni dans ses notes...

Pour commencer, téléchargez l'archive à cette adresse.Vous y trouverez des scripts et des éléments d'enquête.

INFORMATIONS des QRCode :

- **Le coupable est majeur depuis plus de 10 ans**
- **yeux verts**
- **Le revenu annuel du coupable est supérieur à 25000**

A vous d'effectuer les ajustements nécessaires pour faire les exercices suivants.

Exercice 1 :

Prenez connaissance des scripts disponibles en vous assurant de la cohérence de ces derniers. Ajustez si besoin ce que vous estimez devoir l'être, et injectez-les dans votre base de données.

Exercice 2 :

Pour chaque table, afficher sa description à l'aide de commandes PostgreSQL et rediriger l'ensemble du résultat dans un fichier NOMPRENOM_tp2MC_descriptions_tables.txt pour la version tabulaire et NOMPRENOM_tp2MC_descriptions_tables.html pour la version formatée en HTML.

Exercice 3 :

A partir de ces éléments, réaliser le MCD de la base de données (au format pdf ou png) que vous nommerez NOMPRENOM_tp2MC_mcd.

Exercice 4 :

Prenez connaissance des éléments d'enquêtes, résolvez l'énigme en trouvant le coupable et insérez votre conclusion dans la table solution, en précisant votre nom complet dans le champ Nom_etudiant.

Vous créerez un fichier texte NOMPRENOM_tp2MC_demarche.txt dans lequel vous indiquerez le processus qui vous a mené à cette conclusion.

Exercice 5 :

Vous créerez un fichier PDF NOMPRENOM_tp2MC_psql.pdf dans lequel vous indiquerez les commandes psql utilisées dans votre démarche, et les captures d'écran des résultats qui vous ont permis de résoudre cette énigme.

Vous placerez ces fichiers dans un répertoire NOMPRENOM_tp2/mysteryCrimes/ qui sera à placer dans l'archive finale.

Réalisez une archive nommée 2023_PostgreSQL_TP2_NOMPRENOM.zip et contenant votre répertoire NOMPRENOM_tp2. Ce dernier doit bien évidemment contenir les répertoires indexsFonctions et mysteryCrimes, qui auront été alimentés par les fichiers générés lors des exercices ci-dessus. Ces fichiers ne devraient pas être bloqués par les moteurs de mails donc vous devriez pouvoir les joindre directement en pièce jointe. En cas d'erreur, stockez là dans un dossier partagé (par exemple sur Google Drive) et transmettez-moi le lien de téléchargement par mail ;-)

Utilisation de PostgreSQL pour la persistence de données issues de formulaires écrits en PHP

Nous allons réaliser à présent différentes pages web qui permettent de manipuler des données et utiliser PostgreSQL comme SGBD. Manipuler ces éléments de manière est plus aisée que par le biais de la console ;-)

Avant tout : Quelle technologie utiliser ?

Il y a beaucoup de manières de connecter une base de données PostgreSQL, mais certaines sont plus complexes que d'autres. Pour commencer, nous allons limiter les difficultés et utiliser un serveur Apache avec le module PHP.

Pré-requis :

- WampServeur fonctionnel (ou tout autre ensemble Apache / PHP que vous maitrisez ;-))
- Un répertoire alias dédié au projet et fonctionnel
- l'extension php_pdo_pgsql pour PHP activée dans votre php.ini
- l'extension php_pgsql pour PHP activée dans votre php.ini

Rappel sur la création d'un fichier de configuration pour apache :

Dans [apacheConf]/conf-available/myphp_pgsqltp.conf

```
Alias /myphpPgsqltp "C:\Users\guerr\Desktop\BDD-PGSQL\myphp_pgsqlTP/"
<Directory "C:\Users\guerr\Desktop\BDD-PGSQL\myphp_pgsqlTP/">
    Options +Indexes +FollowSymLinks +MultiViews
    AllowOverride all
    Require local
</Directory>
```

et créer un lien dans [apacheConf]/conf-enabled vers ../conf-available/myphp_pgsqltp.conf
Redémarrez votre serveur apache pour que la configuration soit prise en compte

D'après cette configuration, vous devriez accéder à votre fichier en passant par http://localhost/myphpPgsqltp

Vérification de la configuration

Le moteur PHP fournit une fonction qui permet d'afficher la configuration actuellement chargée. Cette fonction s'appelle phpinfo(). Un fichier est parfois disponible nativement selon les outils (par exemple sur wamp), et si ce n'est pas le cas, vous pouvez créer un fichier phpinfo.php contenant le code suivant :

```php
<?php
  echo phpinfo();
```

L'affichage de cette page doit donner le résultat ci-après :

PHP Version 8.1.13

System	Windows NT DESKTOP-IVJL5EV 10.0 build 22621 (Windows 11) AMD64
Build Date	Nov 22 2022 15:45:24
Build System	Microsoft Windows Server 2019 Datacenter [10.0.17763]
Compiler	Visual C++ 2019
Architecture	x64
Configure Command	cscript /nologo /e:jscript configure.js "--enable-snapshot-build" "--enable-debug-pack" "--with-pdo-oci=..\..\..\..\instantclient\sdk,shared" "--with-oci8-19=..\..\..\..\instantclient\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	*no value*
Loaded Configuration File	C:\wamp64\bin\apache\apache2.4.54.2\bin\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20210902
PHP Extension	20210902
Zend Extension	420210902

avec notamment les sections pgsql

pgsql

PostgreSQL Support	enabled
PostgreSQL (libpq) Version	11.4
Multibyte character support	enabled
Active Persistent Links	0
Active Links	0

Directive	Local Value	Master Value
pgsql.allow_persistent	On	On
pgsql.auto_reset_persistent	Off	Off
pgsql.ignore_notice	Off	Off
pgsql.log_notice	Off	Off
pgsql.max_links	Unlimited	Unlimited
pgsql.max_persistent	Unlimited	Unlimited

et pdo_pgsql

pdo_pgsql

PDO Driver for PostgreSQL	enabled
PostgreSQL(libpq) Version	11.4

Rapide présentation du fonctionnement de PHP

Le PHP est un langage basé sur le principe de Client - Serveur. Les affichages sont gérés côté client, les traitements sont gérés côté serveur, et la communication entre les 2 passe par un serveur web.

Aussi, le PHP est vraiment un langage lié avec le HTML, le CSS, et autres scripts (même s'il existe des traitements PHP autonomes en ligne de commande…).

Notion de formulaire

Dans les applications PHP, nous allons permettre l'interaction avec les utilisateurs par le biais de formulaires, contenant des champs de saisie variés qui seront transmis d'une page à l'autre, que ce soit par le biais de l'adresse (méthode GET) soit par une encapsulation dans le système de requête HTTP (méthode POST).

Les données globales de l'application sont généralement initialisées en amont (dépendantes, variables d'environnements, authentification, …), puis les informations de formulaires sont récupérées et traitées, et enfin, les affichages sont générés.

Ainsi, on peut voir une application PHP de manière procédurale étant donné que le fonctionnement est séquentiel, tout en bénéficiant des possibilités liées aux classes d'objets, qui peuvent être vues comme des conteneurs dont l'ensemble des fonctions sont en place et qu'il est possible de manipuler plus aisément sans avoir à tout recoder à chaque appel.

