1. What operations do the following functions perform?

**film\_in\_stock:** this checks if a certain film has available (in-stock) copies in a certain store, and returns the inventory\_id of all available copies that meet this criteria. Note: the output parameter has a misleading name: p\_film\_count, as the returned value(s) are the inventory ids of the films, not the number of available films.

**film\_not\_in\_stock:** This checks if a certain film has copies in a certain store, and returns the inventory\_id of all unavailable copies that meet this criteria. Note: the output parameter has a misleading name: p\_film\_count, as the returned value(s) are the inventory ids of the films, not the number of unavailable films.

**inventory\_in\_stock:** This checks by inventory id if a certain DVD is in stock by checking if it has ever been rented, or if there are any rentals for that DVD that hasn't been returned, and returns TRUE if its in stock, and returns FALSE if it isnt.

**get\_customer\_balance:**

This funtction calculates a customers balance (ammount due – ammount paid in rental fees) at a certain pont in time.

It first returns the rental fee which a certain customer had to pay before the specified date into the v\_rentfees variable. Then it calculates and returns into the v\_overfees variable the customers overdue fees for their rentals which have exceeded the rental period – it checks this by comparing the rental period and the difference between the rental date and the return date (1 dollar for each overdue date). Then it returns the ammount of fees already paid in the time period by the customer into the v\_payments variable. Lastly, it calculates the balance by adding the rental fees and the overdue fees together and substracting the already paid ammount from it (v\_rentfees + v\_overfees - v\_payments).

**inventory\_held\_by\_customer:** This function takes a certain DVD’s inventory\_id as input parameter, and returns the customer\_id of the customer currently renting this DVD.

**rewards\_report:** This function takes a the minimum number of monthly purchases and the minimum ammount paid by a customer for the last 3 months, and returns the customer\_id of all customers who match this criteria.

**last\_day:** This function takes a timestamptz value as input, and returns the last day of the month that timestamp falls into by extracting the year from the timestamp, extracting the month from the timestamp, adding 1 to it, concatenating them to get ’year-next\_month-01’ and then substracting 1 day from it, to get the last day of the given timestamp’s month.

It manages december by CASE function, where if the extracted month from the timestamp = 12, then it adds 1 to the extracted year concatenates it with ’-01-01’, then subtracts 1 day from it to get ’year-12-31’.

(You can find these functions in dvd\_rental database.)

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

This happens because in the payment table the last payment date is from 2017-05-31, and the function counts the start date of the 3 month interval as

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

To correct this, we can specify last\_month\_start as follows:  
  
last\_month\_start := (**SELECT** **MAX**(p.payment\_date) **FROM** public.payment p) - '3 month'::**interval**;

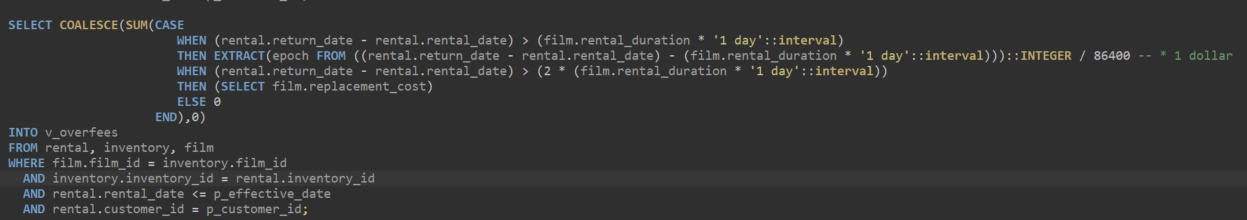
This way, the function always runs on the 3 month period starting from the last payment date present in the payments table - 3 months.

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

Yes, film\_not\_in\_stock, because it is just the inverse of the film\_in\_stock function. It serves no real business purpose, as usually one wants to check if a certain film is in stock, not the other way around. Also, film\_in\_stock implicitly can be used for its purpose: if it returns no values for a given film\_id and store\_id, it means that this film is not in stock in that store.

4. 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.

To impement the requirement „ 3) if a film is more than rental\_duration \* 2 overdue, charge the replacement\_cost”, we can modify the script selecting ammounts into the v\_overfees variable like this:



Added:

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

**THEN** (**SELECT** film.replacement\_cost)

This way when a film is overdue by more than 2 times its rental duration, then its replacment cost gets charged as overfee.

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

group\_concat is an aggregate, which uses the \_group\_concat function. It takes text type inputs from a specified coulmn of a table, and concatenates them, with comma separation.

The \_group\_concat function takes two text type inputs and concatenates them with comma separation. If either input parameter is null, then it returns just the other text value. If both are null, then it returns null.

The group\_concat aggregate loops the \_group\_concat function on a specified column. It keeps a text variabe in memory while running the aggregate (STYPE =text), which starts with a default null value. The aggreagte takes the first row’s value of the specified column, uses it as input parameter for the \_group\_concat function alongside the stored variable. The first time it wil be (null, ’first\_value’) so it returns ’first\_value’ and updates the variable to this state. Then it repeats for each row, always updating the variable, and at the end returns all the rows’ values separated by commas.

It is used in 3 views: actor\_info, film\_list and nicer\_but\_slower\_film\_list. It is used as follows:

actor\_info: to list the films each actor appeared in by category in a single column

film\_list: to list the actors who appear in each film in a single column

nicer\_but\_slower\_film\_list: the same as flm\_list just the casing is nicer

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

It is a trigger function, which automatically sets the affected rows’ value in the last\_update column to the current timestamp when performing INSERT or UPDATE operations.

It is used in the following tables before UPDATE operations:

A screenshot of a computer

AI-generated content may be incorrect.

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

The tmpSQL is a TEXT data type variable, which is used in the function to store code as TEXT, which can be run dynamically in the function using the EXECUTE command. It is first used to insert the customer\_ids of the customers who match the monthly purchase criterias into the created temporary table (tmpCustomer). Then it puts into the rr RECORD variable all the information of these customers from the customer table row by row. After that, tmpSQL is used for „celanup”, to drop the tmpCustomer temporary table. The function then returns all the data in rr as output.

The function can be recreated without EXECUTE and dynamic SQL using a subquery instead, like this:

A screenshot of a computer program

AI-generated content may be incorrect.

This way we skip the steps of inserting the customer\_ids into a temporary table and then returning their info into a record from the customer table. Instead, it returns customer information directly by joining the customer table with the returned customer\_ids of the subquery which checks which customers match the criteria.

Script of this CREATE FUNCTION operation in text format:

-- DROP FUNCTION public.rewards\_report(int4, numeric);

CREATE OR REPLACE FUNCTION public.rewards\_report3(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;

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 := (SELECT MAX(p.payment\_date) FROM public.payment p) - '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);

/\*

Find all customers meeting the monthly purchase requirements

\*/

RETURN QUERY

SELECT c.\*

FROM customer c

INNER JOIN (

SELECT p.customer\_id

FROM payment 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

) AS rewardees

ON c.customer\_id = rewardees.customer\_id;

END

$function$

;