**Lab Exercises**

**Count Employees by Town**

CREATE OR REPLACE FUNCTION fn\_count\_employees\_by\_town(town\_name VARCHAR(20))

RETURNS INT AS

$$

    DECLARE

        employees\_count INT;

    BEGIN

        SELECT

            COUNT(e.employee\_id)

        FROM

            employees AS e

        JOIN

            addresses AS a

        USING

            (address\_id)

        JOIN

            towns AS t

        USING

            (town\_id)

        WHERE

            t.name = town\_name INTO employees\_count;

        RETURN

            employees\_count;

    END

$$

LANGUAGE plpgsql

;

SELECT

    fn\_count\_employees\_by\_town('Sofia')

**Employees Promotion**

CREATE OR REPLACE PROCEDURE sp\_increase\_salaries(department\_name VARCHAR)

AS

$$

    BEGIN

        UPDATE

            employees

        SET

            salary = salary \* 1.05

        WHERE

            department\_id =

                (SELECT department\_id FROM departments WHERE name = department\_name);

    END

$$

LANGUAGE plpgsql

;

**Employees Promotion By ID**

CREATE OR REPLACE PROCEDURE sp\_increase\_salary\_by\_id(id INT)

AS

$$

    BEGIN

        IF

            (SELECT employee\_id FROM employees WHERE employee\_id = id) IS NULL

        THEN RETURN;

        END IF;

        UPDATE employees

        SET

            salary = salary \* 1.05 WHERE employee\_id = id;

        COMMIT;

    END;

$$

LANGUAGE plpgsql

;

**Triggered**

CREATE TABLE deleted\_employees(

    employee\_id SERIAL PRIMARY KEY,

    first\_name VARCHAR(20),

    last\_name VARCHAR(20),

    middle\_name VARCHAR(20),

    job\_title VARCHAR(50),

    department\_id INT,

    salary NUMERIC(19, 4)

)

;

CREATE OR REPLACE FUNCTION fired\_employees()

RETURNS TRIGGER AS

$$

    BEGIN

        INSERT INTO deleted\_employees(first\_name, last\_name, middle\_name, job\_title, department\_id, salary)

        VALUES (

            old.first\_name,

            old.last\_name,

            old.middle\_name,

            old.job\_title,

            old.department\_id,

            old.salary

        );

        RETURN old;

    END

$$

LANGUAGE plpgsql;

CREATE TRIGGER fired\_employees\_trigger

AFTER DELETE ON employees

FOR EACH ROW EXECUTE PROCEDURE fired\_employees();

**Homework Exercises**

**User-defined Function Full Name**

CREATE OR REPLACE FUNCTION fn\_full\_name(first\_name VARCHAR(50), last\_name VARCHAR(50))

RETURNS VARCHAR AS

$$

DECLARE

full\_name VARCHAR(50);

BEGIN

IF first\_name IS NULL AND last\_name IS NULL THEN

full\_name = NULL;

ELSE

SELECT

**CONCAT(INITCAP(first\_name), ' ', INITCAP(last\_name)) INTO full\_name;**

END IF;

RETURN full\_name;

END

$$

LANGUAGE plpgsql;

**User-defined Function Future Value**

CREATE OR REPLACE FUNCTION fn\_calculate\_future\_value(initial\_sum DECIMAL, yearly\_interest\_rate DECIMAL, number\_of\_years INT)

RETURNS DECIMAL AS

$$

    DECLARE

        future\_investment\_value DECIMAL;

    BEGIN

        SELECT initial\_sum \* (1 + yearly\_interest\_rate) ^ number\_of\_years INTO future\_investment\_value;

        RETURN

            TRUNC(future\_investment\_value, 4);

    END

$$

LANGUAGE plpgsql;

SELECT fn\_calculate\_future\_value(2500, 0.30, 2);

**User-defined Function Is Word Comprised**

CREATE OR REPLACE FUNCTION fn\_is\_word\_comprised(set\_of\_letters VARCHAR(50), word VARCHAR(50))

RETURNS BOOLEAN AS

$$

**DECLARE**

**i INT;**

**letter CHAR(1);**

**BEGIN**

**FOR i IN 1..LENGTH(word) LOOP**

**letter := SUBSTRING(LOWER(word), i, 1);**

**IF POSITION(letter IN LOWER(set\_of\_letters)) = 0 THEN**

**RETURN FALSE;**

**END IF;**

**END LOOP;**

**RETURN TRUE;**

END

$$

LANGUAGE plpgsql;

**Game Over**

CREATE OR REPLACE FUNCTION fn\_is\_game\_over(is\_game\_over BOOLEAN)