Utilisation du pilote PDO_PGSQL

PDO_PGSQL est un pilote qui implémente l'interface de PHP Data Objects (PDO) pour autoriser l'accès de PHP aux bases de données PostgreSQL.

La connexion à une base de données passe par le DSN (Data Source Name) de PDO_PGSQL, défini de la manière suivante (le séparateur utilisé est le caractère ";") :

- Préfixe DSN : Le préfixe DSN est pgsql:.
- host : L'hôte sur lequel le serveur de base de données se situe.
- port : L'hôte sur lequel le serveur de base de données se situe.
- dbname : Le nom de la base de données.
- user : Le nom de l'utilisateur pour la connexion.
- password : Le mot de passe de l'utilisateur pour la connexion.
- sslmode : Le mode SSL. Les valeurs prises en charge et leur signification sont listées dans la section » Documentation PostgreSQL.

Exemple :

pgsql:host=localhost;port=5432;dbname=kguerrierdb;user=postgres;password=mypass

Vérification de la connexion

Dans un fichier test_connexion.php, le code suivant :

```php
<?php
try {
    $dbh = new
        PDO(
            "pgsql:host=localhost;port=5432;dbname=kguerrierdb",
            "postgres",
            "motDePasse"
        );
    echo 'Connexion OK !!<br />';
} catch (PDOException $e) {
    $dbh = null;
    echo 'ERREUR Base de données: ' . $e->getMessage();
}
if ($dbh) {
    echo 'Ici on peut donc faire des requêtes';
}
```

devrait vous retourner le résultat suivant :

Connexion OK !!
Ici on peut donc faire des requêtes

Ainsi, pour réaliser une requête qui récupère tous les enregistrements de la table etudiants, on peut utiliser le code suivant :

```php
<?php
// ouverture de la connexion
$host = "localhost";
$dbname = "kguerrierdb";
$port = "5432";
$username = 'postgres';
$password = 'motDePasse';

$dsn = "pgsql:host=" . $host . ";port=" . $port . ";dbname=" . $dbname;
$options = [
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
];
try {
    $db = new PDO($dsn, $username, $password, $options);
} catch (PDOException $e) {
    $db = null;
    echo 'ERREUR Base de données: ' . $e->getMessage();
}

if ($db) {
    // création de la requête
    $sql = "SELECT * FROM etudiants";

    // envoi de la requête et récupération du résultat
    $listeEtu = $db->query($sql)->fetchAll();
?>
    <ul>
        <?php foreach ($listeEtu as $etudiant) { ?>
            <li>
                <?= "{$etudiant['nometu']} - {$etudiant['prenometu']}"; ?>
            </li>
        <?php } ?>
    </ul>
<?php
}
```

Pour rappel, l'utilisation de l'option PDO::FETCH_ASSOC précise que les données seront retournées sous forme de tableau associatif.

Sujet TP - Consultation PostgreSQL via PHP

Retournons à présent à notre atelier pratique autour du sujet de : l'université.

Petit rappel : Vous réaliserez ce Projet de manière individuelle et originale (pas de copie, merci ^_^) et me transmettrez les éléments de votre Projet sous forme d'archive sur l'adresse mail : [[guerrier.k @ gmail.com]]. A noter qu'un lien vers l'archive partagée au sein d'un dossier Google Drive est préférable pour éviter toutes les restrictions liées aux sécurités des mails.

/!\ Une dernière chose importante /!\
Merci de respecter le nommage suivant pour vos archives :
2023_PostgreSQL_TP3_NOMPRENOM.zip

Exercice 1 :

Maintenant que vous maitrisez parfaitement les structures de votre base IUT, réalisez les pages php suivantes qui permettent de lister les informations présentes dans votre base de données :

- liste_etudiants.php : qui liste le contenu de la table etudiants
- liste_enseignants.php : qui liste le contenu de la table enseignants
- liste_matieres.php : qui liste le contenu de la table matière
- liste_epreuves.php : qui liste le contenu de la table epreuves
- liste_modules.php : qui liste le contenu de la table modules

Exercice 2 :

Pour chaque entrée présente dans ces listes, vous afficherez l'ensemble des informations de l'élément concerné, ainsi qu'un lien qui renvoie à la liste :

- affiche_etudiant.php : qui affiche les informations d'un étudiant
- affiche_enseignant.php : qui affiche les informations d'un enseignants
- affiche_matiere.php : qui affiche les informations d'une matière
- affiche_epreuve.php : qui affiche les informations d'une epreuves
- affiche_module.php : qui affiche les informations d'un modules

Réalisez une archive nommée 2023_PostgreSQL_TP3_NOMPRENOM.zip et contenant les fichiers générés lors des exercices ci-dessus. Ces fichiers sont des scripts et **sont donc fortement susceptibles d'être bloqués par les moteurs de mails** donc préférez le stockage dans un dossier partagé (par exemple sur Google Drive) et transmettez-moi le lien de téléchargement par mail ;-)

PL/pgSQL : Les bases

Qu'est-ce qu'un PL ?

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

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

- PL/PgSQL[10] est intégré par défaut dans toute nouvelle base (de par sa présence dans la base modèle template1) ;
- PL/Tcl[11] (existe en version trusted et untrusted) ;
- PL/Perl[12] (existe en version trusted et untrusted) ;
- PL/Python[13] (uniquement en version untrusted).

D'autres langages PL sont accessibles en tant qu'extensions tierces. Les plus stables sont mentionnés dans la documentation[14] , comme PL/Java[15] ou PL/R[16] . Ils réclament généralement d'installer les bibliothèques du langage sur le serveur.

Une liste plus large est par ailleurs disponible sur le wiki PostgreSQL[17], Il en ressort qu'au moins 16 langages sont disponibles, dont 10 installables en production. De plus, il est possible d'en ajouter d'autres, comme décrit dans la documentation[18].

.

Langages trusted vs untrusted

Les langages de confiance (*trusted*) ne peuvent accéder qu'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 utilisables pour compromettre le système. PL/pgSQL est l'exemple typique.

Mais de ce fait, ils offrent moins de possibilités que les autres langages.

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

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

[10] https://docs.postgresql.fr/current/plpgsql.html
[11] https://docs.postgresql.fr/current/pltcl.html
[12] https://docs.postgresql.fr/current/plperl.html
[13] https://docs.postgresql.fr/current/plpython.html
[14] https://docs.postgresql.fr/current/external-pl.html
[15] https://tada.github.io/pljava
[16] https://github.com/postgres-plr/plr
[17] https://wiki.postgresql.org/wiki/PL_Matrix
[18] https://docs.postgresql.fr/current/plhandler.html

Les langages PL de PostgreSQL

La question se pose souvent de placer la logique applicative du côté de la base, dans un langage PL, ou des clients. Il peut y avoir de nombreuses raisons en faveur de la première option. 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.

Nous aborderons donc dans cette partie les avantages de cette pratique.

Centralisation du code :

Si plusieurs applications ont potentiellement besoin d'opérer un même traitement, à fortiori dans des langages différents, porter cette logique dans la base réduit d'autant les risques de bugs et facilite la maintenance.

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

Performances :

Le code s'exécute localement, directement dans le moteur de la base. Il n'y a donc pas tous les changements de contexte et échanges de messages réseaux dus à l'exécution de nombreux ordres SQL consécutifs. L'impact de la latence due au trafic réseau de la base au client est souvent sous-estimée.

