Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Vistas y disparadores:

Administración y diseño de bases de datos

Héctor Abreu Acosta
(alu0101068855@ull.edu.es)
Manuel Ramón Regalado Peraza
(alu0100283433@ull.edu.es)
Marcos Jesús Santana Ramos
(alu0101033471@ull.edu.es)

Índice:

1. Introducción

Actividad con el objetivo de continuar desarrollando habilidades en las operaciones básicas con el SQL y desarrollar actividades básicas con vistas y disparadores.

2. Ejercicio

1) Realice la restauración de la base de datos alquilerdvd.tar.

Una vez realizado el comando :

```
$ pg_restore -U postgres -w -d practica5 ./alquilerpractica.tar
```

Se comprueba que no da ningún tipo de error y proseguimos con el siguiente comando:.

```
$ sudo pg_dump -U postgres -h localhost practica5 > practica5.sql
```

Lo que hace el comando anterior es volcar la base de datos a *practica5.sql*

2) Identifique las tablas, vistas y secuencias.

Tablas:

```
practica5=# \dt
practica5=# \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 | 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
(15 rows)
```

Vistas:

No incluye vistas en un punto inicial.

```
practica5=# \dv
Did not find any relations.
practica5=#
```

Secuencias:

```
practica5=# \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 | 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
(13 rows)
```

3) Identifique las tablas principales y sus principales elementos.

Film:
(PK) film_id, title.

Payment:
(PK) payment_id, (FK) customer_id, (FK) rental_id, (FK) staff_id, amount

Customer:
(PK) customer_id, first_name, last_name, email.

Staff:
(PK) staff_id, first_name, last_name. email.

Address:
(PK) address_id, address, address2, phone.

Rental: (todas)
(PK) rental_id, (FK) customer_id, (FK) inventory_id, (FK) staff_id, rental_date, return_date

Inventory: (todas)
(PK) inventory_id, (FK) film_id, (FK) Store_id

Store: (todas)
(PK) store_id, (FK) address_id, (FK) manager_staff_id

4) Realice las siguientes consultas.

a. Obtenga las ventas totales por categoría de películas ordenadas descendentemente.

```
SELECT category.name AS categoria, COUNT(payment.amount) AS ventas_totales
FROM category, film_category, film, inventory, rental, payment
WHERE category.category_id = film_category.category_id AND film_category.film_id = film.film_id AND film.film_id = inventory.film_id AND inventory.inventory_id = rental.inventory_id AND rental.rental_id = payment.rental_id
GROUP BY category.name
ORDER BY COUNT(payment.amount) DESC;
```

b. Obtenga las ventas totales por tienda, donde se refleje la ciudad, el país (concatenar la ciudad y el país empleando como separador la “,”), y el encargado. Pudiera emplear GROUP BY, ORDER BY

```
SELECT store.store_id, CONCAT(staff.first_name, ' ', staff.last_name) AS manager_staff_name, CONCAT(city.city, ', ', country.country) AS city_country, COUNT(payment.amount) AS total_ventas
FROM  store, staff, address, city, country, payment
WHERE store.manager_staff_id = staff.staff_id AND staff.store_id = store.store_id AND store.address_id = address.address_id AND address.city_id = city.city_id AND city.country_id = country.country_id AND staff.staff_id = payment.staff_id
GROUP BY store.store_id, manager_staff_name, city_country
ORDER BY store.store_id;
```

c. Obtenga una lista de películas, donde se reflejen el identificador, el título, descripción, categoría, el precio, la duración de la película,

clasificación, nombre y apellidos de los actores (puede realizar una concatenación de ambos). Pudiera emplear GROUP BY

```
SELECT film.film_id, title, description, category.name AS category, COUNT(payment.amount) AS cost, length AS duration, rating, STRING_AGG(actor.first_name || ' ' || actor.last_name, ',') as actores
FROM film, category, payment, actor, inventory, rental, film_category, film_actor
WHERE film.film_id = film_category.film_id and film_category.category_id = category.category_id and film.film_id = inventory.film_id and inventory.inventory_id = rental.inventory_id and rental.rental_id = payment.rental_id and film.film_id = film_actor.film_id and film_actor.actor_id = actor.actor_id
group by film.film_id, category;
```

d. Obtenga la información de los actores, donde se incluya sus nombres y apellidos, las categorías y sus películas. Los actores deben de estar agrupados y, las categorías y las películas deben estar concatenados por “:”

