**Task 6**

--1

* **film\_in\_stock**: Checks if a specific film is available for rent in a particular store by verifying if the film's inventory is in stock.
* **film\_not\_in\_stock**: Identifies films that are not available for rent in a specific store, often used to determine out-of-stock items.
* **inventory\_in\_stock**: Determines whether a particular inventory item is currently in stock, based on rental status and return dates.
* **get\_customer\_balance**: Calculates a customer's balance as of a specific date, considering payments, rentals, and any outstanding amounts.
* **inventory\_held\_by\_customer**: Finds the customer currently holding a specific inventory item, based on rental records.
* **rewards\_report**: Generates a report of customers eligible for rewards, typically based on criteria like monthly purchases and total spending.
* **last\_day**: Returns the last day of the month for a given date, useful for monthly reporting and calculations.

--2

The rewards\_report function might return 0 rows because of the missing paranthesis next to HAVING

-- FUNCTION: public.rewards\_report(integer, numeric)

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

CREATE OR REPLACE FUNCTION public.rewards\_report( min\_monthly\_purchases integer, min\_dollar\_amount\_purchased numeric) RETURNS SETOF customer LANGUAGE 'plpgsql' COST 100 VOLATILE SECURITY DEFINER PARALLEL UNSAFE ROWS 1000

AS $BODY$

DECLARE last\_month\_start DATE; last\_month\_end DATE; rr RECORD; tmpSQL TEXT; BEGIN

/\* Some sanity checks... \*/  
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 - '3 month'::interval;  
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);  
  
/\*  
Create a temporary storage area for Customer IDs.  
\*/  
CREATE TEMPORARY TABLE tmpCustomer (customer\_id INTEGER NOT NULL PRIMARY KEY);  
  
/\*  
Find all customers meeting the monthly purchase requirements  
\*/  
  
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 );

--added paranthesis here  
  
EXECUTE tmpSQL;  
  
/\*  
Output ALL customer information of matching rewardees.  
Customize output as needed.  
\*/  
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;  
  
/\* Clean up \*/  
tmpSQL := 'DROP TABLE tmpCustomer';  
EXECUTE tmpSQL;

RETURN; END $BODY$;

ALTER FUNCTION public.rewards\_report(integer, numeric) OWNER TO postgres;

--3

Potentially last\_day function can be removed, as it can also be easily written with built in sql functions like extract.

--4

-- FUNCTION: public.get\_customer\_balance(integer, timestamp with time zone)

-- DROP FUNCTION IF EXISTS public.get\_customer\_balance(integer, timestamp with time zone);

CREATE OR REPLACE FUNCTION public.get\_customer\_balance( p\_customer\_id integer, p\_effective\_date timestamp with time zone) RETURNS numeric LANGUAGE 'plpgsql' COST 100 VOLATILE PARALLEL UNSAFE AS $BODY$

--#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 DECIMAL(5,2); --#FEES PAID TO RENT THE VIDEOS INITIALLY v\_overfees INTEGER; --#LATE FEES FOR PRIOR RENTALS

v\_replacement DECIMAL(6,2); --replacement cost for late returns

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

BEGIN

SELECT

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

FROM film, inventory, rental

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 (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, inventory, film  
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;

--the case when films are overdue more than 2 \* rental\_duration

--then calculate the replacement\_cost

SELECT

COALESCE(SUM( CASE

WHEN (r.return\_date - r.rental\_date) > (f.rental\_duration \* 2 \* INTERVAL '1 day') THEN f.replacement\_cost ELSE 0 END), 0)

INTO v\_replacement

FROM film f

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

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

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

AND r.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 – v\_payments; --consider the v\_replacement cost as well

END $BODY$;

ALTER FUNCTION public.get\_customer\_balance(integer, timestamp with time zone) OWNER TO postgres;

--5

\_group\_concat – is a function that joins to text values with a comma

group\_concat – is an aggregate function that uses \_group\_concat to create a string from multiple rows separated by a comma

They are ofen used in reports or summaries, etc.

--6

The last\_updated function retrives the most recent timestamp indicating the last date of modification of the record. It helps to monitor all the changes made to the data.

--7

The tmpSQL variable in the rewards\_report function is used to store dynamic SQL queries that are constructed and then executed at runtime. It allows the function to perform actions like inserting into a temporary table or querying data dynamically based on user inputs. Without using the EXECUTE statement and dynamic SQL, it would be difficult to accomplish these tasks because the query construction needs to adapt to variable inputs. For example, it dynamically generates the SQL to find customers based on specific conditions like the minimum monthly purchases and dollar amount spent.