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

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

Simplicité :

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

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

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, là aussi utilisables dans la base.

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.

Intérêts de PL/pgSQL en particulier

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

Le PL/pgSQL 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 procédural de SQL.

- PL/pgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux rajoutés par l'utilisateur. Il peut les manipuler de façon transparente.
- PL/pgSQL est trusted. Par défaut, tous les utilisateurs peuvent donc créer des routines dans ce langage. Vous pouvez toujours soit supprimer le langage, soit retirer les droits à un utilisateur sur ce langage (cf. la commande SQL REVOKE).
- PL/pgSQL est donc raisonnablement facile à utiliser : il y a peu de complications, peu de pièges, et il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

Les autres langages PL ont toujours leur intérêt !

Les langages PL « autres », comme PL/perl[19] 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, possèdent des structures avancées comme des tables de hachage, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leur version untrusted, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un service web par exemple, ou d'écrire des données dans un fichier externe.

Il existe des langages PL spécialisés. Le plus emblématique d'entre eux est PL/R[20]. R est un langage utilisé par les statisticiens pour manipuler de gros jeux de données. PL/R permet donc d'effectuer ces traitements R directement en base, traitements qui seraient très pénibles à écrire dans d'autres langages, et avec une latence dans le transfert des données.

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[21], directement. Elles seront compilées à l'extérieur de PostgreSQL, en respectant un certain formalisme, puis seront chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour.

Mais attention :

Toute erreur dans le code C est susceptible d'accéder à toute la mémoire visible par le processus PostgreSQL qui l'exécute, et donc de corrompre les données. Il est donc conseillé de ne faire ceci qu'en dernière extrémité.

Le gros défaut est simple et commun à tous ces langages : ils ne sont pas spécialement conçus 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 ; de même en C. L'accès aux données est assez fastidieux au niveau syntaxique, comparé à PL/pgSQL.

[19] https://docs.postgresql.fr/current/plperl-builtins.html
[20] https://github.com/postgres-plr/plr/blob/master/userguide.md
[21] https://docs.postgresql.fr/current/xfunc-c.html

Un autre problème des langages PL (autre que C et PL/pgSQL), 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 aux 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).

Routines / Procédures stockées / Fonctions

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

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

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

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

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

Les Triggers

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

La fonction trigger doit être définie avant que le trigger lui-même puisse être créé. La fonction trigger doit être déclarée comme une fonction ne prenant aucun argument et retournant un type trigger. Une fois qu'une fonction trigger est créée, le trigger est créé avec CREATE TRIGGER. La même fonction trigger est utilisable par plusieurs triggers.

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.

À noter que les problématiques de visibilité et de volatilité depuis un trigger sont assez complexes dès lors que l'on lit ou modifie les données. Voir la documentation[22] pour plus de détails à ce sujet.

Types de trigger : ligne ou instruction

PostgreSQL offre des déclencheurs par ligne et par instruction. Avec un déclencheur mode ligne, la fonction du déclencheur est appelée une fois pour chaque ligne affectée par l'instruction qui a lancé le déclencheur.

Au contraire, un déclencheur mode instruction n'est appelé qu'une seule fois lorsqu'une instruction appropriée est exécutée, quel que soit le nombre de lignes affectées par cette instruction. En particulier, une instruction n'affectant aucune ligne résultera toujours en l'exécution de tout déclencheur mode instruction applicable.

Ces deux types sont quelque fois appelés respectivement des déclencheurs niveau ligne et des déclencheurs niveau instruction. Les triggers sur TRUNCATE peuvent seulement être définis au niveau instruction, et non pas au niveau ligne.

Fonctions trigger : variables

Des variables spécifiques sont disponibles pour les triggers en mode ligne et permettent d'accéder aux ANCIENNES et NOUVELLES valeurs des champs. Ces variables sont :

- 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

Elles sont accessibles par une notation pointée, par exemple : NEW.champ1 pour accéder à la nouvelle valeur de champ1

Pour les triggers en mode instruction, la version 10 propose les tables de transition.

[22] https://docs.postgresql.fr/current/trigger-datachanges.html

D'autres variables sont également disponibles, à savoir :

- TG_NAME (*obs. TG_RELNAME*) :
  - 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 trigger
- TG_OP :
  - chaîne valant INSERT, UPDATE, DELETE, TRUNCATE suivant l'opération qui a déclenché le trigger
- 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
- 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 ;
```

Et il est possible d'enregistrer des triggers faisant appel à cette fonction de la manière suivante :

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

Fonctions trigger : retour

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

Pour pouvoir être utilisée comme valeur de retour dans la fonction (avec RETURN), une variable doit être de structure identique à celle de la table sur laquelle le trigger a été déclenché.

Les variables spéciales OLD (ancienne valeur avant application de l'action à l'origine du déclenchement) et NEW (nouvelle valeur après application de l'action) sont également disponibles, utilisables et même modifiables.

La valeur de retour d'un trigger de type ligne (ROW) déclenché avant l'opération (BEFORE) peut changer complètement l'effet de la commande ayant déclenché le trigger.

Par exemple, il est possible d'annuler complètement l'action sans erreur (et d'empêcher également tout déclenchement ultérieur d'autres triggers pour cette même action) en retournant NULL.

Il est également possible de changer les valeurs de la nouvelle ligne créée par une action INSERT ou UPDATE en retournant une des valeurs différentes de NEW (ou en modifiant NEW directement).

Attention, dans le cas d'une fonction trigger BEFORE déclenchée par une action DELETE, il faut prendre en compte que NEW contient NULL, en conséquence RETURN NEW; provoquera l'annulation du DELETE ! Dans ce cas, si on désire laisser l'action inchangée, la convention est de faire un RETURN OLD;.

En revanche, la valeur de retour utilisée n'a pas d'effet dans les cas des triggers ROW et AFTER, et des triggers STATEMENT. À noter que bien que la valeur de retour soit ignorée dans ce cas, 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.

Fonctions trigger : Exemple d'horodatage

Si on considère une table disposant de la structure suivante :

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

alors le trigger suivant permettra de mettre à jour constamment les informations d'horodatage :

```
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;
```

Fonctions trigger : Tables de transition

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.

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

```
CREATE TABLE archives (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 $$
  BEGIN
    INSERT INTO archives (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
    RETURN null;
  END
$$;
```

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 ligne dans t1

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

et maintenant : supprimons-les (ne cherchez pas, c'est pour l'exemple !) :

```
DELETE FROM t1;
```

Dans cet exemple, la suppression avec le trigger prend 2 secondes.

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

DELETE FROM t1;
Time: 2141.871 ms (00:02.142)
```

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 archives;
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
EXPLAIN (ANALYZE) DELETE FROM t1;
```

```
TRUNCATE archives;

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

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) :

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

DROP TRIGGER tr1 ON t1;

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

TRUNCATE archives;
TRUNCATE t1;

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

DELETE FROM t1;
```

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

DELETE FROM t1;
Time: 8445.697 ms (00:08.446)
```

```
TRUNCATE archives;
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;
EXPLAIN (ANALYZE) DELETE FROM t1;
```

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

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

Paramètres et Retour des Fonctions et Procédures

Version minimaliste

```
kguerrierdb=# CREATE FUNCTION addition (entier1 integer, entier2 integer)
kguerrierdb-# RETURNS integer
kguerrierdb-# AS ...
```

Ceci une forme de fonction très simple (et très courante) : deux paramètres en entrée (implicitement en entrée seulement), et une valeur en retour.

