--TaskN6

**SELECT** pg\_get\_functiondef(*p*.oid)

**FROM** pg\_proc *p*

**JOIN** pg\_namespace *n* **ON** *p*.pronamespace = *n*.oid

**WHERE** *p*.proname = 'get\_customer\_balance' -- Your function name

**AND** *n*.nspname = 'public'; -- Schema name

--TaskN6.1

--film\_in\_stock

--It filters the inventory table for film\_id AND store\_id

--Then for each inventory\_id found, it calls inventory\_in\_stock(inventory\_id) to check:

--Has it never been rented?

--OR has it been returned?

--Finally returns inventory\_id for all available films in stock

--public.film\_not\_in\_stock

--In the same way almost the same function as film\_in\_stock except not in third statement of where thus it checks for products that are not available in stock

--public.get\_customer\_balance

--Adds rental fees for all films the customer rented before the given date, then adds late fees $1 per day for each day a film was returned late.Subtracts all payments made by the customer before the given date

--inventory\_held\_by\_customer(p\_inventory\_id)

--Checks if a specific inventory item is currently rented out .If yes, it returns the customer\_id of the customer who has it.If no one currently holds it (i.e., it's returned), it returns NULL.

--rewards\_report

--Ensures that the min\_monthly\_purchases and min\_dollar\_amount\_purchased are both greater than 0. If not, it raises an error.

--It calculates the start and end dates of the previous month using CURRENT\_DATE - '3 month'::interval.

--Creates a temporary table (tmpCustomer) to store customer\_ids of qualifying customers.

--The function dynamically creates an INSERT SQL query (tmpSQL) to find customers who made at least min\_monthly\_purchases payments and spent more than the specified min\_dollar\_amount\_purchased in the last 3 months.

--After inserting qualifying customer IDs into the temporary table, it fetches and returns the customer data by joining tmpCustomer with the customer table.

--Drops the temporary table tmpCustomer to clean up after the function execution.

--The function returns a set of customer records for all customers who meet the criteria for the reward.

--last\_day(timestamp with time zone)

--Takes a timestamp with time zone ($1), which is the date for which we want to find the last day of the month.

--The function checks if the month is December (i.e., EXTRACT(MONTH FROM $1) = 12):

--If it's December, it returns the last day of the year by calculating the first day of the next year and subtracting 1 day.

--Otherwise, it calculates the first day of the next month and subtracts 1 day to get the last day of the current month.

--The function returns the last date of the month for the given timestamp as a date.

--TaskN6.2

--The problem was with the dynamic SQL query construction and filtering logic in the HAVING clause. The query wasn’t correctly checking for qualifying customers due to improper date handling and incorrect logic in the aggregation (COUNT and SUM). Additionally, the tmpSQL string concatenation could have led to issues with data types and query execution. By refactoring the query and fixing these issues, the function now properly identifies customers based on the specified criteria

**CREATE** **OR** **REPLACE** **FUNCTION** public.rewards\_report(

min\_monthly\_purchases **integer**,

min\_dollar\_amount\_purchased **numeric**

)

**RETURNS** **SETOF** customer

**LANGUAGE** plpgsql

**SECURITY** **DEFINER**

**AS** **$function$**

**DECLARE**

last\_month\_start **DATE**;

last\_month\_end **DATE**;

rr RECORD;

**BEGIN**

**IF** min\_monthly\_purchases <= 0 **THEN**

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

**END** **IF**;

**IF** min\_dollar\_amount\_purchased <= 0.00 **THEN**

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

**END** **IF**;

last\_month\_start := CURRENT\_DATE - **INTERVAL** '3 month';

last\_month\_start := **to\_date**(**EXTRACT**(**YEAR** **FROM** last\_month\_start) || '-' || **EXTRACT**(**MONTH** **FROM** last\_month\_start) || '-01', 'YYYY-MM-DD');

last\_month\_end := LAST\_DAY(last\_month\_start); -- Using the previously defined 'last\_day' function if needed

