

Anastasia Polskaya

January 2024

SQL Business Report Project (DVD Rental Company)

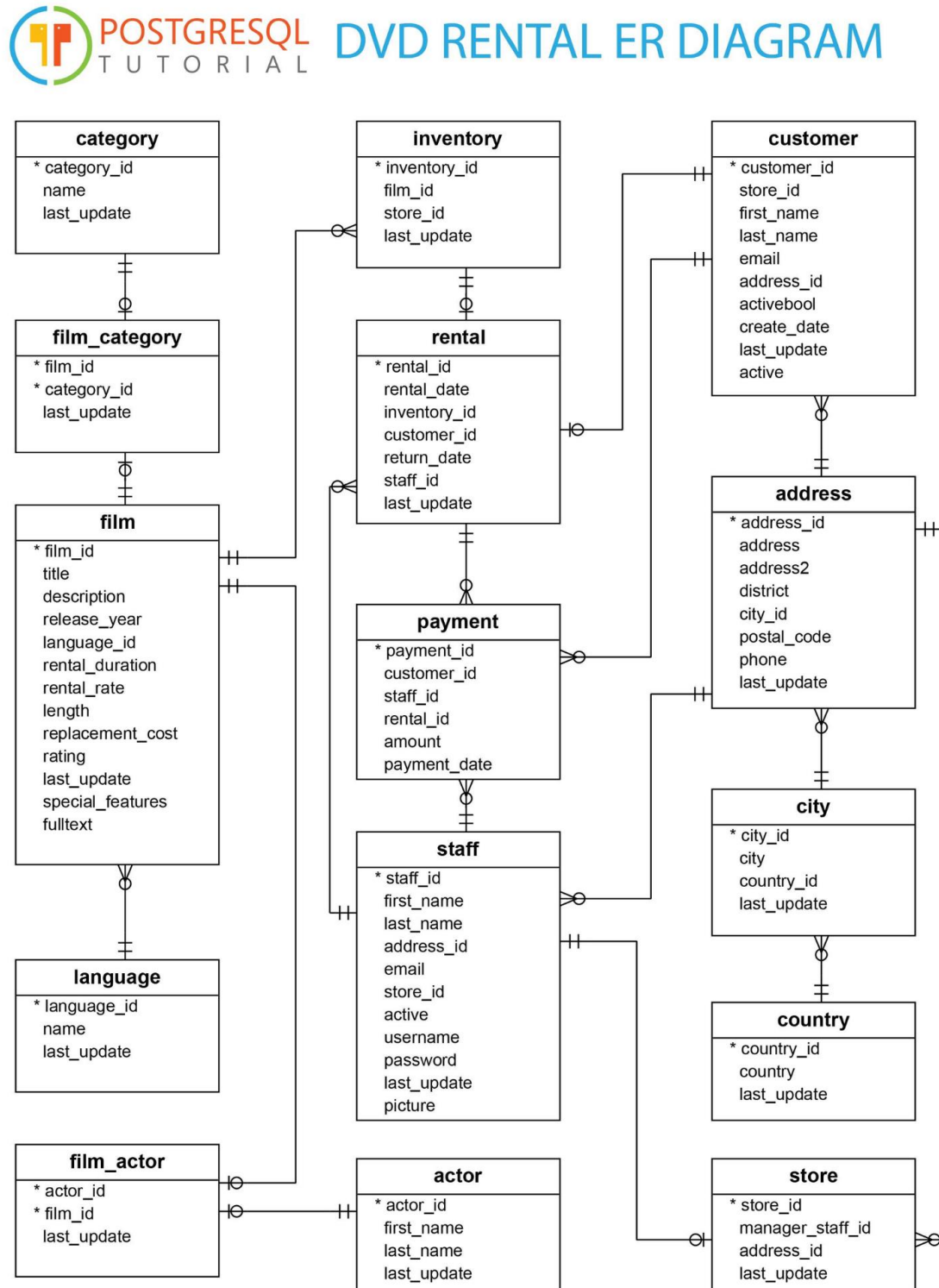
Business question:

A DVD rental company would like to know which film categories their customers are interested in, so they know which films to add to their stock. This report will summarize how many times each film category was rented, which will provide the information necessary for the company to make a decision. Investing in more films in the categories that are rented most often will help the company better align with their customers' interests.