Dans le corps de la fonction, il est aussi possible d'utiliser une notation numérotée au lieu des noms de paramètre : le premier argument a pour nom $1, le deuxième $2, etc. C'est à éviter.

Tous les types sont utilisables, y compris les types définis par l'utilisateur. En dehors des types natifs de PostgreSQL, PL/pgSQL ajoute des types de paramètres spécifiques pour faciliter l'écriture des routines.

Paramètres IN, OUT & retour

```
kguerrierdb=# CREATE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int,
OUT annee int)
kguerrierdb-# AS ...
```

Si le mode d'un argument est omis, IN est la valeur implicite : la valeur en entrée ne sera pas modifiée.

Un paramètre OUT sera modifié. S'il s'agit d'une variable d'un bloc PL appelant, sa valeur sera modifiée. Un paramètre INOUT est un paramètre en entrée mais sera également modifié.

Dans le corps d'une fonction, 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, comme dans l'exemple plus bas.

L'option VARIADIC 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 comme renvoyant une table (cf. clause RETURNS TABLE).

Jusque PostgreSQL 13 inclus, les procédures ne supportent pas les arguments OUT, seulement IN et INOUT.

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.

Type en retour : 1 valeur simple

Le type de retour (clause RETURNS dans l'entête) est obligatoire pour les fonctions et interdit pour les procédures.

Avant la version 11, il n'était pas possible de créer une procédure, mais il était possible de créer une fonction se comportant globalement comme une procédure en utilisant le type de retour void.

Depuis le corps de la fonction, le résultat est renvoyé par un appel à RETURN (PL/pgSQL) ou SELECT (SQL).

Type en retour : 1 lignes, plusieurs champs

S'il y a besoin de renvoyer plusieurs valeurs à la fois, une possibilité est de renvoyer un type composé défini auparavant.

Une alternative courante est d'utiliser plusieurs paramètres OUT (et pas de clause RETURN dans l'entête) pour obtenir un enregistrement composite :

```
CREATE OR REPLACE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int,
OUT annee int)
AS $$
  SELECT extract (day FROM d)::int, extract(month FROM d)::int, extract (year FROM
d)::int
$$
LANGUAGE SQL;
```

Suite à quoi, la commande :

```
kguerrierdb=# SELECT * FROM explose_date ('11-12-2023');
```

devrait retourner le résultat :

```
jour  | mois | annee
------+------ +-------
11    |    12 |  2023
```

(Noter que l'exemple ci-dessus est en simple SQL.)

La clause TABLE est une autre alternative, sans doute plus claire. Cet exemple devient alors, toujours en pur SQL :

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

Type en retour : Retour multi-lignes

Pour renvoyer plusieurs lignes, la première possibilité est de déclarer un type de retour SETOF. Cet exemple utilise RETURN NEXT pour renvoyer les lignes une à une :

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

Suite à quoi, la commande :

```
SELECT * FROM liste_entiers_setof (3) ;
```

devrait retourner le résultat :

```
liste_entiers_setof
-----------------------
                      1
                      2
                      3
(3 lignes)
```

S'il y plusieurs champs à renvoyer, une possibilité est d'utiliser un type dédié (composé), qu'il faudra cependant créer auparavant.

```
CREATE TYPE pgt AS (schemaname text, tablename text) ;
```

L'exemple suivant utilise aussi un RETURN QUERY pour éviter d'itérer sur toutes les lignes du résultat :

```
CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS SETOF pgt
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
              FROM pg_tables WHERE tableowner=p_owner
              ORDER BY tablename ;
END ;
$$ ;
```

Suite à quoi, la commande :

```
SELECT * FROM tables_by_owner ('pgbench') ;
```

devrait retourner le résultat :

```
schemaname | tablename
-----------------+-----------------
public          | pgbench_accounts
public          | pgbench_branches
public          | pgbench_history
public          | pgbench_tellers
(4 lignes)
```

On a vu que la clause TABLE permet de renvoyer plusieurs champs. Or, elle implique aussi SETOF, et les deux exemples ci-dessus peuvent devenir :

```
CREATE OR REPLACE FUNCTION liste_entiers_table (limite int)
RETURNS TABLE (j int)
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    j = i ;
    RETURN NEXT ; -- renvoie la valeur de j en cours
  END LOOP;
END $$
LANGUAGE plpgsql;
```

Suite à quoi, la commande :

```
SELECT * FROM liste_entiers_table (3) ;
```

devrait retourner le résultat :

```
j
1
2
3
(3 lignes)
```

Noter ici que le nom du champ retourné dépend du nom de la variable utilisée, et n'est pas forcément le nom de la fonction.

En effet, chaque appel à RETURN NEXT retourne un enregistrement composé d'une copie de toutes les variables, au moment de l'appel à RETURN NEXT.

En utilisant le type de retour TABLE, la seconde fonction (tables_by_owner) serait donc la suivante :

```
CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS TABLE (schemaname text, tablename text)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
               FROM pg_tables WHERE tableowner=p_owner
               ORDER BY tablename ;
END ;
$$ ;
```

Les fonctions avec RETURN QUERY ou RETURN NEXT stockent tout le résultat avant de le retourner en bloc. Le paramètre work_mem permet de définir la mémoire utilisée avant l'utilisation d'un fichier temporaire, qui a bien sûr un impact sur les performances.

Si RETURNS TABLE est peut-être le plus souple et le plus clair, le choix entre toutes ces méthodes reste une affaire de goût, ou de compatibilité avec du code ancien ou converti d'un produit existant.

Quand plusieurs lignes sont renvoyées, tout est conservé en mémoire jusqu'à la fin de la fonction. Donc, si beaucoup de données sont renvoyées, cela peut poser des problèmes de latence, voire de mémoire.

Exemple de Fonction

Fonction PL/pgSQL simple

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

Suite à quoi, la commande :

```
kguerrierdb=# SELECT addition (1,2);
```

devrait retourner le résultat 3

Fonction PL/pgSQL utilisant la base

Bien évidemment, il est également possible d'interagir avec des données présentes dans la base PostgreSQL (heureusement ^_^) ! Dans cet exemple, on récupère l'estimation du nombre de lignes actives d'une table passée en paramètres.

L'intérêt majeur du PL/pgSQL et du SQL sur les autres langages est la facilité d'accès aux données. Ici, un simple SELECT <champ> INTO <variable> suffit à récupérer une valeur depuis une table dans une variable.

```
kguerrierdb=# CREATE OR REPLACE FUNCTION nb_lignes_table (sch text, tbl text)
kguerrierdb-# RETURNS bigint
kguerrierdb-# STABLE
kguerrierdb$# AS $$
kguerrierdb$# DECLARE n bigint ;
kguerrierdb$# BEGIN
kguerrierdb$# SELECT n_live_tup
kguerrierdb$# INTO n
kguerrierdb$# FROM pg_stat_user_tables
kguerrierdb$# WHERE schemaname = sch AND relname = tbl ;
kguerrierdb$# RETURN n ;
kguerrierdb$# END;
kguerrierdb$# $$;
kguerrierdb$# LANGUAGE plpgsql ;
```

Il serait alors possible d'appeler cette fonction par la commande :

```
kguerrierdb=# SELECT nb_lignes_table ('public', 'pgbench_accounts');
```

Exemple de procédure

La procédure ci-dessous tronque des tables de la base d'exemple pgbench, et annule si dry_run est vrai.

