Cours: SQL & Bases de Données

Ali ZAINOUL <ali.zainoul.az@gmail.com>

Crystal Clear Code
May 27, 2025

** Projet Fil Rouge **

Variables d'environnement

```
DB_USER="ecommerce_user"
DB_NAME="ecommerce_db"
TABLESPACE_NAME="ecommerce_ts"
TABLESPACE_PATH="/Users/Shared/pgsql_tablespaces/${TABLESPACE_NAME}"
PG_SUPERUSER=$(whoami)
```

Explications :

- `DB_USER` : nom du rôle PostgreSQL à créer/utiliser.
- `DB_NAME` : nom de la base de données.
- `TABLESPACE_NAME` : nom personnalisé du tablespace PostgreSQL.
- `TABLESPACE_PATH` : chemin local pour stocker les données du tablespace.
- `PG_SUPERUSER` : utilisateur macOS supposé superutilisateur PostgreSQL (souvent `postgres` ou le compte admin).

Vérification de la version PostgreSQL

```
PSQL_VERSION=$(psql --version | grep -oE '[0-9]+\.[0-9]+')
if [[ $PSQL_VERSION != 15* ]]; then
  echo "   PostgreSQL $PSQL_VERSION détecté. Utilisez bien PostgreSQL 15."
  exit 1
fi
```

- Extraction de la version majeure de `psql`.
- Contrôle que la version commence par `15` (version recommandée).
- En cas de non-conformité, avertissement et arrêt du script.

Démarrage et vérification du service PostgreSQL

```
brew services list | grep postgresql@15 | grep started >/dev/null
if [ $? -ne 0 ]; then
  brew services start postgresql@15
else
  echo "  PostgreSQL 15 déjà actif."
fi
```

- Vérifie si PostgreSQL 15 est lancé via Homebrew.
- Si non actif, démarre le service.
- Sinon, confirme qu'il est déjà actif.

Gestion du rôle PostgreSQL

```
ROLE_EXISTS=$(psql -U "$PG_SUPERUSER" -tAc \
  "SELECT 1 FROM pg_roles WHERE rolname='${DB_USER}'")

if [ "$ROLE_EXISTS" = "1" ]; then
  echo "  Le rôle '${DB_USER}' existe déjà."
else
  psql -U "$PG_SUPERUSER" -c "CREATE ROLE ${DB_USER} WITH LOGIN PASSWORD 'password';"
  psql -U "$PG_SUPERUSER" -c "ALTER ROLE ${DB_USER} CREATEDB;"
fi
```

- Test d'existence du rôle via une requête SQL.
- Création du rôle avec droits de connexion et création de bases si absent.
- Le mot de passe est ici statique (‘password‘), à adapter en production.

Gestion de la base de données

```
DB_EXISTS=$(psql -U "$PG_SUPERUSER" -tAc \
  "SELECT 1 FROM pg_database WHERE datname='${DB_NAME}'")

if [ "$DB_EXISTS" = "1" ]; then
  echo "  La base de données '${DB_NAME}' existe déjà."
else
  psql -U "$PG_SUPERUSER" -c "CREATE DATABASE ${DB_NAME} OWNER ${DB_USER};"
fi
```

- Recherche de la base existante.
- Création si nécessaire, en affectant la propriété au rôle utilisateur.

Gestion du répertoire Tablespace

```
if [ ! -d "${TABLESPACE_PATH}" ]; then
  sudo mkdir -p "${TABLESPACE_PATH}"
  sudo chown "$(whoami)" "${TABLESPACE_PATH}"
  echo "  Répertoire '${TABLESPACE_PATH}' créé."
else
  echo "   Le répertoire '${TABLESPACE_PATH}' existe déjà."
fi
```

- Vérifie l'existence du dossier tablespace.
- Création et attribution des droits à l'utilisateur actuel si absent.

