

Aide-mémoire : Définition de déclencheur

```
CREATE FUNCTION nomFonction() RETURNS TRIGGER AS $$
    [DECLARE]
        -- Déclarations
    BEGIN
        -- Instructions
    END
$$ LANGUAGE plpgsql;

CREATE TRIGGER nomDeclencheur
    BEFORE | AFTER
    INSERT | DELETE | UPDATE | [INSERT] [[OR] DELETE] [[OR] UPDATE]
    ON nomTable
    FOR EACH ROW | FOR EACH STATEMENT
    EXECUTE PROCEDURE nomFonction();
```

Exercice 1 : Transformation automatique

1. Télécharger et exécuter le script SQL indiqué à l'adresse suivante, qui crée la table DEMO_STATES.

<http://eric.univ-lyon2.fr/jdarmont/docs/pgDemoStates.sql>

Conformément aux données existantes, on souhaite que toutes les nouvelles données insérées dans la table soient en majuscules.

2. Définir un déclencheur avant insertion ou modification qui transforme systématiquement en majuscules les valeurs de ST et STATE_NAME, quel que soit leur format d'origine (utiliser la fonction UPPER).
3. Tester l'insertion et la modification de quelques n-uplets et vérifier le contenu de la table DEMO_STATES.

Exercice 2 : Contrainte d'intégrité dynamique

Soit la table CLIBANQUE(idCli, nomCli, idConjoint#), où l'attribut idConjoint est l'identifiant (idCli) du conjoint du client courant. Il s'agit de coder un déclencheur avant insertion ou modification qui contrôle que le nom du/de la conjoint-e d'un-e client-e soit le même que celui de ce-tte client-e (plutôt conservateur, comme politique, n'est-ce pas ?). Si les deux noms diffèrent, une exception doit interrompre l'insertion ou la modification des données.

1. Télécharger et exécuter le script SQL indiqué à l'adresse suivante, qui crée la table CLIBANQUE.

<http://eric.univ-lyon2.fr/jdarmont/docs/pgCliBanque.sql>

2. Créer le déclencheur et la fonction associée.
3. Insérer et modifier quelques n-uplets dans la table CLIBANQUE. Conclusion ?

Exercice 3 : Statistiques d'utilisation

Il s'agit de recenser des statistiques de mise à jour des données (insertions, modifications, suppressions) de la table EMP.

1. Soit la table STATS (typeMaj, nbMaj, timestampModif), à créer à l'aide du script SQL ci-dessous.

<http://eric.univ-lyon2.fr/jdarmont/docs/pgSTATS.sql>

STATS

typeMaj	nbMaj	timestampModif
INSERT	0	NULL
UPDATE	0	NULL
DELETE	0	NULL

2. Définir un déclencheur après insertion, modification ou suppression dans la table EMP qui met à jour automatiquement la table STATS. Tester son fonctionnement en effectuant quelques modifications dans la table EMP.

Indications :

- Le type de mise à jour est donné par la variable système TG_OP.
- Le timestamp courant est donné par la fonction NOW().

3. Tester l'effet des clauses FOR EACH ROW et FOR EACH STATEMENT sur le comportement du déclencheur en utilisant une requête qui modifie plusieurs n-uplets (ex. UPDATE emp SET sal = sal * 1.05).

Exercice 4 : Rafraîchissement de vue matérialisée

1. Créer la table DEMO_CUSTOMERS depuis le script SQL ci-dessous.

<http://eric.univ-lyon2.fr/jdarmont/docs/pgDemoCustomers.sql>

2. Créer une table nommée CUSTNAMES à partir de la requête SQL suivante.

```
CREATE TABLE CUSTNAMES AS
SELECT CUSTOMER_ID, CUST_FIRST_NAME, CUST_LAST_NAME FROM DEMO_CUSTOMERS
```

3. Définir un déclencheur après insertion, modification ou suppression dans la table DEMO_CUSTOMERS qui répercute toutes les mises à jour de DEMO_CUSTOMERS dans CUSTNAMES.

4. Tester le fonctionnement du déclencheur en insérant, modifiant puis supprimant un n-uplet dans la table DEMO_CUSTOMERS. Vérifier à chaque étape si la mise à jour est bien répercutée dans CUSTNAMES.