Les procédures sont récentes dans PostgreSQL (à partir de la version 11). Elles sont à utiliser quand on n'attend pas de résultat en retour. Surtout, elles permettent de gérer les transactions (COMMIT, ROLLBACK), ce qui ne peut se faire dans des fonctions, même si celles-ci peuvent modifier les données.

```
kguerrierdb=# CREATE OR REPLACE PROCEDURE vide_tables (dry_run BOOLEAN)
kguerrierdb$# AS $$
kguerrierdb$# BEGIN
kguerrierdb$# TRUNCATE TABLE pgbench_history ;
kguerrierdb$# TRUNCATE TABLE pgbench_accounts CASCADE ;
kguerrierdb$# TRUNCATE TABLE pgbench_tellers CASCADE ;
kguerrierdb$# TRUNCATE TABLE pgbench_branches CASCADE ;
kguerrierdb$# IF dry_run THEN
kguerrierdb$# ROLLBACK ;
kguerrierdb$# END IF ;
kguerrierdb$# END;
kguerrierdb$# $$;
kguerrierdb$# LANGUAGE plpgsql ;
```

Sujet TP - Environnement JRCanDev

Retournons à présent à notre atelier pratique autour du sujet de : l'université.

Petit rappel : Vous réaliserez ce Projet de manière individuelle et originale (pas de copie, merci ^_^) et me transmettrez les éléments de votre Projet sous forme d'archive sur l'adresse mail : [[guerrier.k @ gmail.com]]. A noter qu'un lien vers l'archive partagée au sein d'un dossier Google Drive est préférable pour éviter toutes les restrictions liées aux sécurités des mails.

/!\ Une dernière chose importante /!\
Merci de respecter le nommage suivant pour vos archives :
2023_PostgreSQL_TP4_NOMPRENOM.zip

Exercice 1 - PHP :

Sur la même base que vous avez réalisé les fiches (étudiants / enseignants / matières / epreuves / modules), vous, réalisez les pages php d'ajout et de modification pour les éléments. Par exemple pour les etudiants :

- ajoute_etudiant.php
- modifie_etudiant.php
- supprime_etudiant.php

De même, vous ajouterez la possibilité de consulter les éléments connexes. Par exemple pour un étudiant :

- etudiant_notes.php qui permet de manager les notes de l'étudiant en cours (CRUD)

ou encore pour un enseignant :

- enseignant_cours.php qui permet de manager les cours de l'enseignant en cours (CRUD)

ou encore pour une matière :

- matiere_enseignants.php qui permet de manager les enseignants d'une matière (CRUD)
- matiere_epreuves.php qui permet de manager les épreuves d'une matière (CRUD)

ou encore pour une épreuve :

- epreuve_notes.php qui permet de manager les notes d'une matière (CRUD)

[… Bref, Je pense que vous avez compris l'idée ^_^]

Exercice 2 - PHP :

L'objectif est d'avoir facilement accès aux différents classements des étudiants ainsi que leurs moyennes, que ce soit de manière globale mais également dans les différentes matières.

Ces informations seront accessibles depuis la fiche d'un étudiant mais également depuis des interfaces dédiées aux classements :

- classement_etudiants.php : affiche les classements des étudiants toutes matières confondues
- classement_matieres.php : affiche les classements des étudiants pour cette matière

[… Bref, Je pense que vous avez compris l'idée ^_^]

Vous constaterez que ces classements sont réalisés à partir de valeurs calculées (soit au sein de vos requêtes, soit au sein de votre code PHP), mais qu'ils sont donc relancés à chaque visite. c'est donc l'occasion de mettre en application des routines en PL/PGSQL pour éviter cette surconsommation de ressources.

Exercice 3 - PL/PGSQL :

Pour chaque requête et son explication, vous enregistrerez dans un fichier nommé NOMPRENOM_tp4_plpgsql.sql vos réponses :

1. Écrire une procédure « affich_note » permettant l'affichage des notes sous la forme :
   roblin lea a eu 15 à interro anglais.
   athur leon a eu 8 à interro anglais.
   minol luc a eu 7 à interro anglais.
   Etc.
2. Écrire une procédure « ajout_enseignant » prenant en paramètre le nom et prénom de l'enseignant. Les autres champs seront laissés vides à l'exception de la date d'embauche qui sera fixée à la date courante.
3. Idem pour « ajout_etudiant ». La date d'entrée sera fixée au 1er septembre de l'année en cours.
4. Écrire une procédure « ajout_note » prenant en paramètre le nom et prénom d'un étudiant, le libellé de l'épreuve ainsi que la note obtenue.
5. Écrire une procédure « modif_note » prenant les mêmes paramètres que la procédure « ajout_note ».

Exercice 4 - PHP :

Vous ajusterez votre code applicatif client PHP afin qu'il prenne en compte les routines écrites à l'exercice 3. Afin de pouvoir évaluer vos réponses de l'exercice 1, vous commenterez les lignes précédemment écrites pour qu'elle reste accessibles.

Exercice 5 - PL/PGSQL :

L'objectif est donc de mettre en place les tables relatives aux différents classements, et de réaliser les routines (fonctions / procédures / …) qui permettront de mettre à jour automatiquement les classements lorsqu'une nouvelle note est ajoutée.

Travail Individuel à me transmettre par email
pour le 31/01/2024 20H au plus tard.
Contenu de l'archive (sur un drive par exemple) :
L'ensemble de vos codes PHP ainsi que
vos fichiers SQL contenant les fonctions / procédures

A partir du prochain TP, vous réaliserez votre Projet en groupe de 3 ou 4 personnes, et vous demanderez la préparation d'un environnement dédié à votre projet sur JRCanDev (via Discord) . Vous me transmettrez par email l'adresse de votre projet sur l'adresse mail : [[guerrier.k @ gmail.com]].

L'environnement qui vous sera fourni sera composé : d'un serveur web avec PHP, d'un serveur PostgreSQL, d'une interface vous permettant d'interagir avec votre serveur PostgreSQL, ainsi qu'une interface vous permettant d'uploader / d'éditer vos fichiers PHP.

PL/pgSQL, on continue

Revenons sur les codes demandés sur le sujet précédent à l'exercice 3 :

1. Écrire une procédure « affich_note » permettant l'affichage des notes sous la forme :
   roblin lea a eu 15 à interro anglais.
   athur leon a eu 8 à interro anglais.
   minol luc a eu 7 à interro anglais.
   Etc.