**CREATE** **TEMPORARY** **TABLE** tmpCustomer (customer\_id **INTEGER** **NOT** **NULL** **PRIMARY** **KEY**);

**INSERT** **INTO** tmpCustomer (customer\_id)

**SELECT** p.customer\_id

**FROM** payment **AS** p

**WHERE** **DATE**(p.payment\_date) **BETWEEN** last\_month\_start **AND** last\_month\_end

**GROUP** **BY** p.customer\_id

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

**AND** **COUNT**(p.customer\_id) >= min\_monthly\_purchases;

**FOR** rr **IN**

**EXECUTE** 'SELECT c.\* FROM tmpCustomer t INNER JOIN customer c ON t.customer\_id = c.customer\_id'

**LOOP**

**RETURN** **NEXT** rr;

**END** **LOOP**;

**DROP** **TABLE** tmpCustomer;

**RETURN**;

**END**;

**$function$**;

--TaskN6.3

--film\_not\_in\_stock function and keep film\_in\_stock, while simply modifying the logic of film\_in\_stock to optionally handle both cases.

-- Both functions are based on the same logic — they are just checking for the inverse condition using NOT inventory\_in\_stock(inventory\_id) in the second case. This makes the functions redundant.

--It's better to keep a single function and pass a flag or adjust the logic to handle both in-stock and out-of-stock cases. This way, the logic is centralized, and the codebase is more maintainable.

--TaskN6.4

**CREATE** **OR** **REPLACE** **FUNCTION** public.get\_customer\_balance(

p\_customer\_id **integer**,

p\_effective\_date **timestamp** **with** **time** **zone**

)

**RETURNS** **numeric**

**LANGUAGE** plpgsql

**AS** **$function$**

**DECLARE**

v\_rentfees **DECIMAL**(5,2);

v\_overfees **DECIMAL**(5,2);

v\_replacement\_fees **DECIMAL**(5,2);

v\_payments **DECIMAL**(5,2);

**BEGIN**

**SELECT** **COALESCE**(**SUM**(film.rental\_rate), 0)

**INTO** v\_rentfees

**FROM** film

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

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

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

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

**SELECT** **COALESCE**(**SUM**(**CASE**

**WHEN** (rental.return\_date - rental.rental\_date) > (film.rental\_duration \* '1 day'::**interval**)

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

**ELSE** 0

**END**), 0)

**INTO** v\_overfees

**FROM** rental

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

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

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

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

**SELECT** **COALESCE**(**SUM**(**CASE**

**WHEN** (rental.return\_date - rental.rental\_date) > (film.rental\_duration \* 2 \* '1 day'::**interval**)

**THEN** film.replacement\_cost

**ELSE** 0

**END**), 0)

**INTO** v\_replacement\_fees

**FROM** rental

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

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

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

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

**SELECT** **COALESCE**(**SUM**(payment.amount), 0)

**INTO** v\_payments

**FROM** payment

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

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

**RETURN** v\_rentfees + v\_overfees + v\_replacement\_fees - v\_payments;

**END**

**$function$**;

--TaskN6.5

--GROUP\_CONCAT (in MySQL) is an aggregate function that concatenates values from multiple rows into a single string, with an optional separator

--\_group\_concat might be a custom PostgreSQL function that uses STRING\_AGG or a similar method to mimic GROUP\_CONCAT

--These functions are used when you want to combine multiple rows into one, typically for generating lists or reports

--TaskN6.6

--last\_updated

--This function sets the last\_update column to the current timestamp

--It’s meant to be used in a trigger that fires on INSERT or UPDATE events.

--TaskN6.7

--The tmpSQL variable in the rewards\_report function is used to build and execute dynamic SQL. It stores a SQL query as a string, which is then run using the EXECUTE statement

--In the rewards\_report function, the tmpSQL variable is used to build a SQL command as a text string, and then that string is run using EXECUTE. This is called dynamic SQL.

--But in this case, we don’t need to build the SQL like that. We can just write a regular SQL query that uses the input parameters directly (like min\_monthly\_purchases), without turning them into text and running them with EXECUTE