Recommended report refresh frequency:

I recommend that this report be refreshed every 3 months. This allows stakeholders to continually stay informed of the customers' interests in film categories and be able to keep up with them. There should be a generous number of rentals made within 3 months, so there will be plenty of data to add to the tables and analyze. With this frequency the business can keep adding films from the top categories to their stores to keep their customers interested. There are many films releasing all year round, and a DVD rental business should be adding films to their stores consistently throughout the year. If the report were to be refreshed less often, such as every 6 months, the business would be way behind in serving their customers' interests. On the other hand, refreshing more often, such as every month, wouldn't be enough time to gather relevant rental data that will accurately portray the top categories.

Here is the ERD of the DVD rental database



1. First I identified all fields that I will need for the detailed and summary report tables

Field Name	Going into Summary or Detailed table?	Which database table is it from?	Field Datatype	Description of Field
rental_id	Detailed	Rental	integer	Stores the unique id for each rental
inventory_id	Detailed	Inventory	integer	Stores the unique id for each DVD the company owns/has at their stores
film_id	Detailed	Film Category	integer	Stores the unique id for each film
category_id	Detailed and Summary	Category	integer	Stores the unique id for each film category
category_name	Detailed and Summary	Category	string	Stores the name of each film category
number_of_rentals	Summary	N/A (Aggregate)	integer	This field is an aggregate of how many times each film category is rented

Business uses of the detailed and summary tables:

The detailed table provides information such as: which films are rented most often, which categories are rented most often, which category each film is, and how many rentals there have been in total. A stakeholder could use this information to evaluate their customers' interests and determine how to better align to these interests and/or needs. They could also see how many total rentals they have had so far to evaluate how the business is doing.

The summary table shows each film category and how many times each of them were rented. A stakeholder can see which film categories are rented the most often by their customers. They can use this table to decide which categories of films they should invest in/which categories of films they should add more of to their stores to better suit their customers.

2. Transformation with a user-defined function:

The category_id and category_name fields need a transformation with a user-defined function. It would be convenient to see both the category_id and category_name in one column, so we can reference which id corresponds to each category name. I will concatenate them so that the output field is “category_name (category_id)”. This transformation will allow us to have less columns in the summary table and will allow for easier reading of the summary data table.

```
--B. Create a user-defined function that performs the transformation identified in in part A4
CREATE OR REPLACE FUNCTION category(category_name VARCHAR(50), category_id INTEGER)
RETURNS VARCHAR(50) AS $$
BEGIN
    RETURN CONCAT(category_name, ' (' ,category_id, ')');
END; $$
LANGUAGE plpgsql

-- Testing the function for part B with random arguments
SELECT category('sports', '1');
```

Data Output Explain Messages

	category	
	character varying	🔒
1	sports (1)	

3. Creating the detailed and summary tables:

Here I am only creating the tables with their columns. No data is added just yet, because we want to later incorporate a trigger and stored procedure that will keep the tables up to date.

```
--C. Create the detailed and summary tables to hold your report table sections.  
-- DETAILED TABLE  
DROP TABLE IF EXISTS detailed;  
CREATE TABLE detailed(  
    rental_id INT,  
    inventory_id INT,  
    film_id INT,  
    category VARCHAR(50)  
);  
-- SUMMARY TABLE  
DROP TABLE IF EXISTS summary;  
CREATE TABLE summary(  
    category VARCHAR(50),  
    number_of_rentals INT  
);
```

Data Output	Explain	Messages
-------------	---------	----------






CREATE TABLE		
--------------	--	--

Query returned successfully in 41 msec.

4. Extracting raw data for insertion into the detailed table:

I used joins to gather all necessary columns from four different database tables. This part was a little bit tricky for me at first because I haven't had hands-on experience with joins before. With trial and error, as well as a tutorial or two, I managed to get it working! I learned a great deal about joins during this process.

```
--D. Extract the raw data needed for the detailed section of your report from the source database.
INSERT INTO detailed
  SELECT r.rental_id,r.inventory_id, i.film_id, category(c.name, c.category_id)
  FROM category c
  JOIN film_category fc
  USING (category_id)
  JOIN film f
  USING (film_id)
  JOIN inventory i
  USING (film_id)
  JOIN rental r
  USING (inventory_id);
--Testing the data extraction
SELECT * from detailed;
```

