Bases de Données Avancées

Rapport du *Projet PL/pgSQL* en Binôme

41LIDF01 / 41LIDF02
Professeur : Walid Bechkit

Eoin Brereton Hurley & Angela Pineda

December 19, 2025

1 Introduction

Ce rapport présente la conception et l'implémentation d'un ensemble de fonctions, de triggers et de tests en *PostgreSQL* dans le cadre du *Projet PL/pgSQL* du cours Bases de Données Avancées. L'objectif est de gérer le fonctionnement d'une bibliothèque, en automatisant les opérations courantes telles que l'ajout de livres et d'emprunteurs, la gestion des emprunts et des réservations, ainsi que l'extraction d'informations statistiques. Le travail est structuré en une partie guidée, répondant aux exigences du sujet, et une partie ouverte, proposant des extensions et des fonctionnalités supplémentaires.

L'intégralité du code est disponible dans le dépôt *GitHub* suivant : https://github.com/EoinBH/projet_M1_BDDA/

2 Partie Guidée

2.1 Fonctions

Toutes les fonctions de base ont été implémentées, ainsi que quatre fonctions supplémentaires que l'on estime nécessaires pour l'initialisation et l'alimentation de la base de données, à savoir : `ajouter_livre_texte`, `ajouter_categorie`, `ajouter_emprunteur` et `ajouter_emprunt`.

- Fonctions de base

  1. `ajouter_auteur(nom, pays)` : Cette fonction permet d'insérer le nom de famille et le pays d'origine d'un auteur dans la table `auteur`.
  2. `ajouter_livre(titre, id_auteur, id_categorie, annee, nb_exemplaires)` : Cette fonction permet d'insérer un livre dans la table `livre`. Une vérification est effectuée afin de s'assurer que le nombre d'exemplaires est strictement positif avant l'insertion. Pourtant, la conception de cette fonction suppose que l'on sache déjà l'identifiant de l'auteur et celui de la catégorie du livre. Cette conception est fonctionnelle, mais elle peut s'avérer peu pratique lors de l'initialisation de la base de données, car elle impose de connaître à l'avance les identifiants internes. Il serait peut-être utile d'avoir une fonction qui fait la même chose que celle-ci mais qui prend des chaînes de caractères en paramètres.
     - On a donc conçu une fonction `ajouter_livre_texte(titre TEXT, nom_auteur TEXT, nom_categorie TEXT, annee INT, nb_exemplaires INT)` pour pallier cette limitation. Elle permet d'insérer un livre en fournissant directement le nom de famille de l'auteur et le nom de la catégorie sous forme de

chaînes de caractères. La fonction récupère ensuite les identifiants associés pour les stocker dans la table.

```
SELECT id_auteur FROM auteur WHERE nom = nom_auteur
    INTO id_aut;
SELECT id_categorie FROM categorie WHERE nom = nom_categorie
    INTO id_cat;
```

Cette approche simplifie l'ajout de données lors de l'initialisation de la base de données.

3. `ajouter_categorie` : Cette fonction permet d'insérer une catégorie dans la table `categorie`. Dans la version actuelle du modèle, un livre est associé à une seule catégorie.

   > Une amélioration possible de la base de données consisterait à permettre à un livre d'être rattaché à plusieurs catégories, par exemple en introduisant une table associative.

4. `nb_livres_categorie(id_categorie)` : Cette fonction retourne le nombre total de livres appartenant à une catégorie donnée. Elle prend en paramètre l'identifiant de la catégorie et utilise une requête d'agrégation afin de compter les livres associés.
5. `ajouter_emprunteur(p_nom TEXT)` : Cette fonction permet d'insérer un emprunteur dans la table `emprunteur`.
6. `ajouter_emprunt(p_titre_livre TEXT, p_nom_emprunteur TEXT, duree INT)` : Cette fonction permet d'enregistrer un nouvel emprunt dans la base de données. Elle prend en paramètre le titre du livre, le nom de l'emprunteur ainsi que la durée du prêt, *exprimée en nombre de jours*. À partir de la date courante (`CURRENT_DATE`) et de la durée fournie, la fonction calcule la date de retour prévue.