Création ou vérification du tablespace

```
TABLESPACE_EXISTS=$(psql -U "$PG_SUPERUSER" -tAc \
  "SELECT 1 FROM pg_tablespace WHERE spcname = '${TABLESPACE_NAME}'")

if [ "$TABLESPACE_EXISTS" = "1" ]; then
  echo "   Le tablespace '${TABLESPACE_NAME}' existe déjà."
else
  psql -U "$PG_SUPERUSER" -c \
    "CREATE TABLESPACE ${TABLESPACE_NAME} LOCATION '${TABLESPACE_PATH}';"
fi
```

- Contrôle de l'existence du tablespace.
- Création si nécessaire avec le chemin local dédié.

Connexion test utilisateur

```
psql -U "${DB_USER}" -d "${DB_NAME}" -c "\l"
echo "  Script terminé avec succès."
```

- Teste la connexion à la base avec l'utilisateur créé.
- Affiche la liste des bases disponibles.
- Confirme la fin réussie du script.

Installation et Configuration sous Windows PowerShell

- Merci de télécharger et d'exécuter le script du lien suivant : lien suivant
- Ouvrir une fenêtre powershell en tant qu'administrateur, et exécuter avec: `.\sql_.bat`

Résumé des étapes principales

- Définir les variables d'environnement pour faciliter la gestion.
- Vérifier la version de PostgreSQL (v15 recommandée).
- Démarrer le service PostgreSQL si nécessaire.
- Créer le rôle utilisateur PostgreSQL avec droits.
- Créer la base de données et affecter le propriétaire.
- Créer un tablespace personnalisé (répertoire local + tablespace SQL).
- Tester la connexion avec le nouvel utilisateur.

Introduction au projet

Objectif : Comprendre le contexte métier et les enjeux fonctionnels.
Ce projet a pour but de concevoir, manipuler et interroger une base de données PostgreSQL pour un site e-commerce.
Le système devra :

- Gérer clients, produits, commandes, paiements, livraisons ;
- Maintenir les stocks en temps réel ;
- Générer des statistiques commerciales (produits populaires, panier moyen, chiffre d'affaires) ;
- Garantir la cohérence des données via contraintes, transactions et triggers.

Partie 2.1 – Entités fonctionnelles identifiées

Objectif : Identifier les entités métier pour la modélisation relationnelle.

Entité	Description
Customer	Client du site e-commerce
Product	Article à vendre
Order	Commande passée par un client
Order_Item	Produits détaillés dans une commande
Payment	Paiement associé à une commande
Shipment	Livraison d'une commande

Chaque entité deviendra une table avec relations et contraintes adaptées.

Partie 2.2 – Modélisation des relations

Objectif : Comprendre les liens entre entités et leurs cardinalités.

- **Customer ↔ Order** : relation 1-n (un client peut avoir plusieurs commandes).
- **Order ↔ Order_Item** : relation 1-n (une commande contient plusieurs lignes).
- **Order_Item ↔ Product** : relation n-m, résolue par la table `order_items`.
- **Order ↔ Payment** : relation 1-1 (simplification : un paiement par commande).
- **Order ↔ Shipment** : relation 1-1 (une livraison par commande).

Partie 2.3 – Choix techniques des tables

Objectif : Comprendre les types et contraintes utilisés.

- Clés primaires : `SERIAL` ou `GENERATED ALWAYS AS IDENTITY` (PostgreSQL 13+).
- Références via `FOREIGN KEY` explicites.
- Contraintes : `NOT NULL`, `UNIQUE`, `CHECK` pour garantir l'intégrité.
- Types spécifiques :
  - `TIMESTAMP DEFAULT CURRENT_TIMESTAMP` pour les dates ;
  - `NUMERIC(10,2)` pour les montants (éviter flottants) ;
  - `TEXT` pour les chaînes flexibles.

Partie 3 – Ordre de création des tables

Objectif : Respecter les dépendances pour éviter erreurs lors de la création.
Ordre conseillé :

1. `customers` (aucune dépendance) ;
2. `products` (aucune dépendance) ;
3. `orders` (dépend de `customers`) ;
4. `order_items` (dépend de `orders` et `products`) ;
5. `payments` (dépend de `orders`) ;
6. `shipments` (dépend de `orders`).