Si lo que se pide es categorías:películas con sus actores:

```
SELECT CONCAT(category.name, ':  ', film.title) AS category_film,
STRING_AGG(actor.first_name || ' ' || actor.last_name, ',') as actores
FROM actor, film_actor, film, film_category, category
WHERE actor.actor_id = film_actor.actor_id AND film.film_id = film_actor.film_id AND film.film_id = film_category.film_id AND category.category_id = film_category.category_id
GROUP BY category_film;
```

Si lo que se pide es actores con todas sus categorías:películas de cada actor:

```
SELECT Concat(actor.first_name || ' ' || actor.last_name, ',') as actores
, STRING_AGG(category.name || ':' || film.title, ',') AS category_film
FROM actor, film_actor, film, film_category, category
WHERE actor.actor_id = film_actor.actor_id AND film.film_id = film_actor.film_id AND film.film_id = film_category.film_id AND category.category_id = film_category.category_id
GROUP BY actores;
```

5) Realice todas las vistas de las consultas anteriores. Coloque el prefijo view_ a su denominación.

Las vistas han sido denominadas como view_ seguido de la letra que le corresponde del apartado anterior. En el caso del apartado D al tener dos opciones, han sido denominadas como d1 y d2.

a. view_a

```
CREATE VIEW view_a AS
SELECT category.name AS categoria, COUNT(payment.amount) AS ventas_totales
FROM category, film_category, film, inventory, rental, payment
WHERE category.category_id = film_category.category_id AND film_category.film_id = film.film_id AND film.film_id = inventory.film_id AND inventory.inventory_id = rental.inventory_id AND rental.rental_id = payment.rental_id
GROUP BY category.name
ORDER BY COUNT(payment.amount) DESC;
```

b. view_b

```
CREATE VIEW view_b AS
SELECT store.store_id, CONCAT(staff.first_name, ' ', staff.last_name) AS
manager_staff_name, CONCAT(city.city, ', ', country.country) AS city_country,
COUNT(payment.amount) AS total_ventas
FROM  store, staff, address, city, country, payment
WHERE store.manager_staff_id = staff.staff_id AND staff.store_id = store.store_id AND
store.address_id = address.address_id AND address.city_id = city.city_id AND
city.country_id = country.country_id AND staff.staff_id = payment.staff_id
GROUP BY store.store_id, manager_staff_name, city_country
ORDER BY store.store_id;
```

c. view_c

```
CREATE VIEW view_c AS
SELECT film.film_id, title, description, category.name AS category,
COUNT(payment.amount) AS cost, length AS duration, rating,
STRING_AGG(actor.first_name || ' ' || actor.last_name, ',') as actores
FROM film, category, payment, actor, inventory, rental, film_category, film_actor
WHERE film.film_id = film_category.film_id and film_category.category_id =
category.category_id and film.film_id = inventory.film_id and inventory.inventory_id =
rental.inventory_id and rental.rental_id = payment.rental_id and film.film_id =
film_actor.film_id and film_actor.actor_id = actor.actor_id
group by film.film_id, category;
```