   ```
   date_fin DATE = CURRENT_DATE + duree;
   ```

   Elle récupère ensuite l'identifiant du livre correspondant au titre donné ainsi que l'identifiant de l'emprunteur à partir de son nom de famille. Enfin, ces identifiants et la date de retour calculée, sont insérés dans la table `emprunt`.

- Fonctions avec boucle

  1. `maj_annee_livres()` : Cette fonction parcourt l'ensemble des livres dont l'année de publication est antérieure à 2000. À l'aide d'une boucle `FOR`, elle affiche pour chaque livre concerné son titre ainsi que son année de publication.

- Fonction avec curseur

1. `liste_livres_auteur(nom_auteur)` : Cette fonction retourne tous les livres écrits par un auteur donné. Pour préciser, elle renvoie un ensemble d'enregistrements de type `tLivre`, qui contient l'identifiant de l'auteur ainsi que les titres de chacun de ses livres. Cette fonction marche à l'aide d'un curseur explicite qui parcourt la table `livre`. À chaque itération, la fonction vérifie si l'identifiant de l'auteur correspond à celui recherché, et ajoute le livre au résultat si besoin.

```
WHILE FOUND LOOP
    IF tNuplet.id_aut = id_aut THEN
        RETURN NEXT tNuplet;
    END IF;
    FETCH cursLivres INTO tNuplet;
END LOOP;
```

- Fonction avec SQL dynamique

1. `compter_elements(table_name TEXT)` : Cette fonction permet de compter le nombre d'enregistrements présents dans une table dont le nom est fourni en paramètre. Elle nécéssite une requête SQL dynamique parce que le nom de la table est inconnu à la compilation. Il faut donc construire la chaîne de caractères qui constitue cette requête en concaténant le nom de la table. Cette approche permet de rendre la fonction générique et réutilisable pour différentes tables de la base de données.

```
EXECUTE 'SELECT COUNT (*) FROM ' || table_name INTO nombreEnreg;
```

- Trigger guidé

1. `verif_disponibilite` : Selon notre interprétation du schéma fourni, l'attribut `nb_exemplaires` de la table `livre` correspond au nombre d'exemplaires actuellement disponibles à la bibliothèque. Lorsqu'un nouvel emprunt est inséré dans la table emprunt, le trigger `verif_disponibilite` est déclenché. La fonction associée vérifie que le nombre d'exemplaires disponibles est strictement positif. Ensuite, s'il y a un exemplaire disponible, on s'assure de bien décrémenter `nb_exemplaires` pour indiquer que le livre a été emprunté. Si ce n'est pas le cas, une exception est levée afin d'empêcher l'emprunt.

```
IF nombreExem <= 0 THEN
    RAISE EXCEPTION 'Il n''y a plus d''exemplaires !';
END IF;
--Diminuer le nombre d'exemplaires :
UPDATE livre SET nb_exemplaires = nb_exemplaires - 1 WHERE
id_livre = NEW.id_livre;
```

3 Partie Ouverte

- Fonctions « classiques »

  1. `afficher_livres_totals()` : Ceci est une fonction statistique qui affiche le nombre de livres disponibles ainsi que les nombre de livres empruntés. En plus, elle renvoie un ensemble d'enregistrements (de type `tDispo`), qui contient les mêmes informations.

     ```
     RAISE NOTICE 'Livres disponisbles : %, Livres empruntés : %',
         livres_dispo, livres_empruntes;
     ```

  2. `livre_le_plus_emprunte()` : Cette fonction est également une fonction statistique. Elle affiche le titre du livre le plus emprunté ainsi que le nombre de prêts associés. Comme la dernière fonction, elle renvoie un ensemble d'enregistrements (de type `tEmpruntes`), qui contient les mêmes informations.