Création de la table `customers`

Objectif : Créer la table client avec les contraintes essentielles.

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Création de la table `products`

Objectif : Définir les produits disponibles et stocker leurs informations.

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT,
    price NUMERIC(10,2) NOT NULL CHECK (price >= 0),
    stock_quantity INTEGER NOT NULL CHECK (stock_quantity >= 0),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Création de la table `orders`

Objectif : Enregistrer les commandes passées par les clients.

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL REFERENCES customers(customer_id),
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status TEXT DEFAULT 'pending'
);
```

Création de la table `order_items`

Objectif : Détail des produits commandés, quantité et prix unitaire.

```
CREATE TABLE order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INTEGER NOT NULL REFERENCES orders(order_id),
    product_id INTEGER NOT NULL REFERENCES products(product_id),
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    unit_price NUMERIC(10,2) NOT NULL CHECK (unit_price >= 0)
);
```

Création de la table `payments`

Objectif : Stocker les paiements liés aux commandes.

```
CREATE TABLE payments (
    payment_id SERIAL PRIMARY KEY,
    order_id INTEGER NOT NULL REFERENCES orders(order_id),
    payment_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    amount NUMERIC(10,2) NOT NULL CHECK (amount >= 0),
    method TEXT NOT NULL,
    status TEXT DEFAULT 'completed'
);
```

Création de la table `shipments`

Objectif : Enregistrer les informations de livraison.

```
CREATE TABLE shipments (
    shipment_id SERIAL PRIMARY KEY,
    order_id INTEGER NOT NULL REFERENCES orders(order_id),
    shipment_date TIMESTAMP,
    delivery_address TEXT NOT NULL,
    status TEXT DEFAULT 'processing'
);
```

Partie 4 – Premiers jeux de données

Objectif : Insérer des exemples pour tester la base.

```
INSERT INTO customers (first_name, last_name, email)
VALUES
('Alice', 'Durand', 'alice@example.com'),
('Bruno', 'Martin', 'bruno@example.com');

INSERT INTO products (name, description, price, stock_quantity)
VALUES
('Laptop', 'Ordinateur portable 15 pouces', 899.99, 50),
('Souris', 'Souris sans fil ergonomique', 29.90, 200);
```

Partie 4 – Premières requêtes de test

Objectif : Valider les données insérées et tester la structure.

```
-- Voir tous les produits disponibles
SELECT * FROM products;

-- Rechercher un client par email
SELECT * FROM customers WHERE email = 'alice@example.com';

-- Voir toutes les commandes d'un client
SELECT * FROM orders WHERE customer_id = 1;
```

Résumé et objectifs atteints

À la fin de cette session, vous êtes capables de :

- Créer un rôle PostgreSQL sécurisé et une base propre ;
- Concevoir un modèle relationnel adapté au domaine métier ;
- Créer des tables robustes avec bonnes pratiques SQL (types, contraintes, relations) ;
- Insérer des données initiales pour tester la base ;
- Exécuter des requêtes simples pour valider la conception.

Prochaines étapes : approfondir les contraintes, transactions et optimisation.

Jour 2 — Requêtes intermédiaires, vues et agrégation

- **Partie 5** : Requêtes avancées et jointures
- **Partie 6** : Agrégats et statistiques
- **Partie 7** : Création de vues
- **Partie 8** : Requêtes imbriquées / CTE

Rappel : Jointures

- Une jointure permet de combiner des données de plusieurs tables en une seule requête.
- Types principaux :
  - `INNER JOIN` : correspondances exactes
  - `LEFT JOIN` : tous les enregistrements de la table de gauche, même sans correspondance
- Utilisation typique pour relier clients, commandes, produits, etc.

Partie 5 : Jointures — Commandes avec noms de produits et quantités

- Affiche les commandes, les noms des produits commandés et les quantités
- Nécessite de relier 3 tables via des `JOIN`

