

Disparadores y Vistas en SQL

Samuel Martín Morales

November 7, 2023

Contents

1	Introducción	2
2	Resultados	3
2.1	Ejercicio 1	3
2.2	Ejercicio 2	4
2.2.1	Ventas totales	4
2.2.2	Ventas totales por tienda	5
2.2.3	Lista de películas	6
2.2.4	información de los actores	7
2.3	Ejercicio 3	8
2.3.1	Vista 1: Ventas totales	8
2.3.2	Vista 2: Ventas totales por tienda	9
2.3.3	Vista 3: Lista de películas	9
2.3.4	Vista 4: Información de los actores	9
2.4	Ejercicio 4	9
2.5	Ejercicio 5	9
2.6	Ejercicio 6	9
2.7	Ejercicio 7	11
2.8	Ejercicio 8	12
3	Conclusiones	13
4	Bibliografía	14

Chapter 1

Introducción

Para esta cuarta práctica de la asignatura *Administración y Diseño de Bases de Datos* se solicita el empleo de una base de datos que debe de ser restaurada de manera previa a la implementación de una serie de ejercicios que son demandados haciendo uso de dicha base de datos.

En este caso, la base de datos a emplear se denomina como ***alquilerdvd.tar*** y se encuentra disponible en el campus virtual de la asignatura. Pero, puede ser descargada desde el siguiente enlace de GitHub.

Dicha base de datos se encuentra en formato *.tar* por lo que, para poder restaurarla, se debe de emplear el siguiente comando:

```
$ pg_restore -U postgres -d alquilerdvd alquilerdvd.tar
```

Es decir, el comando anterior restaura la base de datos *alquilerdvd* haciendo uso del fichero *alquilerdvd.tar* y empleando el usuario *postgres*.

Una vez restaurada la base de datos, se puede proceder a la realización de los distintos ejercicios.

Chapter 2

Resultados

2.1 Ejercicio 1

Para el primer ejercicio de la práctica, se deben de identificar las distintas tablas, vistas y secuencias que tiene la base de datos que ha sido restaurada.

Tras la carga de la base de datos a partir del fichero con extensión *.tar*, se puede observar que la base de datos *alquilerdvd* cuenta con un total de 15 tablas, 0 vistas y 15 secuencias. Para poder visualizar todos estos datos comentados sobre la base de datos, se hace uso de la terminal interactiva de *PostgreSQL*, es decir, de *psql*, y, una vez dentro de la base de datos se ejecutan los siguientes comandos para obtener los distintos valores obseados anteriormente:

```
# \dt -- Muestra las tablas de la base de datos
# \dv -- Muestra las vistas de la base de datos
# \ds -- Muestra las secuencias de la base de datos
```

Para comprobar todo esto anterior se puede observar la siguiente imagen 2.1:

```

postgres=# \c rent
You are now connected to database "rent" as user "postgres".
rent=# \dt
      List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | actor          | table | postgres
public | address        | table | postgres
public | category       | table | postgres
public | city           | table | postgres
public | country        | table | postgres
public | customer       | table | postgres
public | deleting_film_rows | table | postgres
public | film           | table | postgres
public | film_actor     | table | postgres
public | film_category  | table | postgres
public | inventory      | table | postgres
public | language       | table | postgres
public | payment        | table | postgres
public | rental         | table | postgres
public | staff          | table | postgres
public | store          | table | postgres
public | updated_table_film | table | postgres
(17 rows)

rent=# \dv
      List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | actor_list      | view | postgres
public | films_list      | view | postgres
public | total_rent_per_category | view | postgres
public | total_rent_per_store | view | postgres
(4 rows)

rent=# \ds
      List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | actor_actor_id_seq | sequence | postgres
public | address_address_id_seq | sequence | postgres
public | category_category_id_seq | sequence | postgres
public | city_city_id_seq | sequence | postgres
public | country_country_id_seq | sequence | postgres
public | customer_customer_id_seq | sequence | postgres
public | deleting_film_rows_delete_id_seq | sequence | postgres
public | film_film_id_seq | sequence | postgres
public | inventory_inventory_id_seq | sequence | postgres
public | language_language_id_seq | sequence | postgres
public | payment_payment_id_seq | sequence | postgres
public | rental_rental_id_seq | sequence | postgres
public | staff_staff_id_seq | sequence | postgres
public | store_store_id_seq | sequence | postgres
public | updated_table_film_id_updated_table_film_seq | sequence | postgres
(15 rows)

rent=#

```

Figure 2.1: Ejecución de los comandos dt, dv, ds en la terminal psql.