Cette demande pourrait finalement se faire sous forme de fonction dont le code pourrait donc être le suivant :

```
CREATE OR REPLACE FUNCTION affich_note()
RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
  req CURSOR FOR
    SELECT nometu, prenometu, note, libepr
    FROM etudiants E
    INNER JOIN avoir_note A ON E.numetu=A.numetu
    INNER JOIN epreuves Ep ON A.numepr=Ep.numepr
    ORDER BY e.numetu;
  vnometu etudiants.nometu%type;
  vprenometu etudiants.prenometu%type;
  vnote avoir_note.note%type;
  vlibepr epreuves.libepr%type;
BEGIN
  OPEN req;
  LOOP
    FETCH req INTO vnometu, vprenometu, vnote, vlibepr;
    EXIT WHEN NOT FOUND;
    RAISE INFO '% % a eu % à %',vnometu, vprenometu, vnote, vlibepr;
  END LOOP;
CLOSE req;
END;
$$ ;
```

Nous y retrouvons notamment :

- la déclaration d'un curseur qui permettra de parcourir l'ensemble des enregistrements d'une requête ;
- la déclaration de plusieurs variables dont le type est établie en fonction du type d'un champs dans une table (via la syntaxe table.champs%type) ;
- l'ouverture du curseur, son parcours et l'attribution de chaque ligne dans les variables ;
- la condition de sortie de boucle ;
- la levée d'information pour chaque enregistrement trouvé, formaté comme demandé ;
- et enfin, la fermeture du curseur ;

2. Écrire une procédure « ajout_enseignant » prenant en paramètre le nom et prénom de l'enseignant. Les autres champs seront laissés vides à l'exception de la date d'embauche qui sera fixée à la date courante.

   Le code pour cette procédure pourrait donc être le suivant :

```
CREATE OR REPLACE PROCEDURE ajout_enseignant(
    nom in enseignants.nomens%type,
    prenom in enseignants.preens%type
)
LANGUAGE plpgsql
AS $$
DECLARE
  date_embauche date;
BEGIN
    IF EXISTS (SELECT 1 FROM enseignants WHERE nomens = nom AND preens = prenom ) THEN
        RAISE EXCEPTION USING MESSAGE = 'Cet enseignant existe déjà';
    END IF;
    date_embauche = NOW();
    INSERT INTO enseignants (nomens, preens, datembens)
    VALUES (nom, prenom, date_embauche);
END;
$$;
```

Nous y retrouvons notamment :

- la déclaration des paramètres de la fonction dont le type est établie en fonction du type d'un champs dans une table (via la syntaxe table.champs%type) ;
- un test d'existence sur utilisant une requête vérifiant si il y aurait un doublon sur le nom, prénom d'un enseignant ;
- la levée d'exception le cas échéant utilisant un message personnalisé ;
- la définition de la valeur d'une variable à partir d'une fonction ;
- l'exécution d'une requête d'insertion avec les données préparées ;

3. Idem pour « ajout_etudiant ». La date d'entrée étant fixée au 1er septembre de l'année en cours.

```
CREATE OR REPLACE PROCEDURE ajout_etudiant(
    nom in etudiants.nometu%type,
    prenom in etudiants.prenometu%type
)
LANGUAGE plpgsql
AS $$
DECLARE
  now date;
  day_inscription int;
  date_inscription date;
BEGIN
    IF EXISTS (SELECT 1 FROM etudiants WHERE nometu = nom AND prenometu = prenom ) THEN
        RAISE EXCEPTION USING MESSAGE = 'Cet étudiant existe déjà';
    END IF;
    now = NOW();
    year_ins = (SELECT EXTRACT(YEAR FROM NOW()));
    date_inscription = (SELECT make_date(year_ins, 9, 1));
    IF (date_inscription > now) THEN
      date_inscription = (SELECT make_date((year_ins-1), 9, 1));
    END IF;
    INSERT INTO etudiants (nometu, prenometu, datentetu, annetu)
    VALUES (nom, prenom, date_inscription, 1);
END;
$$;
```

Nous y retrouvons notamment :

- la déclaration des paramètres de la fonction dont le type est établie en fonction du type d'un champs dans une table (via la syntaxe table.champs%type) ;
- un test d'existence sur utilisant une requête vérifiant si il y aurait un doublon sur le nom, prénom d'un étudiant ;
- la levée d'exception le cas échéant utilisant un message personnalisé ;
- la définition de la valeur d'une variable à partir d'une fonction ;
- la définition de la valeur d'une variable à partir du résultat d'une fonction ;
- la modification de la valeur d'une variable sous condition ;
- l'exécution d'une requête d'insertion avec les données préparées ;

4. Écrire une procédure « ajout_note » prenant en paramètre le nom et prénom d'un étudiant, le libellé de l'épreuve ainsi que la note obtenue.

```
CREATE OR REPLACE PROCEDURE ajout_note(
  vnometu in etudiants.nometu%type,
  vprenometu in etudiants.prenometu%type,
  vnote in avoir_note.note%type,
  vlibepr in epreuves.libepr%type)
LANGUAGE plpgsql
AS $$
DECLARE
  vnumetu etudiants.numetu%type;
  vnumepr epreuves.numepr%type;
BEGIN
  SELECT numetu INTO vnumetu FROM etudiants
  WHERE upper(nometu) = upper(vnometu) AND upper(prenometu) = upper(vprenometu);

  IF vnumetu IS NULL THEN
    RAISE SQLSTATE '10000';
  END IF;

  SELECT numepr INTO vnumepr FROM epreuves WHERE upper(libepr) = upper(vlibepr);

  IF vnumepr IS NULL THEN
    RAISE SQLSTATE '20000';
  END IF;

  INSERT INTO avoir_note VALUES(vnumetu, vnumepr, vnote);
  EXCEPTION
  WHEN SQLSTATE '10000' THEN
    RAISE INFO 'etudiant non trouve';
  WHEN SQLSTATE '20000' THEN
    RAISE INFO 'epreuve non trouvée';
END;
$$;
```

Nous y retrouvons notamment :

- l'affectation directe du résultat d'un champs d'une requête dans une variable
- la levée de différents états SQL internes sur des codes personnalisés en fonctions de tests spécifiques ;
- l'exécution d'une requête d'insertion avec les données préparées ;
- la définition des exceptions relatives aux codes d'état SQL internes précédemments établis et levant des informations sous forme de message personnalisé ;

5. Écrire une procédure « modif_note » prenant les mêmes paramètres que la procédure « ajout_note ».

```
CREATE OR REPLACE PROCEDURE modif_note(
  vnometu in etudiants.nometu%type,
  vprenometu in etudiants.prenometu%type,
  vnote in avoir_note.note%type,
  vlibepr in epreuves.libepr%type)
LANGUAGE plpgsql
AS
$$
DECLARE
  vnumetu etudiants.numetu%type;
  vnumepr epreuves.numepr%type;
  ancienne_note avoir_note.note%type;
BEGIN
  SELECT numetu INTO vnumetu
  FROM etudiants
  WHERE upper(nometu) = upper(vnometu)
  AND upper(prenometu) = upper(vprenometu);
  IF vnumetu IS NULL THEN
    RAISE SQLSTATE '10000';
  END IF;

  SELECT numepr INTO vnumepr
  FROM epreuves
  WHERE upper(libepr) = upper(vlibepr);
  IF vnumepr IS NULL THEN
    RAISE SQLSTATE '20000';
  END IF;

  SELECT note INTO ancienne_note
  FROM avoir_note
  WHERE numetu=vnumetu
  AND numepr=vnumepr;
  IF ancienne_note IS NULL THEN
    RAISE SQLSTATE '30000';
  END IF;

  UPDATE avoir_note
  SET note=vnote
  WHERE numetu=vnumetu
  AND numepr=vnumepr;

  EXCEPTION
```

```
  WHEN SQLSTATE '10000' THEN
    RAISE INFO 'etudiant non trouve';
  WHEN SQLSTATE '20000' THEN
    RAISE INFO 'epreuve non trouvée';
  WHEN SQLSTATE '30000' THEN
    RAISE INFO 'pas de note dans cette epreuve pour cet étudiant';
END;
$$;
```

Avec les lignes de code que nous avons vu dans les précédentes procédures, vous devriez avoir une parfaite compréhension de ces éléments.

