**Answers for questions in task 6. In SQL\_DDL\_Functions homework**

1. What operations do the following functions perform: film\_in\_stock, film\_not\_in\_stock, inventory\_in\_stock, get\_customer\_balance, inventory\_held\_by\_customer, rewards\_report, last\_day? You can find these functions in dvd\_rental database.

For this question i would have to explain what each function performs.

* 1. film\_in\_stock

This function checks whether a specific film is curently available in stock. Function uses the inventory table to check if the stock level for a particular film is greater then zero, meaning the film is available for rental.

It returns a boolean value indicating if the film is in stock.

* 1. film\_not\_in\_stock

This function checks whether a specific film is not in stock. Function also uses inventory table to verify stock for a film (if it is zero or less). It returnes a boolean value.

* 1. inventory\_in\_stock

This function likely checks the total number of films available for renal in the inventory. It uses inventory table to count all films that have a positive available\_stock value. It returns total count of films available in stock.

* 1. get\_customer\_balance

This function retrives the balance of a specific costumer, used to track the remaining amount owed by a customer for rental or late fees. Uses tables like payments, rental to calculate the customers balance. It returns the balance amount (positive or negative) for the customer.

* 1. inventory\_held\_by\_customer

This function identifies the films that are currently held by a particular customer. It uses table rentals to find out which films a customer has rented, including details (rental date, return date). It uses table rentals to find out whick films has been rented. It returns a list of films that are currently rented and not yet returned by the customer.

* 1. rewards\_report

This function generates a report based on customer reward points by tracking customer activity such as rentals or purchases. Uses tables customers or rentals to aggregate date like how may points a customer has earned or it could track activites. It returns a report showing the reward points earned by each customer.

* 1. last\_day

This function is used to determine the last day of the month from the time zone that is given. It uses date to determine the last day of current month, and december (to change to january). It returns a date representing last day of the month for the provided timestamp.

1. Why does ‘rewards\_report’ function return 0 rows? Correct and recreate the function, so that it's able to return rows properly.
   1. The condition uses the “>” operator for both the total amount and the count of payments. It is recommended to allow equality (using “>=”) rather than strictly greater-than.
   2. The date interval for “last month” is calculated using a 3‐month offset.
   3. Using a temporary table without first checking if one exists may cause issues in some context.

In the function it is changed:

* 1. The code now uses date\_trunc('month', current\_date) - interval '1 month' to capture the first day of the previous calendar month.
  2. The variable last\_month\_end is then determined using the provided last\_day function.
  3. Changed from using > to >= so that customers who exactly meet the minimum purchase count and dollar amount are included.
  4. Uses if not exists with the clause on commit drop to ensure that the table is removed at the end of the transaction and to avoid issues with duplicate temporary table creation.

Here is updated code for rewards\_report:

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;

tmpSQL TEXT;

BEGIN

-- ✅ IF min\_monthly\_purchases <= 0 THEN

~~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

~~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 := date\_trunc('month', current\_date) - interval '1 month';

~~last\_month\_start := CURRENT\_DATE - '3 month'::interval;~~

-- ✅ last\_month\_start := to\_date(to\_char(last\_month\_start, 'yyyy-mm-01'),'yyyy-mm-dd');

~~last\_month\_start := to\_date(~~

~~(EXTRACT(YEAR FROM last\_month\_start) || '-' || EXTRACT(MONTH FROM last\_month\_start) || '-01'),~~

~~'YYYY-MM-DD'~~

~~);~~

-- ✅ last\_month\_end := public.last\_day(last\_month\_start);

~~last\_month\_end := LAST\_DAY(last\_month\_start);~~

-- ✅ CREATE TEMPORARY TABLE tmpcustomer (...) ON COMMIT DROP;

~~CREATE TEMPORARY TABLE tmpCustomer (~~

~~customer\_id INTEGER NOT NULL PRIMARY KEY~~

~~);~~

-- ✅ It was necessary to use static SQL:

-- INSERT INTO tmpcustomer(customer\_id)

-- SELECT p.customer\_id FROM payment p ...

tmpSQL := 'INSERT INTO tmpCustomer (customer\_id)

SELECT p.customer\_id

FROM payment AS p

WHERE DATE(p.payment\_date) BETWEEN ' || quote\_literal(last\_month\_start) || ' AND ' || quote\_literal(last\_month\_end) || '

GROUP BY customer\_id

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

AND COUNT(customer\_id) > ' || min\_monthly\_purchases;

EXECUTE tmpSQL;

-- ✅ It should have used a regular **SELECT**, without **EXECUTE**:

-- FOR rr IN SELECT c.\* FROM tmpcustomer t ...