Data Output		Explain	Messages		
	 rental_id integer	 inventory_id integer	 film_id integer	 category character varying (50)	
1	2	1525	333	Music (12)	
2	3	1711	373	Children (3)	
3	4	2452	535	Horror (11)	
4	5	2079	450	Children (3)	
5	6	2792	613	Comedy (5)	
6	7	3995	870	Horror (11)	
7	8	2346	510	Animation (2)	
8	9	2580	565	Foreign (9)	
9	10	1824	396	Drama (7)	
10	11	4443	971	Foreign (9)	
11	12	1584	347	Travel (16)	
12	13	2294	499	Family (8)	

5. Creating a trigger on the detailed table that will update the summary table when data is added:

The trigger consists of the trigger function and the trigger statement. Since I wanted the summary table to have a column showing the number of rentals for each category, I incorporated into the trigger function a piece of code that will count the rentals for each category.

```
--E. Create a trigger on the detailed table of the report that will continually update the summary table
--Create trigger function
CREATE OR REPLACE FUNCTION trigger_function()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    DELETE FROM summary;
    INSERT INTO summary
    SELECT d.category, COUNT(r.rental_id)
    FROM rental r
    JOIN detailed d
    USING (rental_id)
    GROUP BY d.category
    ORDER BY count(r.rental_id) DESC;
    RETURN NEW;
END; $$;
--Create trigger statement
CREATE TRIGGER new_summary
AFTER INSERT ON detailed
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_function();
```

Data Output	Explain	Messages
-------------	---------	----------

CREATE TRIGGER		
----------------	--	--

Query returned successfully in 41 msec.		
---	--	--

6. Creating a stored procedure that can be called to refresh the data in both tables:

The stored procedure will clear both tables and perform the raw data extraction to populate the detailed table-- which will then trigger the summary to populate. As mentioned earlier, it is recommended that this report be refreshed every 3 months. The pgAgent job scheduling tool can be used to automate the stored procedure refresh. The tool can be scheduled to CALL the stored procedure every 3 months.

I've never heard of a stored procedure before, so this was a fun learning opportunity.

```
--F. Provide an original stored procedure in a text format that can be used to refresh the data in both
CREATE OR REPLACE PROCEDURE refresh()
LANGUAGE plpgsql
AS $$
DECLARE
BEGIN
    DELETE from detailed;
    DELETE from summary;
    INSERT INTO detailed
    SELECT r.rental_id, r.inventory_id, i.film_id, category(c.name, c.category_id)
    FROM category c
    JOIN film_category fc
    USING (category_id)
    JOIN film f
    USING (film_id)
    JOIN inventory i
    USING (film_id)
    JOIN rental r
    USING (inventory_id);
END; $$
```

Data Output Explain Messages

CREATE TRIGGER

Query returned successfully in 41 msec.

7. The results summary:

In the resulting summary table, we are able to see how many times each film category was rented. The sports category has the greatest number of rentals, totaling 1,179. The least rented category is music, with a total of 830 rentals. With this information the business stakeholders can determine which film categories they should stock up on to stay relevant to their customers' needs.

	category character varying (50)	number_of_rentals integer
1	Sports (15)	1179
2	Animation (2)	1166
3	Action (1)	1112
4	Sci-Fi (14)	1101
5	Family (8)	1096
6	Drama (7)	1060
7	Documentary (6)	1050
8	Foreign (9)	1033
9	Games (10)	969
10	Children (3)	945
11	Comedy (5)	941
12	New (13)	940
13	Classics (4)	939
14	Horror (11)	846
15	Travel (16)	837
16	Music (12)	830

Resources

PostgreSQL create trigger. PostgreSQL Tutorial. (n.d.).

<https://www.postgresqltutorial.com/postgresql-triggers/creating-first-trigger-postgresql/>

Koidan, K. (2020, August 6). *Can you join two tables without a common column?*.

LearnSQL.com. <https://learnsql.com/blog/join-tables-without-common-column/>

PostgreSQL CONCAT function. PostgreSQL Tutorial. (n.d.-a).

<https://www.postgresqltutorial.com/postgresql-string-functions/postgresql-concat-function/>

PostgreSQL create function statement. PostgreSQL Tutorial. (n.d.-b).

<https://www.postgresqltutorial.com/postgresql-plpgsql/postgresql-create-function/#:~:text=In%20the%20function%20body%3A,and%20logic%20of%20the%20function.>

Johnson, B. (n.d.). *Working with Triggers & Stored Procedures.* Panopto.

<https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=cdeef825-e36a-4e9a-a157-aeed0139b592>

Johnson, B. (n.d.-b). *Writing & Using Functions in PostgreSQL.* Panopto.

<https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=0eabc1aa-1e70-43ac-bb1e-addb013a26e8>