2.2 Ejercicio 2

Tras la identificación de las distintas tablas más importantes de la base de datos junto con sus atributos y relaciones entre las distintas tablas, se procede a la implementación de distintas consultas que permitan obtener aquella información que es solicitada en el enunciado del ejercicio.

2.2.1 Ventas totales

Para obtener las ventas totales por categoría de películas ordenadas de manera descendente, se emplea la siguiente consulta:

```

SELECT COUNT(*) AS total_rent, category.name AS category_name
FROM rental
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
GROUP BY category_name

```

ORDER BY total_rent DESC;

Resultado de la consulta anterior 2.2:

```
1  SELECT COUNT(*) AS total_rent, category.name AS category_name
2  FROM rental
3  INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
4  INNER JOIN film ON inventory.film_id = film.film_id
5  INNER JOIN film_category ON film.film_id = film_category.film_id
6  INNER JOIN category ON film_category.category_id = category.category_id
7  GROUP BY category_name
8  ORDER BY total_rent DESC;
```

1. Query			
#	total_rent INT8	category_name ...	
1	1179	Sports	
2	1166	Animation	
3	1112	Action	
4	1101	Sci-Fi	
5	1096	Family	
6	1060	Drama	
7	1050	Documentary	
8	1033	Foreign	
9	969	Games	
10	945	Children	
11	941	Comedy	
12	940	New	
13	880	Classics	

16 rows 249 ms 10 seconds ago

Figure 2.2: Resultado de la consulta de las ventas totales por categoría.

2.2.2 Ventas totales por tienda

Para obtener las ventas totales por tienda donde se refleja la ciudad, el país y el encargado, se emplea la siguiente consulta:

```
SELECT COUNT(*) AS total_rent , store.store_id AS store_id, city.city || ', ' || country.country AS cityu_
FROM rental
INNER JOIN inventory on rental.inventory_id = inventory.inventory_id
INNER JOIN store ON inventory.store_id = store.store_id
INNER JOIN staff ON store.manager_staff_id = staff.staff_id
INNER JOIN address ON store.address_id = address.address_id
INNER JOIN city ON address.city_id = city.city_id
INNER JOIN country ON city.country_id = country.country_id
```

GROUP BY store.store_id, manager_staff_first_name, manager_staff_last_name, city, country
ORDER BY total_rent DESC;

Resultado de la consulta anterior 2.3:

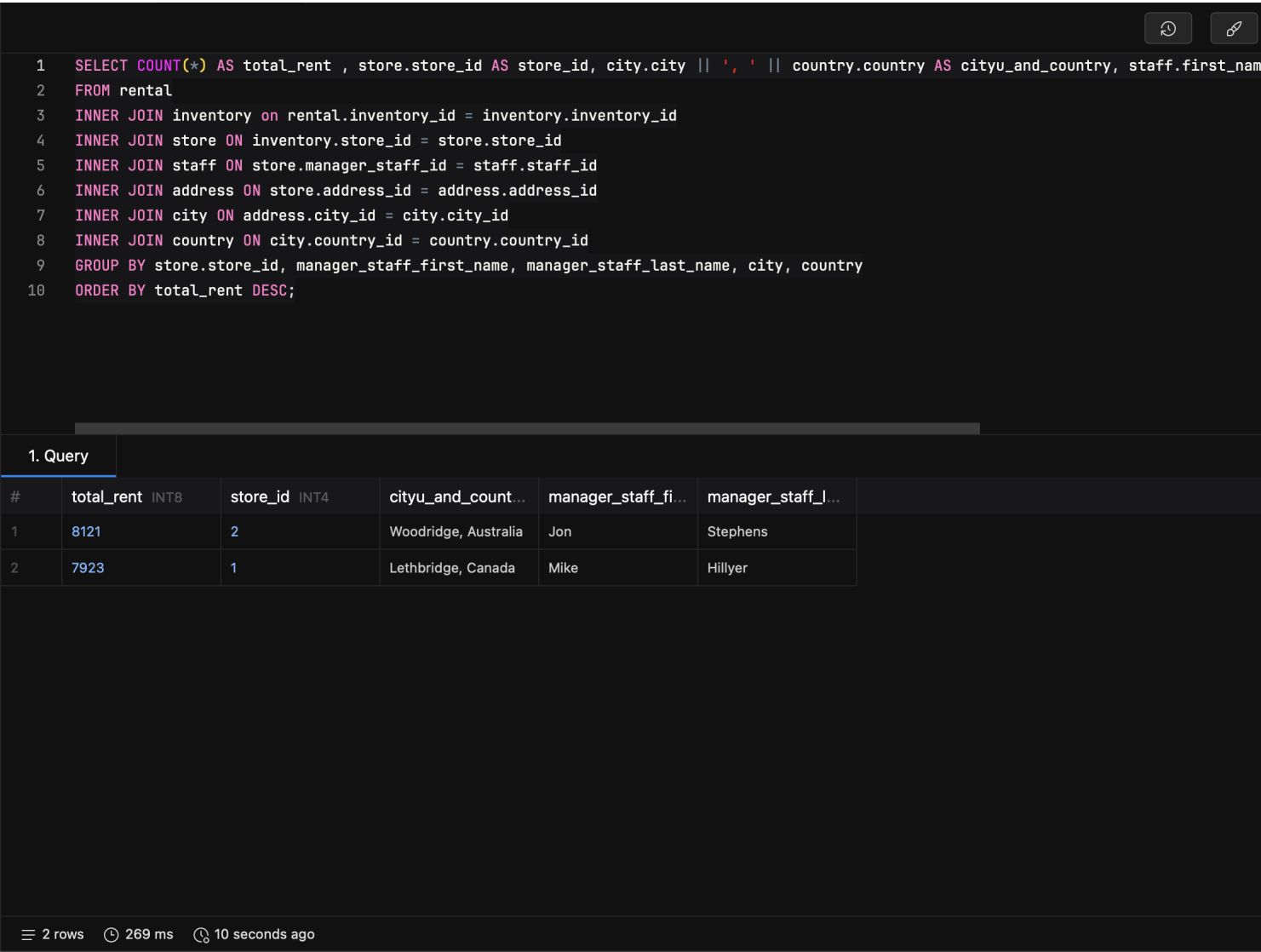


Figure 2.3: Resultado de la consulta de las ventas totales por tienda.

Nota: para la consulta anterior se ha empleado una concatenación de cadenas de caracteres para poder obtener la ciudad y el país en una misma columna, para ello, se ha hecho uso de la doble barra vertical que permite establecer la concatenación por pares de elementos, y para el ejemplo de consulta observado anteriormente, se hace uso del separador “,” para realizar esta operación.

2.2.3 Lista de películas

Para obtener una lista de películas junto con sus actores, se emplea la siguiente consulta:

SELECT film.film_id, title, description, category.name AS category_name, rental_rate, length, rating , actor

```

FROM film
INNER JOIN film_actor ON film.film_id = film_actor.film_id
INNER JOIN actor ON film_actor.actor_id = actor.actor_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
ORDER BY film_id;

```

Resultado de la consulta anterior 2.4:

```

1  SELECT film.film_id, title, description, category.name AS category_name, rental_rate, length, rating, actor.first_name || ' ' || actor
2  FROM film
3  INNER JOIN film_actor ON film.film_id = film_actor.film_id
4  INNER JOIN actor ON film_actor.actor_id = actor.actor_id
5  INNER JOIN film_category ON film.film_id = film_category.film_id
6  INNER JOIN category ON film_category.category_id = category.category_id
7  ORDER BY film_id;

```

#	film_id	title	description	category_name	rental_rate	length	rating	actor_name
1	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Rock Dukaki
2	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Mary Keitel
3	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Johnny Cag
4	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Penelope Gu
5	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Sandra Peck
6	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Christian Ga
7	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Oprah Kilme
8	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Warren Nolt
9	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Lucille Tracy
10	1	Academy Dinosaur	A Epic Drama of a F...	Documentary	0.9900	86	Unsupported value	Mena Templ
11	2	Ace Goldfinger	A Astounding Epistl...	Horror	4.9900	48	Unsupported value	Minnie Zellw
12	2	Ace Goldfinger	A Astounding Epistl...	Horror	4.9900	48	Unsupported value	Chris Depp

5462 rows 1.164 s 10 seconds ago

Figure 2.4: Resultado de la consulta de la lista de películas.