Attardons nous à présent sur la manière dont nous pouvons utiliser ces éléments côté PHP.

Voici tout d'abord un exemple utilisant une requête préparée classique relative à l'ajout d'un nouvel enseignant :

```php
// récupération des données du formulaire
$nomens = GETPOST('nomens') ?? "";
$preens = GETPOST('preens') ?? "";
$foncens = GETPOST('foncens') ?? "";
$adrens = GETPOST('adrens') ?? "";
$vilens = GETPOST('vilens') ?? "";
$cpens = GETPOST('cpens') ?? "";
$telens = GETPOST('telens') ?? "";
$datnaiens = GETPOST('datnaiens') ?? "";
$datembens = GETPOST('datembens') ?? "";

// création de la requête
$sql = "INSERT INTO enseignants (nomens, preens, foncens, adrens, vilens, cpens, telens, datnaiens, datembens)
VALUES (:nomens, :preens, :foncens, :adrens, :vilens, :cpens, :telens, :datnaiens, :datembens)";

try {
    // envoi de la requête
    $statement = $db->prepare($sql);
    $statement->execute(compact(['nomens', 'preens', 'foncens', 'adrens', 'vilens', 'cpens', 'telens', 'datnaiens', 'datembens']));
    $id = $db->lastInsertId();
    // redirection
    header("location:afficher_enseignant.php?id=" . $id);
    exit();
} catch (Exception $e) {
    $_SESSION['mesgs']['errors'][] = $e->getMessage();
}
header("location:ajouter_enseignant.php");
exit();
```

Maintenant que nous avons écrit une procédure, il faut l'utiliser à la place de l'exécution directe de la requête. Cela pourrait se transcrire par les lignes de code suivantes :

```php
// récupération des données du formulaire
$nomens = GETPOST('nomens') ?? "";
$preens = GETPOST('preens') ?? "";
// Version avec Appel Fonction via CALL
$sql = "CALL ajout_enseignant(:nomens, :preens)";

try {
    // envoi de la requête
    $statement = $db->prepare($sql);
    $statement->execute(compact(['nomens', 'preens']));
    $id = $db->lastInsertId();
    // redirection
    header("location:afficher_enseignant.php?id=" . $id);
    exit();
} catch (Exception $e) {
    $_SESSION['mesgs']['errors'][] = $e->getMessage();
}
header("location:ajouter_enseignant.php");
exit();
```

De cette manière, l'évolution de la "Transaction" relative à l'ajout d'un enseignant peut être réalisée directement dans la procédure au sein de PostgreSQL sans que cela n'impacte le côté applicatif (tant que l'entête de soit pas modifiée bien évidemment).

Concernant l'ajout d'étudiant, le principe est identique. Voici un exemple utilisant une requête préparée classique :

```php
// récupération des données du formulaire
$nometu = GETPOST('nometu') ?? "";
$prenometu = GETPOST('prenometu') ?? "";
$adretu = GETPOST('adretu') ?? "";
$viletu = GETPOST('viletu') ?? "";
$cpetu = GETPOST('cpetu') ?? "";
$teletu = GETPOST('teletu') ?? "";
$datentetu = GETPOST('datentetu') ?? "";
$annetu = GETPOST('annetu') ?? "";
$remetu = GETPOST('remetu') ?? "";
$sexetu = GETPOST('sexetu') ?? "";
$datnaietu = GETPOST('datnaietu') ?? "";

$sql = "INSERT INTO etudiants (nometu, prenometu, adretu, viletu, cpetu, teletu, datentetu, annetu, remetu, sexetu, datnaietu)
VALUES (:nometu, :prenometu, :adretu, :viletu, :cpetu, :teletu, :datentetu, :annetu, :remetu, :sexetu, :datnaietu)";
try {
    $statement = $db->prepare($sql);
    $statement->execute(compact(
       ['nometu', 'prenometu', 'adretu', 'viletu', 'cpetu', 'teletu', 'datentetu', 'annetu', 'remetu', 'sexetu', 'datnaietu']
    ));
    $id = $db->lastInsertId();
    header("location:afficher_etudiant.php?id=" . $id);
    exit();
} catch (Exception $e) {
    $_SESSION['mesgs']['errors'][] = $e->getMessage();
}
header("location:ajouter_etudiant.php");
exit();
```

Et concernant l'appel à cette procédure. Cela pourrait se transcrire par les lignes de code suivantes :

```php
// récupération des données du formulaire
$nometu = GETPOST('nometu') ?? "";
$prenometu = GETPOST('prenometu') ?? "";

// Version avec Appel Fonction via CALL
$sql = "CALL ajout_etudiant(:nometu, :prenometu)";

try {
    // envoi de la requête
    $statement = $db->prepare($sql);
    $statement->execute(compact(['nometu', 'prenometu']));
    $id = $db->lastInsertId();
    // redirection
    header("location:afficher_etudiant.php?id=" . $id);
    exit();
} catch (Exception $e) {
    $_SESSION['mesgs']['errors'][] = $e->getMessage();
}
header("location:ajouter_etudiant.php");
exit();
```

De cette manière, l'évolution de la "Transaction" relative à l'ajout d'un étudiant peut être réalisée directement dans la procédure au sein de PostgreSQL sans que cela n'impacte le côté applicatif (tant que l'entête de soit pas modifiée bien évidemment).

Concernant l'ajout de note sur une épreuve, nous avons un peu plus de modifications. Voici tout d'abord une possibilité d'ajout des notes à partir d'une épreuve. Le formulaire serait alors celui-ci, présent sous la fiche d'une épreuve :

#	Nom	Prenom	Annee	Note
	roblin	lea	1	10
	macarthur	leon	1	11
	minol	luc	1	12
	bagnole	sophie	1	13
	bury	marc	1	14
	vendraux	marc	1	15
	vendermaele	helene	1	16
	GUERRIER	Kevin	1	17

Enregistrer

et le formulaire retourné serait alors de la forme

```
Array
(
    [id] => 6
    [action] => ajouter_note
    [note] => Array
        (
            [1] => 10
            [2] => 11
            [3] => 12
            [4] => 13
            [5] => 14
            [6] => 15
            [7] => 16
            [20] => 17
        )

    [b_save] => Enregistrer
)
```

- l'index id précise l'identifiant de l'épreuve,
- le tableau note est indexé par l'identifiant de l'étudiant :
    - et chaque entrée correspond à la note qui lui est rattachée.