```
SELECT o.order_id,
       p.product_name,
       oi.quantity
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id;
```

Clients n’ayant jamais commandé

Objectif : Identifier les clients qui n’ont jamais passé de commande (via `LEFT JOIN` + filtre `IS NULL`)

```
SELECT c.customer_id,
       c.customer_name
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

Produits les plus vendus

Objectif : Lister les produits avec la quantité totale vendue, triés par popularité (agrégation + tri)

```
SELECT p.product_id,
       p.product_name,
       SUM(oi.quantity) AS total_sold
FROM products p
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name
ORDER BY total_sold DESC;
```

Rappel : Fonctions d'agrégation

- Permettent de résumer des données :
  - `SUM()`, `AVG()`, `COUNT()`, `MAX()`, `MIN()`
- Nécessitent souvent une clause `GROUP BY`
- Utiles pour analyser les ventes, les clients, les performances, etc.

Partie 6 : Agrégats — Panier moyen

Objectif : Calculer le montant moyen des commandes (moyenne par commande)

```
SELECT AVG(order_total) AS average_basket
FROM (
    SELECT o.order_id,
           SUM(oi.quantity * p.unit_price) AS order_total
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
    GROUP BY o.order_id
) subquery;
```

Chiffre d'affaires mensuel

Objectif : Calculer les revenus générés chaque mois (agrégat + date)

```
SELECT DATE_TRUNC('month', o.order_date) AS month,
       SUM(oi.quantity * p.unit_price) AS revenue
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
GROUP BY month
ORDER BY month;
```

Produits en rupture de stock

Objectif : Lister les produits dont la quantité en stock est nulle ou négative

```
SELECT product_id,
       product_name,
       stock_quantity
FROM products
WHERE stock_quantity <= 0;
```

Partie 7 : Définition — Qu’est-ce qu’une vue SQL ?

- Une vue est une requête enregistrée, accessible comme une table
- Permet de :
  - Simplifier des requêtes complexes
  - Réutiliser une logique métier
  - Masquer la complexité pour les utilisateurs
- Syntaxe : `CREATE OR REPLACE VIEW nom_vue AS SELECT ...`

Vue `customer_order_summary`

Objectif : Résumer le nombre de commandes et total dépensé par client

```
CREATE OR REPLACE VIEW customer_order_summary AS
SELECT c.customer_id,
       c.customer_name,
       COUNT(o.order_id) AS total_orders,
       SUM(oi.quantity * p.unit_price) AS total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
LEFT JOIN order_items oi ON o.order_id = oi.order_id
LEFT JOIN products p ON oi.product_id = p.product_id
GROUP BY c.customer_id, c.customer_name;
```

Vue `product_performance`

Objectif : Suivre les ventes et le stock de chaque produit

```
CREATE OR REPLACE VIEW product_performance AS
SELECT p.product_id,
       p.product_name,
       SUM(oi.quantity) AS total_sold,
       p.stock_quantity
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name, p.stock_quantity;
```

Partie 8 : Requêtes imbriquées — Sous-requêtes et CTE

- Une sous-requête est une requête à l'intérieur d'une autre :
  - Dans la clause `SELECT`, `FROM` ou `WHERE`
- Un `CTE` (Common Table Expression) est une requête temporaire nommée, plus lisible
- Syntaxe des CTE : `WITH nom_cte AS (SELECT ...)`

Filtrer les clients ayant dépensé plus de 1000 — Sous-requête

Objectif : Trouver les clients ayant dépensé plus de 1000

```
SELECT c.customer_id,
       c.customer_name
FROM customers c
WHERE (
    SELECT SUM(oi.quantity * p.unit_price)
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
    WHERE o.customer_id = c.customer_id
) > 1000;
```