Correction

-- Exercice 1

-- Déclencheur

```
CREATE OR REPLACE FUNCTION stateMAJ() RETURNS TRIGGER AS $$
BEGIN
    NEW.st := UPPER(NEW.st);
    NEW.state_name := UPPER(NEW.state_name);
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER stateMAJ -- Le déclencheur a souvent le même nom que la fonction
BEFORE INSERT OR UPDATE
ON demo_states
FOR EACH ROW
EXECUTE PROCEDURE stateMAJ();
```

-- Tests

```
INSERT INTO demo_states VALUES('xx', 'Phantom State');
UPDATE demo_states SET state_name = '51st state' WHERE st = 'xx';
SELECT * FROM demo_states;
```

-- Exercice 2

-- Déclencheur

```
CREATE OR REPLACE FUNCTION testConjoints() RETURNS TRIGGER AS $$
DECLARE
    nomConjoint clibanque.nomcli%TYPE;
BEGIN
    IF NEW.idConjoint IS NOT NULL THEN
        SELECT nomcli INTO nomConjoint FROM clibanque
        WHERE idcli = NEW.idconjoint;
        IF nomConjoint <> NEW.nomcli THEN
            RAISE EXCEPTION 'Noms des conjoints différents !';
        END IF;
    END IF;
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER testConjoints
BEFORE INSERT OR UPDATE
ON clibanque
FOR EACH ROW
EXECUTE PROCEDURE testConjoints();
```

-- Tests

```
INSERT INTO clibanque VALUES (1, 'Darmont', NULL);
INSERT INTO clibanque VALUES (2, 'Darmont', 1);
UPDATE clibanque SET idConjoint = 2 WHERE idcli = 1;
INSERT INTO clibanque VALUES (3, 'Bentayeb', NULL);
INSERT INTO clibanque VALUES (4, 'NotBentayeb', 3);
UPDATE clibanque SET idConjoint = 3 WHERE idcli = 1;
UPDATE clibanque SET nomcli = 'Darmon' WHERE idcli = 1;
```

-- Exercise 3

-- Déclencheur

```
CREATE OR REPLACE FUNCTION statsEmp() RETURNS TRIGGER AS $$
BEGIN
    UPDATE stats SET nbMaj = nbMaj + 1, timestampModif = NOW()
        WHERE typeMaj = TG_OP;
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER statsEmp
AFTER INSERT OR UPDATE OR DELETE
ON emp
FOR EACH ROW
EXECUTE PROCEDURE statsEmp();
```

-- Tests

```
INSERT INTO emp VALUES (9999, 'JACK', 'SALESMAN', 7698, null, 2500, 500, 30);
DELETE FROM emp WHERE empno = 9999;
UPDATE emp SET sal = sal * 1.05;
SELECT * FROM stats;
```

-- Exercise 4

-- Déclencheur

```
CREATE OR REPLACE FUNCTION refreshCustnames() RETURNS TRIGGER AS $$
BEGIN
    CASE TG_OP
        WHEN 'INSERT' THEN
            INSERT INTO custnames VALUES(NEW.customer_id,
                NEW.cust_first_name, NEW.cust_last_name);
        WHEN 'DELETE' THEN
            DELETE FROM custnames WHERE customer_id = OLD.customer_id;
        WHEN 'UPDATE' THEN
            UPDATE custnames SET      customer_id = NEW.customer_id,
                                    cust_first_name = NEW.cust_first_name,
                                    cust_last_name = NEW.cust_last_name
                WHERE customer_id = OLD.customer_id;
    END CASE;
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER refreshCustnames
AFTER INSERT OR UPDATE OR DELETE
ON demo_customers
FOR EACH ROW
EXECUTE PROCEDURE refreshCustnames();
```

-- Tests

```
INSERT INTO demo_customers VALUES(8, 'Darmont', 'Jérôme', NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL, NULL);
UPDATE demo_customers SET customer_id = 9, cust_first_name = 'Albertine'
    WHERE customer_id = 6;
DELETE FROM demo_customers WHERE customer_id = 4;
SELECT * FROM custnames;
```