d. view_d1

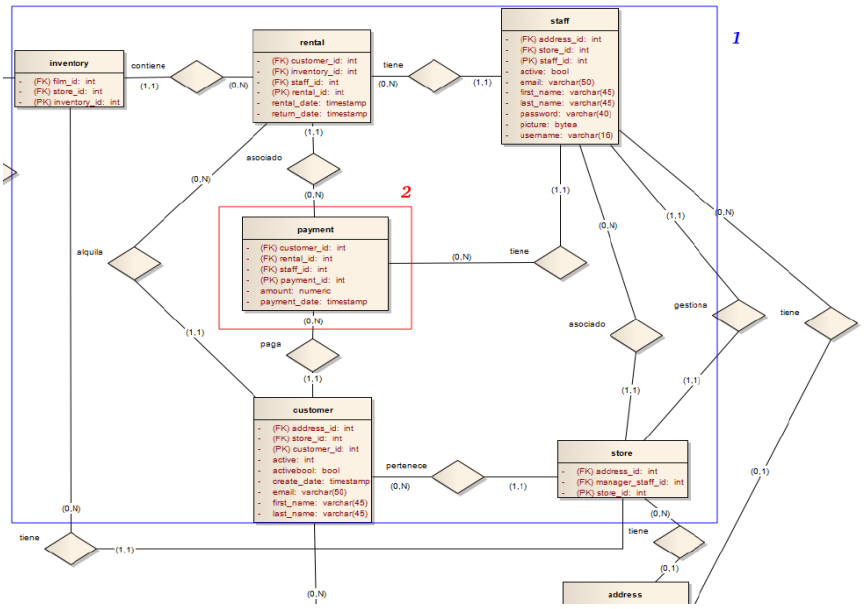
```
CREATE VIEW view_d1 AS
SELECT CONCAT(category.name, ': ', film.title) AS category_film, STRING_AGG(actor.first_name || ' ' || actor.last_name, ',') as actores
FROM actor, film_actor, film, film_category, category
WHERE actor.actor_id = film_actor.actor_id AND film.film_id = film_actor.film_id AND film.film_id = film_category.film_id AND category.category_id = film_category.category_id
GROUP BY category_film;
```

e. view_d2

```
CREATE VIEW view_d2 AS
SELECT CONCAT(actor.first_name || ' ' || actor.last_name, ',') as actores, STRING_AGG(category.name || ':' || film.title, ',') AS category_film
FROM actor, film_actor, film, film_category, category
WHERE actor.actor_id = film_actor.actor_id AND film.film_id = film_actor.film_id AND film.film_id = film_category.film_id AND category.category_id = film_category.category_id
GROUP BY actores;
```

6) Haga un análisis del modelo e incluya las restricciones Check que considere necesarias.

Nos dimos cuenta que el 'núcleo del funcionamiento' de la base de datos gira entorno al segmento de la misma que muestra esta imagen :



Como el sentido del negocio es alquilar DVDs, los elementos principales para hacer esa operación y las relacionadas confluyen en las tablas y sus relaciones que podemos ver enmarcadas dentro del cuadrado azul de la imagen superior.

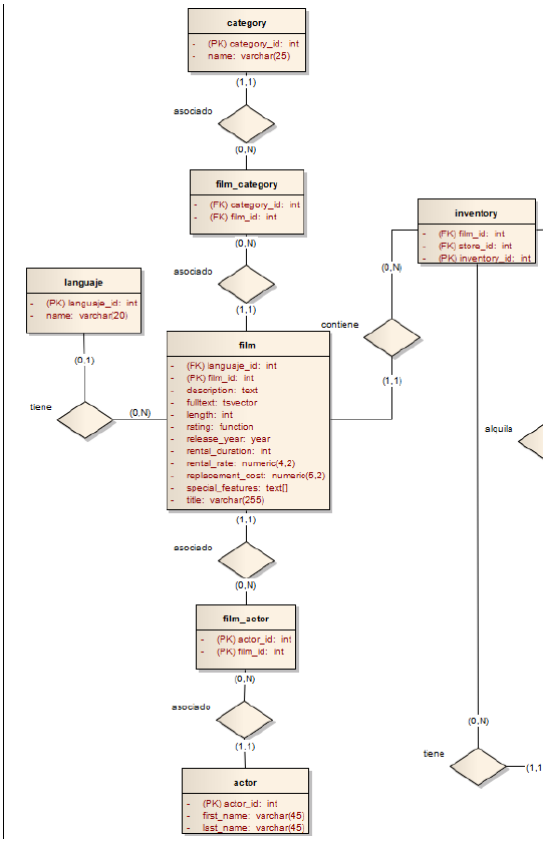
Son las tablas *inventory*, *rental*, *staff*, *payment*, *customer* y *store*, que contienen los datos del personal, el inventario del videoclub, los clientes registrados, las operaciones de alquiler/pagos y la tienda en sí misma.

Por supuesto, las tablas mencionadas pueden depender o no de otras tablas fuera del 'núcleo' (la imagen mostrada).