FOR rr IN EXECUTE '

SELECT c.\*

FROM tmpCustomer AS t

INNER JOIN customer AS c ON t.customer\_id = c.customer\_id

' LOOP

RETURN NEXT rr;

END LOOP;

-- ✅ This should have been omitted, because **ON COMMIT DROP** automatically drops the table. tmpSQL := 'DROP TABLE tmpCustomer';

EXECUTE tmpSQL;

RETURN;

END;

$function$;

1. Is there any function that can potentially be removed from the dvd\_rental codebase? If so, which one and why?

From the database functions the one function that stands out as potentially redundant is \_group\_concat. Both group\_concat and \_group\_concat typically serve the same purpose: aggregating multiple row values into a single, concatenated string. In many cases, the underscore-prefixed version (i.e., \_group\_concat) is either a legacy helper function or simply an alias/duplicate of the main group\_concat functionality. Maintaining two functions that perform the same task adds unnecessary complexity to the codebase and can lead to confusion about which one should be used.

1. \* 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.
   1. Rental Fees: The sum of the rental rates (from the film table) for all rentals that occurred on or before the effective date.
   2. Overdue Fees: For each rental, it calculates the overdue days as the difference between the return date and the rental date minus the allowed rental duration (converted into an interval). If the overdue period is less than or equal to zero, no overdue fee is charged. If the overdue period is more than rental\_duration × 2, the replacement cost is charged. Otherwise, a fee of $1 per day is applied for each overdue day.
   3. Payments: The total of all payments made on or before the effective date is subtracted from the sum of rental and overdue fees.

Here’s the updated function:

DROP FUNCTION public.get\_customer\_balance(int4, timestamptz);

CREATE OR REPLACE FUNCTION public.get\_customer\_balance(p\_customer\_id integer, p\_effective\_date timestamp with time zone)

RETURNS numeric

LANGUAGE plpgsql

AS $function$

--#OK, WE NEED TO CALCULATE THE CURRENT BALANCE GIVEN A CUSTOMER\_ID AND A DATE

--#THAT WE WANT THE BALANCE TO BE EFFECTIVE FOR. THE BALANCE IS:

--# 1) RENTAL FEES FOR ALL PREVIOUS RENTALS

--# 2) ONE DOLLAR FOR EVERY DAY THE PREVIOUS RENTALS ARE OVERDUE

--# 3) IF A FILM IS MORE THAN RENTAL\_DURATION \* 2 OVERDUE, CHARGE THE REPLACEMENT\_COST

--# 4) SUBTRACT ALL PAYMENTS MADE BEFORE THE DATE SPECIFIED

DECLARE

✅v\_rentfees NUMERIC(10,2); -- fees paid to rent the videos initially

~~v\_rentfees DECIMAL(5,2); --#FEES PAID TO RENT THE VIDEOS INITIALLY~~

✅v\_overfees NUMERIC (10,2); -- late fees for overdue rentals

~~v\_overfees INTEGER; --#LATE FEES FOR PRIOR RENTALS~~

✅v\_payments NUMERIC (10,2); -- total of payments made before the effective date

~~v\_payments DECIMAL(5,2); --#SUM OF PAYMENTS MADE PREVIOUSLY~~

BEGIN

SELECT COALESCE(SUM(f.rental\_rate), 0)

INTO v\_rentfees

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

FROM film f

~~FROM film, inventory, rental~~

INNER JOIN inventory i ON f.film\_id = i.film\_id

INNER JOIN rental r ON i.inventory\_id = r.inventory\_id

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

AND r.customer\_id = p\_customer\_id;

~~WHERE film.film\_id = inventory.film\_id~~

~~AND inventory.inventory\_id = rental.inventory\_id~~

~~AND rental.rental\_date <= p\_effective\_date~~

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

SELECT COALESCE(SUM(

CASE

WHEN r.return\_date IS NULL THEN 0

WHEN (r.return\_date - r.rental\_date) - (f.rental\_duration \* INTERVAL '1 day') <= INTERVAL '0 day' THEN 0

WHEN (r.return\_date - r.rental\_date) - (f.rental\_duration \* INTERVAL '1 day') > (f.rental\_duration \* INTERVAL '2 day')

THEN f.replacement\_cost

ELSE EXTRACT(DAY FROM ((r.return\_date - r.rental\_date) - (f.rental\_duration \* INTERVAL '1 day')))::INTEGER

END

), 0)

~~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 r

~~FROM rental, inventory, film~~

INNER JOIN inventory i ON r.inventory\_id = i.inventory\_id

INNER JOIN film f ON i.film\_id = f.film\_id

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

~~WHERE film.film\_id = inventory.film\_id~~

AND r.customer\_id = p\_customer\_id;

~~AND inventory.inventory\_id = rental.inventory\_id~~

~~AND 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\_payments;

END

$function$

;

1. How do ‘group\_concat’ and ‘\_group\_concat’ functions work? (database creation script might help) Where are they used?
   1. \_group\_concat:

A helper SQL function that connect two text strings with a comma and space separator. If one of the inputs is null, it returns the non-null input.

* 1. group\_concat:

An aggregate function built on top of \_group\_concat. It applies the \_group\_concat function iteratively to combine multiple text values from grouped rows into a single, comma-separated string.

These functions are used to aggregate text from multiple rows (e.g., aggregating employee names by department) to form a concise, readable list in reports, views, or complex queries, compensating for PostgreSQL’s lack of a built-in group concatenation function.

1. What does ‘last\_updated’ function do? Where is it used?

The last\_updated function is a trigger function designed to automatically update a record's last\_update column whenever the record is modified.