CTE pour les clients ayant dépensé plus de 1000

Avantage : le `WITH` permet une meilleure lisibilité et réutilisabilité

```
WITH customer_spending AS (
    SELECT o.customer_id,
           SUM(oi.quantity * p.unit_price) AS total_spent
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
    GROUP BY o.customer_id
)
SELECT c.customer_id,
       c.customer_name
FROM customers c
JOIN customer_spending cs ON c.customer_id = cs.customer_id
WHERE cs.total_spent > 1000;
```

Jour 3 — Transactions, triggers, performances

- **Partie 9** : Transactions et contraintes
- **Partie 10** : Triggers
- **Partie 11** : Indexation et optimisation
- **Partie 12** : Sécurité et rôles

Partie 9 : Transactions et contraintes — Ajout d'une commande complète

Insertion d'une commande avec ses items et paiement dans une transaction pour garantir la cohérence

```
BEGIN;

INSERT INTO orders (order_id, customer_id, order_date)
VALUES (DEFAULT, 123, CURRENT_DATE)
RETURNING order_id INTO new_order_id;

INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (new_order_id, 10, 2, 15.00),
       (new_order_id, 20, 1, 30.00);
```

Rollback en cas d'erreur et isolation des transactions

Gestion des erreurs et niveaux d'isolation des transactions :

```
BEGIN;

-- Tentative d'insertion
INSERT INTO orders (...) VALUES (...);

-- En cas d'erreur
ROLLBACK;

-- Niveaux d'isolation (exemple en PostgreSQL)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- **READ COMMITTED** : niveau par défaut

Partie 10 : Triggers — Mise à jour automatique du stock après commande

Création d'un trigger pour décrémenter le stock à chaque ajout d'un item de commande

```
CREATE OR REPLACE FUNCTION update_stock_after_order() RETURNS TRIGGER
BEGIN
    UPDATE products
    SET stock_quantity = stock_quantity - NEW.quantity
    WHERE product_id = NEW.product_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_stock
```

Envoi automatique d'un statut de livraison

Trigger qui met à jour le statut de livraison après changement de paiement

```
CREATE OR REPLACE FUNCTION update_delivery_status() RETURNS TRIGGER A
BEGIN
    UPDATE orders
    SET delivery_status = 'Shipped'
    WHERE order_id = NEW.order_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_delivery
AFTER INSERT ON payments
FOR EACH ROW EXECUTE FUNCTION update_delivery_status();
```

Partie 11 : Indexation et optimisation — Création d'index

Créer des index pour optimiser les recherches fréquentes

```
CREATE INDEX idx_customers_email ON customers(email);
CREATE INDEX idx_order_items_product_id ON order_items(product_id);
CREATE INDEX idx_orders_order_id ON orders(order_id);
```

Utilisation de `EXPLAIN ANALYZE` pour analyser les performances

Exemple d'analyse d'une requête avec EXPLAIN ANALYZE

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE order_id = 123;
```

- Affiche le plan d'exécution de la requête
- Permet d'identifier si les index sont utilisés

Cas de non-utilisation d'index

Impact d'une requête sans index sur la performance

```
-- Supposons absence d'index sur order_date
EXPLAIN ANALYZE
SELECT * FROM orders WHERE order_date = '2025-01-01';
```

- Peut entraîner un `Seq Scan` (lecture séquentielle)
- Dégradation des performances pour les grandes tables

Partie 12 : Sécurité et rôles — Création de rôles

Création de rôles avec privilèges spécifiques

```
CREATE ROLE admin LOGIN PASSWORD 'admin_password';
CREATE ROLE readonly NOLOGIN;
CREATE ROLE client_user LOGIN PASSWORD 'client_password';
```

- `admin` : droits complets
- `readonly` : accès en lecture seule
- `client_user` : accès limité, par exemple à ses propres données

Gestion des droits sur les tables

Exemple d'attribution des droits pour chaque rôle

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin;

GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;

GRANT SELECT, INSERT ON orders TO client_user;
GRANT SELECT ON customers TO client_user;
```