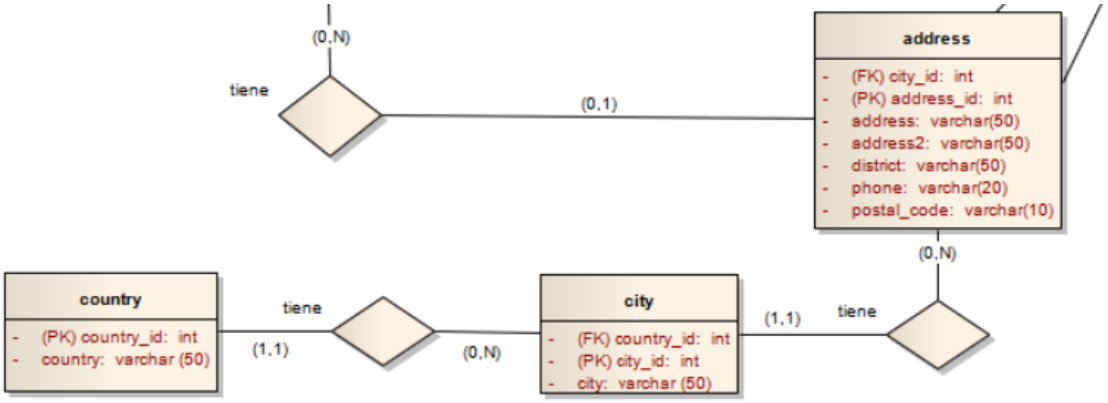
Creemos además, que todo confluye en la tabla central de la imagen *payment*, rodeada del cuadrado rojo. Vemos que depende de las tablas de 'alquiler', 'personal' y 'cliente' (*rent*, *staff* y *customer* respectivamente).

De las tablas restantes, van a estar asociadas entorno a:

- La película a alquilar: En *language* los datos de lenguaje de la película, en *actor* los datos de los actores, en *film_actor* todos los actores que participan en una película, en *film_category* todas las categorías que tiene una película y en *film* los datos generales de cada película.



- La localización del lugar (los videoclubes) : Datos de la dirección de la tienda en address y otras tablas con los datos de países (*country*) y ciudades (*city*) de tales tiendas.



Checks añadidos:

- El pago del alquiler no puede ser un valor negativo (tabla 'payment'):

```
ALTER TABLE payment
ADD CONSTRAINT pago_no_negativo
CHECK (
 amount > -1
);
```

- El password del personal (staff) debe ser al menos de tamaño 6:

```
ALTER TABLE staff
ADD CONSTRAINT password_al_menos_6
CHECK (
 char_length(password) > 6
);
```

- La fecha de devolución debe ser mayor que la fecha de alquiler:

```
ALTER TABLE rental
ADD CONSTRAINT fecha_devolucion_mayor_fecha_alquiler
CHECK (
 rental_date < return_date
);
```

7) Explique la sentencia que aparece en la tabla customer
Triggers:
last_updated BEFORE UPDATE ON customer FOR EACH ROW EXECUTE PROCEDURE last_updated()

Antes de actualizar la tabla customer se ejecuta para cada fila la función last_updated() que afecta a la columna last_update. Esto hace que todas las filas tengan en la columna *last_update* la fecha y hora en la que se hizo por última vez una operación UPDATE sobre la tabla.

El código de la función en el volcado de la base de datos a un **.sql* :

```
217 --
218 -- Name: last_updated(); Type: FUNCTION; Schema: public; Owner: postgres
219 --
220
221 CREATE FUNCTION public.last_updated() RETURNS trigger
222     LANGUAGE plpgsql
223     AS $$
224 BEGIN
225     NEW.last_update = CURRENT_TIMESTAMP;
226     RETURN NEW;
227 END $$;
228
229
230 ALTER FUNCTION public.last_updated() OWNER TO postgres;
```

Identifique alguna tabla donde se utilice una solución similar.

La función anterior se encuentra también en las tablas:

staff, *inventory*, *film*, *film_category*, *film_actor*, *actor*, *language*, *rental*, *customer*, *store*, *address*, *city*, *country*. *payment* no tiene ningún trigger.