**RETURNS TABLE (**

**name VARCHAR(50),**

**game\_type\_id INT,**

**is\_finished BOOLEAN)**

AS

$$

BEGIN

**RETURN QUERY**

SELECT

g.name,

g.game\_type\_id,

g.is\_finished

FROM

games AS g

WHERE

g.is\_finished = is\_game\_over;

END

$$

LANGUAGE plpgsql;

**Difficulty Level**

CREATE OR REPLACE FUNCTION fn\_difficulty\_level(level INT)

RETURNS VARCHAR

AS

$$

DECLARE

difficulty\_level VARCHAR;

BEGIN

IF level <= 40 THEN difficulty\_level := 'Normal Difficulty';

ELSIF level BETWEEN 41 AND 60 THEN difficulty\_level := 'Nightmare Difficulty';

ELSE difficulty\_level := 'Hell Difficulty';

END IF;

RETURN difficulty\_level;

END

$$

LANGUAGE plpgsql;

**Cash in User Games Odd Rows**

CREATE OR REPLACE FUNCTION fn\_cash\_in\_users\_games(game\_name VARCHAR(50))

RETURNS TABLE(

total\_cash NUMERIC

)

AS

$$

BEGIN

RETURN QUERY

**WITH**

**ranked\_games**

**AS (**

SELECT

cash,

**ROW\_NUMBER() OVER (ORDER BY cash DESC) AS "row\_num"**

FROM

users\_games AS ug

JOIN

games AS g

ON

ug.game\_id = g.id

WHERE

g.name = game\_name

)

SELECT

ROUND(SUM(cash), 2)

FROM

**ranked\_games**

WHERE

**row\_num % 2 <>0;**

END

$$

LANGUAGE plpgsql;

**Deposit Money**

CREATE OR REPLACE PROCEDURE sp\_deposit\_money(

account\_id INT, money\_amount NUMERIC(10,4))

AS

$$

    BEGIN

        UPDATE

            accounts

        SET

            balance = balance + money\_amount

        WHERE

            id = account\_id;

    END

$$

LANGUAGE plpgsql;

**Withdraw Money**

CREATE OR REPLACE PROCEDURE sp\_withdraw\_money(

account\_id INT,

money\_amount NUMERIC(10, 4)

)

AS

$$

BEGIN

IF **(SELECT balance FROM accounts WHERE id = account\_id)** >= money\_amount THEN

UPDATE

accounts

SET balance = balance - money\_amount

WHERE

id = account\_id;

ELSE

RAISE NOTICE 'Insufficient balance to withdraw %', money\_amount;

END IF;

END

$$

LANGUAGE plpgsql;

**Money Transfer**

CREATE OR REPLACE PROCEDURE sp\_transfer\_money(

sender\_id INT,

receiver\_id INT,

amount NUMERIC(10,4)

)

AS

$$

BEGIN

**CALL sp\_withdraw\_money(sender\_id, amount);**

**IF (SELECT balance FROM accounts WHERE sender\_id = id) >= amount THEN**

**CALL sp\_deposit\_money(receiver\_id, amount);**

END IF;

END

$$

LANGUAGE plpgsql;

**Log Accounts Trigger**

CREATE TABLE logs(

id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

account\_id INT,

old\_sum NUMERIC,

new\_sum NUMERIC

);

CREATE OR REPLACE FUNCTION trigger\_fn\_insert\_new\_entry\_into\_logs()

RETURNS TRIGGER

AS

$$

BEGIN

INSERT INTO logs(account\_id, old\_sum, new\_sum)

VALUES

(OLD.id, OLD.balance, new.balance);

RETURN NEW;

END

$$

LANGUAGE plpgsql;

CREATE TRIGGER tr\_account\_balance\_change

AFTER UPDATE OF balance ON accounts

FOR EACH ROW

WHEN

(OLD.balance <> NEW.BALANCE)

EXECUTE FUNCTION trigger\_fn\_insert\_new\_entry\_into\_logs();

**Notification Email on Balance Change**

CREATE TABLE notification\_emails(

id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

recipient\_id INT,

subject VARCHAR,

body TEXT

);

CREATE OR REPLACE FUNCTION

trigger\_fn\_send\_email\_on\_balance\_change()

RETURNS TRIGGER AS

$$

BEGIN

INSERT INTO notification\_emails(recipient\_id, subject, body)

VALUES

(

NEW.account\_id,

'Balance change for account: %', NEW.account\_id,

'On % your balance was changed from % to %.', DATE(NOW()), NEW.old\_sum, NEW.new\_sum

);

RETURN NEW;

END

$$

LANGUAGE plpgsql;

CREATE TRIGGER tr\_send\_email\_on\_balance\_change

AFTER UPDATE ON logs

FOR EACH ROW

EXECUTE FUNCTION trigger\_fn\_send\_email\_on\_balance\_change();