     ```
     RAISE NOTICE 'Nom du livre le plus emprunté : %, Nombre de prêts : %',
         nom_Livre, compte;
     ```

- Fonction utilisant un **curseur implicite** :

  1. `ajouter_reservation(p_livre TEXT, p_emprunteur TEXT)` : Cette fonction permet de réserver un livre s'il n'est pas déjà emprunté. Les curseurs implicites sont automatiquement déclenchés pour les requêtes `SELECT ... INTO`. *PostgreSQL* exécute la requête et prend directement le résultat pour le mettre dans une variable. La fonction vérifie d'abord que le livre existe, puis compare le nombre d'emprunts au nombre d'exemplaires disponibles. Si tous les exemplaires sont empruntés, elle crée une réservation ; sinon elle refuse (exception) car le livre est encore disponible.

- Fonction utilisant un **curseur explicite** :

  1. `liste_reservation_detail()` : Cette fonction retourne la liste des réservations avec le titre du livre, le nom de l'emprunteur, la date et le statut, grâce à une jointure entre `reservation`, `livre` et `emprunteur`. Elle utilise un curseur explicite : elle ouvre le curseur, récupère chaque ligne avec `FETCH` dans une boucle, renvoie chaque ligne avec `RETURN NEXT`, puis ferme le curseur.

- Fonction utilisant du **SQL paramétré**

  1. `nb_livres_par_categorie(p_categorie TEXT)`: Cette fonction calcule et retourne le nombre total d'exemplaires de livres pour la catégorie donnée, via `SUM(nb_exemplaires)` après jointure entre `livre` et `categorie`. C'est un exemple de SQL paramétré car la

requête utilise le paramètre `p_categorie` dans le `WHERE c.nom = p_categorie`.

```
SELECT SUM(nb_exemplaires) INTO total FROM livre l
JOIN categorie c ON l.id_categorie = c.id_categorie
WHERE c.nom = p_categorie;
```

- Fonction utilisant du **SQL dynamique**

1. `total_exemplaires_filtre(p_type_filtre TEXT, p_valeur TEXT)` : Cette fonction permet de compter le nombre total d'exemplaires après l'application d'un filtre. (On peut filtrer par titre, auteur ou catégorie du livre). Elle utilise du SQL dynamique car la requête change selon le type de filtre demandé (`titre`, `auteur` ou `categorie`) : ce n'est pas juste une valeur qui change, c'est la *structure de la requête* (tables et jointures). Elle construit donc une chaîne SQL différente (avec ou sans `JOIN`), l'exécute avec `EXECUTE ... INTO total`, puis retourne la somme des `nb_exemplaires` (0 si aucun résultat).

```
EXECUTE sql_query INTO total;
```

> Cette fonction est susceptible à *l'injection SQL*, en raison de la manière dont la requête dynamique est construite. Le paramètre `p_valeur`, fourni par l'utilisateur, est directement concaténé dans la chaîne SQL exécutée par `EXECUTE`, sans aucune protection. Si `p_valeur` contient du code SQL malveillant, celui-ci sera interprété comme faisant partie de la requête.

- Trigger

1. `trig_verif_reservation_validee` : Ce trigger est pertinent parce qu'il empêche une opération incohérente : valider une réservation pour un emprunteur qui a *déjà le même livre en emprunt*. Il se déclenche avant toute mise à jour du champ `statut` dans `reservation`. Si on passe de '`en attente`' a '`validee`', il compte les emprunts existants pour le couple (`id_livre`, `id_emprunteur`). Ensuite, s'il en trouve au moins un, alors l'emprunteur a déjà ce livre en prêt et l'opération est bloquée avec une exception. Sinon, il renvoie `NEW` et la mise à jour du statut est '`validee`'.

4 Conclusion

Ce projet a permis de mettre en œuvre les principaux mécanismes offerts par *PostgreSQL*, notamment les fonctions stockées, les boucles, les curseurs,

le SQL dynamique et les triggers. Les fonctionnalités développées assurent la cohérence des données et automatisent la gestion d'une bibliothèque de manière efficace. La partie ouverte démontre la possibilité d'enrichir la base de données par des fonctionnalités plus avancées. Enfin, des améliorations éventuelles peuvent être apportées à ce projet telles qu'une gestion plus fine des catégories ou l'implémentation des protections contre *l'injection SQL*.