2.2.4 información de los actores

Para obtener información de los distintos actores junto con sus películas existentes en la base de datos, se implementa la siguiente consulta:

```

SELECT actor.actor_id, actor.first_name, actor.last_name, film.title || ' : ' || film.description || ' : '
FROM actor
INNER JOIN film_actor ON actor.actor_id = film_actor.actor_id

```



```

INNER JOIN film ON film_actor.film_id = film.film_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name, films_made
ORDER BY actor.actor_id;

```

Resultado de la consulta anterior 2.5:

```

1 SELECT actor.actor_id, actor.first_name, actor.last_name, film.title || ' : ' || film.description || ' : ' || category.name AS films_ma
2 FROM actor
3 INNER JOIN film_actor ON actor.actor_id = film_actor.actor_id
4 INNER JOIN film ON film_actor.film_id = film.film_id
5 INNER JOIN film_category ON film.film_id = film_category.film_id
6 INNER JOIN category ON film_category.category_id = category.category_id
7 GROUP BY actor.actor_id, actor.first_name, actor.last_name, films_made
8 ORDER BY actor.actor_id;

```

1. Query					
#	actor_id INT4	first_name VARC...	last_name VARC...	films_made TEXT	
1	1	Penelope	Guiness	Academy Dinosaur : ...	
2	1	Penelope	Guiness	Anaconda Confessi...	
3	1	Penelope	Guiness	Angels Life : A Thou...	
4	1	Penelope	Guiness	Bulworth Command...	
5	1	Penelope	Guiness	Cheaper Clyde : A E...	
6	1	Penelope	Guiness	Color Philadelphia : ...	
7	1	Penelope	Guiness	Elephant Trojan : A ...	
8	1	Penelope	Guiness	Gleaming Jawbreak...	
9	1	Penelope	Guiness	Human Graffiti : A B...	
10	1	Penelope	Guiness	King Evolution : A A...	
11	1	Penelope	Guiness	Lady Stage : A Bea...	
12	1	Penelope	Guiness	Language Cowboy : ...	

≡ 5462 rows ⌚ 1.388 s ⌚ 10 seconds ago

Figure 2.5: Resultado de la consulta de la información de los actores.

2.3 Ejercicio 3

Implementación de todas las vistas a partir de las consultas realizadas en el ejercicio anterior.

2.3.1 Vista 1: Ventas totales

Para la implementación de la primera vista, se emplea la siguiente consulta:

```

CREATE VIEW total_rent_per_category AS
SELECT COUNT(*) AS total_rent, category.name AS category_name

```

```

FROM rental
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
GROUP BY category_name
ORDER BY total_rent DESC;

```

2.3.2 Vista 2: Ventas totales por tienda

Para la implementación de la segunda vista, se emplea la siguiente consulta:

```

CREATE VIEW total_rent_per_store AS
SELECT COUNT(*) AS total_rent , store.store_id AS store_id, city.city || ', ' || country.country AS cityu_
FROM rental
INNER JOIN inventory on rental.inventory_id = inventory.inventory_id
INNER JOIN store ON inventory.store_id = store.store_id
INNER JOIN staff ON store.manager_staff_id = staff.staff_id
INNER JOIN address ON store.address_id = address.address_id
INNER JOIN city ON address.city_id = city.city_id
INNER JOIN country ON city.country_id = country.country_id
GROUP BY store.store_id, manager_staff_first_name, manager_staff_last_name, city, country
ORDER BY total_rent DESC;

```

2.3.3 Vista 3: Lista de películas

Para la implementación de la tercera vista, se emplea la siguiente consulta:

```

CREATE VIEW films_list AS
SELECT film.film_id, title, description, category.name AS category_name, rental_rate, length, rating ,actor
FROM film
INNER JOIN film_actor ON film.film_id = film_actor.film_id
INNER JOIN actor ON film_actor.actor_id = actor.actor_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
ORDER BY film_id;

```

2.3.4 Vista 4: Información de los actores

Para la implementación de la cuarta vista, se emplea la siguiente consulta:

```

CREATE VIEW actor_list AS
SELECT actor.actor_id, actor.first_name, actor.last_name, film.title || ' : ' || film.description || ' : '
FROM actor
INNER JOIN film_actor ON actor.actor_id = film_actor.actor_id
INNER JOIN film ON film_actor.film_id = film.film_id
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id = category.category_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name, films_made
ORDER BY actor.actor_id;

```

2.4 Ejercicio 4

2.5 Ejercicio 5

2.6 Ejercicio 6

Para el sexto ejercicio solicitado, se debe de contruir un disparador que se encargue de guardar en una nueva tabla creada la fecha de cuando se insertó un nuevo registro dentro de la tabla *film*.