De cette manière, nous avons donc la partie de code suivante qui permettra d'enregistrer ces éléments :

```php
// Traitement de l'ajout de notes
if ($action == "ajouter_note" && GETPOST('b_save') == "Enregistrer") {
    if ($db) {
        // récupération des données du formulaire
        $list_notes = GETPOST('note') ?? array();
        // création de la requête
        $sql = "INSERT INTO avoir_note (numetu, numepr, note) VALUES (:numetu, :numepr, :note)";
        try {
            $statement = $db->prepare($sql);
            // envoi de la requête pour chaque note renseignée
            foreach ($list_notes as $numetu => $note) {
                $statement->execute(compact(['numetu', 'numepr', 'note']));
            }
        } catch (Exception $e) {
            $_SESSION['mesgs']['errors'][] = $e->getMessage();
        }
        header("location:afficher_epreuve.php" . $numepr);
        exit();
    }
}
```

Cependant, la procédure qui a été écrite nécessite des éléments différents (des valeurs textuelles) de ce que nous avons l'habitude de traiter via les formulaires (des identifiants).

Il faut donc adapter le formulaire posté pour pouvoir transmettre les bonnes valeurs lors de l'appel de la fonction.

Nous pouvons par exemple ajouter :

- un tableau indexé pour l'ensemble des étudiants présents sur l'épreuve et dans lequel nous intégrons les noms et prénoms pour chacun.
- et, au même titre que l'intégration de l'identifiant de l'épreuve concernée, son libellé.

```
Array
(
    [id] => 6
    [action] => ajouter_note
    [libepr] => interro maths
    [etu] => Array
        (
            [1] => Array
                (
                    [nometu] => roblin
                    [prenometu] => lea
                )

            [2] => Array
                (
                    [nometu] => macarthur
                    [prenometu] => leon
                )

            [3] => Array
                (
                    [nometu] => minol
                    [prenometu] => luc
                )

            [4] => Array
                (
                    [nometu] => bagnole
                    [prenometu] => sophie
                )

            [5] => Array
                (
                    [nometu] => bury
                    [prenometu] => marc
                )

            [5] => Array
                (
                    [nometu] => bury
                    [prenometu] => marc
                )

            [6] => Array
                (
                    [nometu] => vendraux
                    [prenometu] => marc
                )

            [7] => Array
                (
                    [nometu] => vendermaele
                    [prenometu] => helene
                )

            [20] => Array
                (
                    [nometu] => GUERRIER
                    [prenometu] => Kevin
                )

        )

    [note] => Array
        (
            [1] => 7
            [2] => 8
            [3] => 9
            [4] => 10
            [5] => 11
            [6] => 12
            [7] => 13
            [20] => 14
        )

    [b_save] => Enregistrer
)
```

Ainsi, l'appel à la procédure ajout_note pourra se faire de la manière suivante :

```php
// récupération des données du formulaire
$list_infos_etu = GETPOST('etu') ?? array();
$list_notes = GETPOST('note') ?? array();
$libepr = GETPOST('libepr') ?? $elt['libepr'];

// création de la requête
$sql = "CALL ajout_note(:nometu, :prenometu, :note, :libepr)";

try {
    // envoi de la requête
    $statement = $db->prepare($sql);

    foreach ($list_notes as $numetu => $note) {
        $nometu = $list_infos_etu[$numetu]['nometu'];
        $prenometu = $list_infos_etu[$numetu]['prenometu'];
        $statement->execute(compact(['nometu', 'prenometu', 'note', 'libepr']));
    }
} catch (Exception $e) {
    $_SESSION['mesgs']['errors'][] = $e->getMessage();
}
header("location:afficher_epreuve.php?id=" . $numepr);
exit();
```

Sujet TP - Environnement JRCanDev

Retournons à présent à notre atelier pratique autour du sujet de : l'université.

Petit rappel : Vous réaliserez ce Projet de manière individuelle et originale (pas de copie, merci ^_^) et me transmettrez les éléments de votre Projet sous forme d'archive sur l'adresse mail : [[guerrier.k @ gmail.com]]. A noter qu'un lien vers l'archive partagée au sein d'un dossier Google Drive est préférable pour éviter toutes les restrictions liées aux sécurités des mails.

A partir de ce TP, vous réaliserez votre Projet en groupe de 3 ou 4 personnes, et vous demanderez la préparation d'un environnement dédié à votre projet sur JRCanDev (via Discord) . Vous me transmettrez par email l'adresse de votre projet sur l'adresse mail : [[guerrier.k @ gmail.com]].

L'environnement qui vous sera fourni sera composé : d'un serveur web avec PHP, d'un serveur PostgreSQL, d'une interface vous permettant d'interagir avec votre serveur PostgreSQL, ainsi qu'une interface vous permettant d'uploader / d'éditer vos fichiers PHP.

/!\ Une dernière chose importante /!\
Merci de respecter le nommage suivant pour vos archives :
2023_PostgreSQL_TP5_GROUPE00.zip

Exercice 1 - Information du groupe :

Vous réaliserez une page groupe.php à la racine de votre projet et qui affichera simplement le numéro du groupe que vous aurez précisé dans le nom de votre fichier (avec un lien téléchargeable vers cette archive), ainsi que les noms et prénoms des membres du groupe.

PS : Pensez à vous assurer de mettre à jour votre archive une fois vos exercices terminés ;-)

Exercice 2 - Script d'installation :

Ce projet commence à prendre de l'ampleur et comporte aussi bien des scripts au niveau applicatif (PHP) qu'au niveau SGBD (PostgreSQL). Aussi, vous réaliserez un script install.php qui permettra de déployer ce projet sur un nouveau serveur.

Ce script permettra notamment de connecter le serveur PostgreSQL, de réaliser les instructions de création de tables, de créations des fonctions et procédures, d'injection des données, …

Enfin, notez que ce script me permettra également de réinstaller votre projet sur ma machine pour évaluer vos réalisations ^_^ .

Exercice 3 - PL/PGSQL :

Enfin, vous réfléchirez aux imbrications possibles et utiles que vous pourriez mettre en place entre vos fonctions / procédures. Par exemple, lorsqu'un élément est déjà présent lors de l'insertion, est-ce qu'il est judicieux d'appeler une mise à jour à la place ?

Vous expliquerez ces éléments au sein d'un fichier details_traitements.php

Informations SAE

Fonctionnalités		PHP	PLPGSQL
Présentation	Page de présentation du groupe, de qui a fait quoi, et des fonctionnements implémentés sur le projet		
Général			
	Pages en accès public / utilisateur / administrateur	session	
	Visuel graphique un minimum travaillé	css (bootstrap ?)	
	Entêtes / Pieds de page / menus / ...centralisés	requires partials	
	Système de notifications des messages (confirms / erreurs / ...)	session	
BONUS ??	Interface Mutlilangue EN/FR	gettext	
Fonctions Utilisateur			
	Inscription Nouvel Utilisateur		via procédure
Administration			
	Possibilité de basculer un utilisateur en admin et inversement		trigger empêchant l'absence d'administrateur
	Possibilité de télécharger un installeur	Archive avec scripts install	
Installeur			
	Lien avec une archive contenant un installeur :		
	- install.php	unzip le archive.zip, demande les infos de la bd	
	- archive.zip	et execute tous les scripts sql nécessaires à l'installation	
Principes de classement			
	Visualisation d'au moins 1 classement	page php	PAS DE VUE
	Mise à jour automatique après chaque événement		via trigger
Votre projet			
	Réaliser des courses / des combats / des jeux / ...	Programmation Objet	procédures / fonctions / triggers
	Participer à un tournoi / championnat / ...		
	...		