**1)** Operations Performed by Functions:

**film\_in\_stock:** The film\_in\_stock function checks the availability of a specific film in stock at a particular store.

* It takes two inputs: the film ID (p\_film\_id) and the store ID (p\_store\_id).
* It returns a set of inventory IDs for copies of the film that are in stock at the specified store,
* using the inventory\_in\_stock function to determine stock availability.
* The function returns multiple inventory IDs if there are multiple copies of the film in stock.

**film\_not\_in\_stock:** The film\_not\_in\_stock function checks for films that are not in stock at a specific store.

* It takes two inputs: the film ID (p\_film\_id) and the store ID (p\_store\_id).
* It returns a set of inventory IDs for copies of the film that are not in stock at the specified store,
* using the inventory\_in\_stock function to check stock status.
* If no inventory items are out of stock, the function will return an empty set.

**inventory\_in\_stock:** The inventory\_in\_stock function checks whether a specific inventory item is in stock.

* It takes one input parameter:<p\_inventory\_id integer>: The ID of the inventory item to check.
* The function returns a boolean value (TRUE or FALSE), indicating whether the specified inventory item is currently in stock.

**get\_customer\_balance:** The get\_customer\_balance function computes a customer's balance based on:

* Total rental fees for films rented until the effective date.
* Late fees for overdue rentals.
* Payments made up to the specified date.
* It sums up these components and returns the final balance owed by the customer.

**inventory\_held\_by\_customer:** The function checks which customer is currently renting a given inventory item by searching the rental table for records where the item is rented but not yet returned (return\_date IS NULL). It returns the customer\_id of the renter or NULL if the item is not currently held.

**rewards\_report:** The rewards\_report function generates a list of customers who meet certain criteria based on their purchases in the last three months. Specifically, it identifies customers who have:

* Made at least a certain number of purchases (defined by the min\_monthly\_purchases parameter).
* Spent at least a specified dollar amount (defined by the min\_dollar\_amount\_purchased parameter) during the last three months.

**last\_day:** The last\_day function returns the last day of the month for a given timestamp with time zone.

* If the month is December, it returns the last day of that month.
* For other months, it calculates the last day of the current month by first determining the first day of the next month and then subtracting one day.

**2)** Why does rewards\_report return 0 rows? Correct and recreate the function, so that it's able to return rows properly.

The rewards\_report function returns no rows because of several issues in the function’s code. Here are the main issues:

Temporary Table Scope: In PL/pgSQL functions, the temporary table tmpCustomer is created for the session but may not persist properly across multiple executions or might not hold values expected by the function. Using a temporary table this way can be problematic in functions.

Dynamic Query Construction and Execution: The dynamic SQL used to populate the temporary table may not execute as expected, particularly if there are mismatches in column types or issues with SQL syntax within tmpSQL.

RETURN Statement: The function lacks an explicit RETURN statement at the end, which can prevent it from properly returning the expected set of rows.

To resolve these issues, here’s a corrected version of the function that avoids using a temporary table by directly constructing the required query. Additionally, the function is modified to return customer data directly without intermediate dynamic SQL steps.

DROP FUNCTION IF EXISTS dvdrental.rewards\_report(integer, numeric);

CREATE OR REPLACE FUNCTION dvdrental.rewards\_report(

min\_monthly\_purchases integer,

min\_dollar\_amount\_purchased numeric)

RETURNS SETOF dvdrental.customer

LANGUAGE 'plpgsql'

COST 100

VOLATILE SECURITY DEFINER PARALLEL UNSAFE

ROWS 1000

AS $BODY$

DECLARE

last\_month\_start DATE;

last\_month\_end DATE;

BEGIN

-- Sanity checks for input parameters

IF min\_monthly\_purchases <= 0 THEN

RAISE EXCEPTION 'Minimum monthly purchases parameter must be > 0';

END IF;

IF min\_dollar\_amount\_purchased <= 0 THEN

RAISE EXCEPTION 'Minimum monthly dollar amount purchased parameter must be > 0.00';

END IF;

-- Calculate the start and end dates for the last month

last\_month\_start := DATE\_TRUNC('month', CURRENT\_DATE - INTERVAL '1 month');

last\_month\_end := last\_month\_start + INTERVAL '1 month' - INTERVAL '1 day';

-- Return customers who meet the reward criteria directly

RETURN QUERY

SELECT c.\*

FROM dvdrental.customer c

JOIN dvdrental.payment p ON c.customer\_id = p.customer\_id

WHERE p.payment\_date BETWEEN last\_month\_start AND last\_month\_end

GROUP BY c.customer\_id

HAVING SUM(p.amount) > min\_dollar\_amount\_purchased

AND COUNT(p.payment\_id) > min\_monthly\_purchases;

END;

$BODY$;

ALTER FUNCTION dvdrental.rewards\_report(integer, numeric)

OWNER TO postgres;

**3)** Potential Function Removal:

Functions like film\_not\_in\_stock and inventory\_in\_stock could potentially be removed if they duplicate other functionalities (e.g., use of film\_in\_stock).