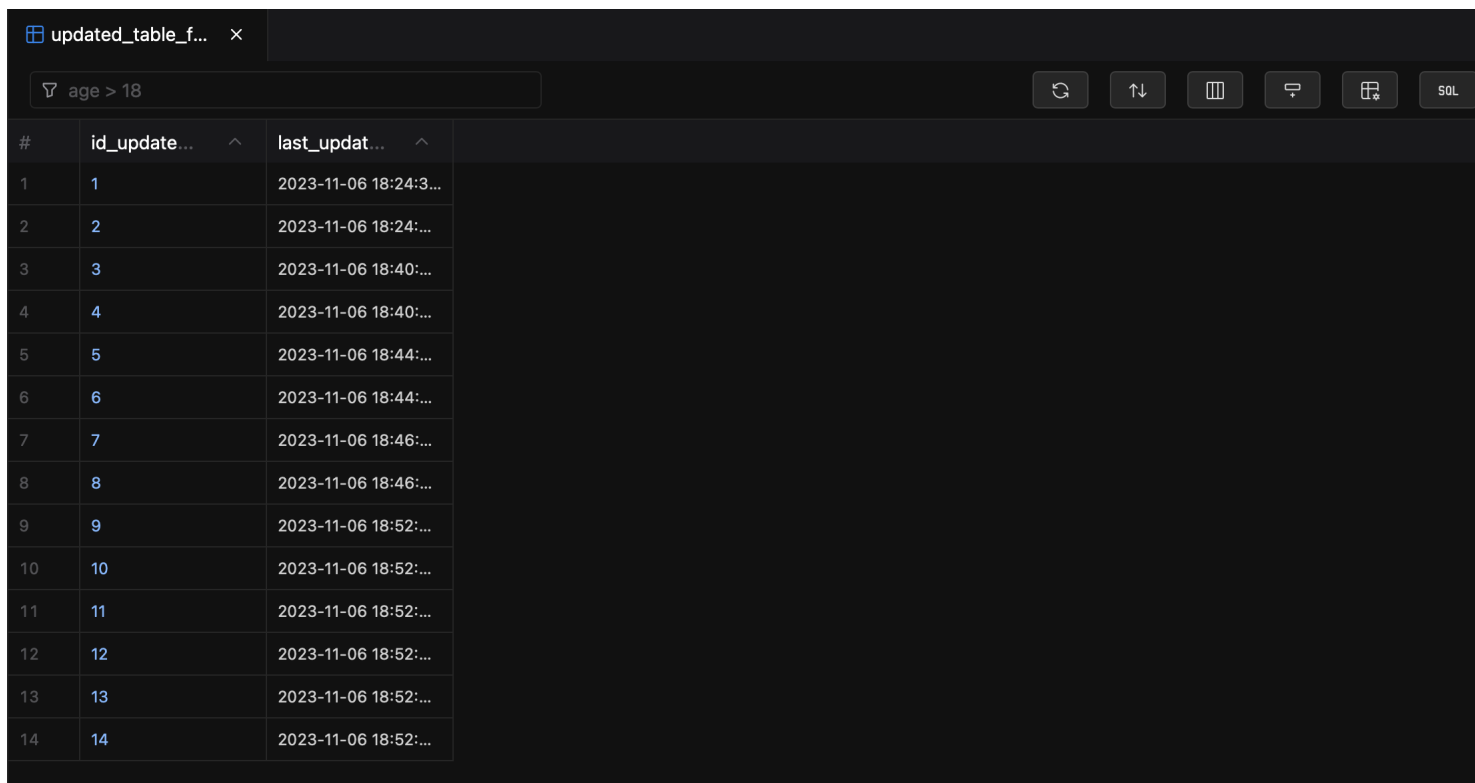
Cabe destacar que para la implementación de dicho disparador se ha creado de manera previa una nueva tabla denominada como *updated-table-film*, la cual tiene la siguiente estructura:

```
CREATE TABLE updated_table_film (  
  id_updated_table_film SERIAL PRIMARY KEY,  
  last_update TIMESTAMP NOT NULL  
);
```

Una vez creada la tabla, se procede a la implementación del disparador, para ello se emplea la siguiente consulta:

```
-- First we create the function  
CREATE FUNCTION delete_table_film() RETURNS TRIGGER AS $$  
BEGIN  
  INSERT INTO updated_table_film (last_update) VALUES (NOW());  
  RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
-- Then we create the trigger  
CREATE TRIGGER trigger_delete_table_film  
AFTER INSERT ON film  
FOR EACH ROW  
EXECUTE FUNCTION delete_table_film();
```

Comprobación del funcionamiento del disparador 2.6:



The screenshot shows a database client interface with a tab titled 'updated_table_f...'. Below the tab is a search bar containing 'age > 18'. To the right of the search bar are several icons: a circular arrow, a double-headed vertical arrow, a table icon, a document icon, a plus icon, and a button labeled 'SQL'. Below these elements is a table with two columns: 'id_update...' and 'last_updat...'. The table contains 14 rows of data, numbered 1 through 14 in the first column. The second column contains timestamps starting from '2023-11-06 18:24:3...' and ending with '2023-11-06 18:52:...'. The table is displayed in a dark-themed interface.

#	id_update...	last_updat...
1	1	2023-11-06 18:24:3...
2	2	2023-11-06 18:24:...
3	3	2023-11-06 18:40:...
4	4	2023-11-06 18:40:...
5	5	2023-11-06 18:44:...
6	6	2023-11-06 18:44:...
7	7	2023-11-06 18:46:...
8	8	2023-11-06 18:46:...
9	9	2023-11-06 18:52:...
10	10	2023-11-06 18:52:...
11	11	2023-11-06 18:52:...
12	12	2023-11-06 18:52:...
13	13	2023-11-06 18:52:...
14	14	2023-11-06 18:52:...

Figure 2.6: Comprobación del funcionamiento del disparador, tabla updated-table-film.

2.7 Ejercicio 7

Para este ejercicio se solicita la construcción de un disparador que guarde en una nueva tabla creada la fecha de cuando se eliminó un registro en la tabla *film*.

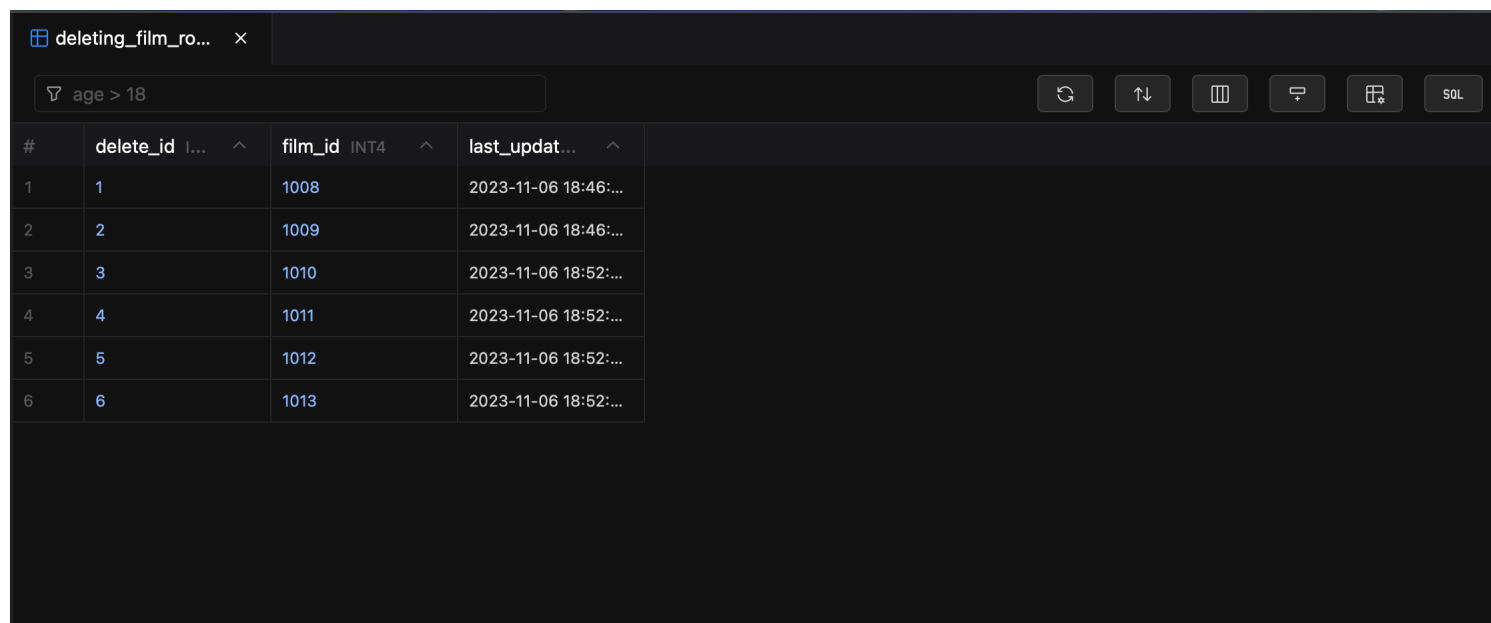
Para la implementación de dicho disparador se ha creado de manera previa una nueva tabla denominada como *deleting-film-rows*, la cual tiene la siguiente estructura:

```
CREATE TABLE deleting_film_rows (  
  delete_id SERIAL PRIMARY KEY,  
  film_id INT NOT NULL,  
  last_update TIMESTAMP NOT NULL  
);
```