**4)** How do group\_concat and \_group\_concat work?

* In databases like MySQL, GROUP\_CONCAT is an aggregate function that concatenates values from multiple rows into a single string, separated by a delimiter.
* Here function \_group\_concat is not an aggregate function like GROUP\_CONCAT, but a helper function that concatenates two text inputs.
* It can be used in conjunction with STRING\_AGG or other aggregate functions to simulate a similar behavior in PostgreSQL.
* It concatenates two values, ensuring that if one is NULL, the other is returned, or it concatenates both with a comma and a space if neither are NULL.
* It can be used within aggregation queries to concatenate multiple rows of values.

**5)** What does last\_updated do?

The function dvdrental.last\_updated() is a trigger function designed to automatically update a last\_update timestamp column in a table whenever a row is modified (for example, during an UPDATE operation).

* RETURNS trigger: Specifies that this function is a **trigger function**. Trigger functions are not called directly; instead, they are executed automatically when specified events (e.g., INSERT, UPDATE, DELETE) occur on a table.
* NEW.last\_update = CURRENT\_TIMESTAMP;: Sets the last\_update column of the row being modified to the current timestamp.
* RETURN NEW;: Returns the modified row (NEW), which includes the updated last\_update timestamp. This step is necessary because trigger functions that modify rows need to return the row to finalize the update.

**6)** What is tmpSQL variable for in ‘rewards\_report’ function? Can this function be recreated without EXECUTE statement and dynamic SQL? Why?

The tmpSQL variable in the rewards\_report function is used to hold a dynamically constructed SQL statement as a TEXT string.

This SQL statement is later executed by the EXECUTE command, allowing the function to build and run the SQL dynamically based on the

parameters provided (min\_monthly\_purchases and min\_dollar\_amount\_purchased).

It’s possible to recreate rewards\_report without dynamic SQL. In the existing function,

EXECUTE is mainly used to build a query string that incorporates the values of min\_monthly\_purchases and min\_dollar\_amount\_purchased.

By using a WHERE clause and parameterized SQL, we could avoid dynamic SQL, thus improving readability, maintainability, and security.

Removing dynamic SQL reduces the risk of SQL injection, especially in functions that use user input.

tatic SQL can be precompiled and optimized by the database, whereas dynamic SQL is parsed and planned at runtime, which can incur additional overhead.

Alternative Solution Without Dynamic SQL

DROP FUNCTION IF EXISTS dvdrental.rewards\_report(integer, numeric);

CREATE OR REPLACE FUNCTION dvdrental.rewards\_report(

min\_monthly\_purchases integer,

min\_dollar\_amount\_purchased numeric)

RETURNS SETOF dvdrental.customer

LANGUAGE 'plpgsql'

COST 100

VOLATILE SECURITY DEFINER PARALLEL UNSAFE

ROWS 1000

AS $BODY$

DECLARE

last\_month\_start DATE;

last\_month\_end DATE;

BEGIN

-- Sanity checks for input parameters

IF min\_monthly\_purchases <= 0 THEN

RAISE EXCEPTION 'Minimum monthly purchases parameter must be > 0';

END IF;

IF min\_dollar\_amount\_purchased <= 0 THEN

RAISE EXCEPTION 'Minimum monthly dollar amount purchased parameter must be > 0.00';

END IF;

-- Calculate the start and end dates for the last month

last\_month\_start := DATE\_TRUNC('month', CURRENT\_DATE - INTERVAL '1 month');

last\_month\_end := last\_month\_start + INTERVAL '1 month' - INTERVAL '1 day';

-- Return customers who meet the reward criteria directly

RETURN QUERY

SELECT c.\*

FROM dvdrental.customer c

JOIN dvdrental.payment p ON c.customer\_id = p.customer\_id

WHERE p.payment\_date BETWEEN last\_month\_start AND last\_month\_end

GROUP BY c.customer\_id

HAVING SUM(p.amount) > min\_dollar\_amount\_purchased

AND COUNT(p.payment\_id) > min\_monthly\_purchases;

END;

$BODY$;

ALTER FUNCTION dvdrental.rewards\_report(integer, numeric)

OWNER TO postgres;

**7)** The ‘get\_customer\_balance’ function describes the business requirements for calculating the client balance. Unfortunately, not all of them are implemented in this function. Try to change function using the requirements from the comments.