Una vez creada la tabla, se procede a la implementación del disparador, para ello se emplea la siguiente consulta:

```
-- First we create the function  
CREATE FUNCTION delete_table_film() RETURNS TRIGGER AS $$  
BEGIN  
  INSERT INTO deleting_film_rows (film_id, last_update)  
  VALUES (OLD.film_id, NOW());  
  RETURN OLD;  
END;  
$$ LANGUAGE plpgsql;  
  
-- Then we create the trigger  
CREATE TRIGGER delete_film_trigger  
AFTER DELETE ON film  
FOR EACH ROW  
EXECUTE PROCEDURE delete_table_film();
```

Comprobación del funcionamiento del disparador 2.7:



#	delete_id	film_id	last_update
1	1	1008	2023-11-06 18:46:00
2	2	1009	2023-11-06 18:46:00
3	3	1010	2023-11-06 18:52:00
4	4	1011	2023-11-06 18:52:00
5	5	1012	2023-11-06 18:52:00
6	6	1013	2023-11-06 18:52:00

Figure 2.7: Comprobación del funcionamiento del disparador, tabla deleting-film-rows.

2.8 Ejercicio 8

Para el octavo y último ejercicio de la práctica se solicita la explicación del término *sequence* y su uso en *PostgreSQL*.

Es por ello que, las *sequence* en postgresql son objetos que se utilizan para generar valores numéricos de manera secuencial. Estos valores pueden ser utilizados para generar valores de clave primaria o valores de cualquier otra columna.

Es bastante útil e importante debido a que garantiza la unicidad y la no repetición de valores en una columna, sobre todo importante para el caso de las columnas de clave primaria o identificadores únicos.

Un ejemplo de creación de una *sequence* en *PostgreSQL* sería el siguiente:

```
CREATE SEQUENCE nombre_de_la_secuencia START 1 INCREMENT 2;
```

Chapter 3

Conclusiones

Example....

Chapter 4

Bibliografía

Autor(es). (Año). Título de la página o artículo. Nombre del Sitio web. URL

1. OpenAI. (2023). ChatGPT. OpenAI. <https://chat.openai.com/>
2. PostgreSQL. (2023). PostgreSQL: Documentation: 13: CREATE SEQUENCE. PostgreSQL. <https://www.postgresql.org/docs/13/sql-createsequence.html>
3. PostgreSQL. (2023). PostgreSQL: Documentation: 13: CREATE TRIGGER. PostgreSQL. <https://www.postgresql.org/docs/13/sql-createtrigger.html>
4. PostgreSQL. (2023). PostgreSQL: Documentation: 13: CREATE VIEW. PostgreSQL. <https://www.postgresql.org/docs/13/sql-createview.html>
5. PostgreSQL. (2023). PostgreSQL: Documentation: 13: CREATE FUNCTION. PostgreSQL. <https://www.postgresql.org/docs/13/sql-createfunction.html>
6. Samuel Martín Morales. (2023). PostgreSQL-Rent. GitHub. <https://github.com/Samuelmm15/PostgreSQL-Rent.git>