Requirements Checklist:

* ✅ Rental Fees for All Previous Rentals (Implemented):

The calculation of v\_rentfees correctly sums up the rental\_rate of all films rented by the customer before the specified date.

* $1 for Every Day Rentals are Overdue (Partially Implemented):

The calculation of v\_overfees attempts to determine overdue days and applies a penalty of $1 per day. However:

It does not consider cases where the return\_date is NULL (e.g., rentals not returned).

It does not cap penalties at twice the rental duration as required.

* Replacement Cost for Rentals Overdue by More Than Twice the Rental Duration (Not Implemented):

There is no logic to apply the replacement\_cost for films overdue by more than rental\_duration \* 2 days.

* ✅ Subtract All Payments Made Before the Specified Date (Implemented):

The calculation of v\_payments correctly sums up payments made before the effective date.

CREATE OR REPLACE FUNCTION dvdrental.get\_customer\_balance(

p\_customer\_id integer,

p\_effective\_date timestamp with time zone)

RETURNS numeric

LANGUAGE 'plpgsql'

COST 100

VOLATILE PARALLEL UNSAFE

AS $BODY$

DECLARE

v\_rentfees DECIMAL(10,2); -- Fees for renting videos

v\_overfees DECIMAL(10,2); -- Late fees for overdue rentals

v\_replacement\_cost DECIMAL(10,2); -- Replacement costs for excessively overdue films

v\_payments DECIMAL(10,2); -- Payments made before the effective date

BEGIN

-- Calculate rental fees

SELECT COALESCE(SUM(dvdrental.film.rental\_rate), 0) INTO v\_rentfees

FROM dvdrental.film

JOIN dvdrental.inventory ON dvdrental.film.film\_id = dvdrental.inventory.film\_id

JOIN dvdrental.rental ON dvdrental.inventory.inventory\_id = dvdrental.rental.inventory\_id

WHERE dvdrental.rental.rental\_date <= p\_effective\_date

AND dvdrental.rental.customer\_id = p\_customer\_id;

-- Calculate overdue fees and replacement costs

SELECT COALESCE(SUM(

CASE

-- Rentals with a return\_date (calculate overdue fees)

WHEN dvdrental.rental.return\_date IS NOT NULL THEN

GREATEST(

EXTRACT(epoch FROM (dvdrental.rental.return\_date - (dvdrental.rental.rental\_date + dvdrental.film.rental\_duration \* '1 day'::interval)))::INTEGER / 86400,

0

)

-- Rentals not yet returned (use p\_effective\_date as return date)

ELSE

GREATEST(

EXTRACT(epoch FROM (p\_effective\_date - (dvdrental.rental.rental\_date + dvdrental.film.rental\_duration \* '1 day'::interval)))::INTEGER / 86400,

0

)

END), 0) AS overdue\_days,

COALESCE(SUM(

CASE

-- Charge replacement cost for rentals overdue by more than twice rental duration

WHEN COALESCE(dvdrental.rental.return\_date, p\_effective\_date) > dvdrental.rental.rental\_date + dvdrental.film.rental\_duration \* 2 \* '1 day'::interval

THEN dvdrental.film.replacement\_cost

ELSE 0

END), 0) AS replacement\_fees

INTO v\_overfees, v\_replacement\_cost

FROM dvdrental.rental

JOIN dvdrental.inventory ON dvdrental.rental.inventory\_id = dvdrental.inventory.inventory\_id

JOIN dvdrental.film ON dvdrental.inventory.film\_id = dvdrental.film.film\_id

WHERE dvdrental.rental.rental\_date <= p\_effective\_date

AND dvdrental.rental.customer\_id = p\_customer\_id;

-- Calculate payments

SELECT COALESCE(SUM(dvdrental.payment.amount), 0) INTO v\_payments

FROM dvdrental.payment

WHERE dvdrental.payment.payment\_date <= p\_effective\_date

AND dvdrental.payment.customer\_id = p\_customer\_id;

-- Return the calculated balance

RETURN v\_rentfees + v\_overfees + v\_replacement\_cost - v\_payments;

END

$BODY$;

ALTER FUNCTION dvdrental.get\_customer\_balance(integer, timestamp with time zone)

OWNER TO postgres;