En la tabla film se encuentra:

film_fulltext_trigger BEFORE INSERT OR UPDATE ON film FOR EACH ROW EXECUTE FUNCTION tsvector_update_trigger('fulltext', 'pg_catalog.english', 'title', 'description')

Antes de insertar o actualizar en la tabla film, ejecuta la función tsvector_update_trigger(). El código que se usó para crear el disparador se ve a continuación.

```
--
-- Name: film film_fulltext_trigger; Type: TRIGGER; Schema: public; Owner: postgres
--

CREATE TRIGGER film_fulltext_trigger BEFORE INSERT OR UPDATE ON public.film FOR EACH ROW EXECUTE FUNCTION tsvector_update_trigger('fulltext', 'pg_catalog.english', 'title', 'description');
```

Su uso parece ser que es el de actualizar el campo *fulltext* que es de tipo *tsvector* y además otros campos relacionados con el mismo (en este caso son *title* y *description*, *pg_catalog.english* no es un campo sino un catálogo para ayudar a hacer búsquedas de texto en inglés).

tsvector es un tipo de campo de texto de gran longitud que ayuda a la hora de buscar texto dentro de él (facilidad con los lexemas, etc).

8) Construya un disparador que guarde en una nueva tabla creada por usted la fecha de cuando se insertó un nuevo registro en la tabla film.

Primero se crea la nueva tabla:
CREATE TABLE fecha_registrto_film (
fechas DATE);

Después la función:
CREATE FUNCTION public.new_register() returns trigger language plpgsql as $$
begin
insert into fecha_registrto_film (fechas) values(current_date);
END;
$$
;

Finalmente se aplica el trigger :

CREATE TRIGGER register before INSERT ON film
FOR EACH ROW EXECUTE FUNCTION new_register();

```
practica5=# CREATE FUNCTION public.new_register() returns trigger language plpgsql  as $$
practica5$# begin
practica5$# insert into fecha_registrto_film (fechas) values(current_date);
practica5$# END;
practica5$# $$
practica5-# ;
CREATE FUNCTION
practica5=# CREATE TRIGGER register before INSERT ON film
practica5-# FOR EACH ROW EXECUTE FUNCTION new_register();
CREATE TRIGGER
```

Se comprueba que ha sido añadida como triggers a la tabla film.

```
Triggers:
    film_fulltext_trigger BEFORE INSERT OR UPDATE ON film FOR EACH ROW EXECUTE FUNCTION tsvector_update_trigger('fulltext', 'pg_cata
og.english', 'title', 'description')
    last_updated BEFORE UPDATE ON film FOR EACH ROW EXECUTE FUNCTION last_updated()
    register BEFORE INSERT ON film FOR EACH ROW EXECUTE FUNCTION new_register()
```

9) Comente el significado y la relevancia de las sequence.

Podemos ver que la palabra clave SEQUENCE se está usando en el script del fichero **.sql* volcado que generamos sobre la base de datos.

```
--
-- Name: address_address_id_seq; Type: SEQUENCE; Schema: public; Owner: postgres
--

CREATE SEQUENCE public.address_address_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;


ALTER TABLE public.address_address_id_seq OWNER TO postgres;

--
-- Name: address; Type: TABLE; Schema: public; Owner: postgres
--

CREATE TABLE public.address (
    address_id integer DEFAULT nextval('public.address_address_id_seq'::regclass) NOT NULL,
    address character varying(50) NOT NULL,
    address2 character varying(50),
    district character varying(20) NOT NULL,
    city_id smallint NOT NULL,
    postal_code character varying(10),
    phone character varying(20) NOT NULL,
    last_update timestamp without time zone DEFAULT now() NOT NULL
);
```

Se suele usar cuando queremos personalizar un campo de tipo entero en los que los valores se van a insertar automáticamente y autoincrementalmente, como son los id de una tabla. A diferencia de aplicar un SERIAL para hacer un campo autoincremental, con SEQUENCE podemos indicar desde qué número empezar, hasta que valor hacerlo, cuanto se incrementa el entero con cada nueva fila insertada, etc.

Mencionar también que no se puede aplicar a un campo directamente, sino que se debe guardar como una “variable” y después aplicarlo al campo que queremos de la tabla.

En la imagen anterior se crea una secuencia que empieza en 1 y se autoincrementa en 1 con cada nueva fila insertada en la tabla. Se le asocia después al usuario postgres con ALTER TABLE y finalmente, al crear la tabla, se asigna dicha SEQUENCE al campo *address_id* de la tabla *address*.

NOTA : Como están configuradas las SEQUENCE en el código actúan prácticamente igual que asignar la propiedad SERIAL al campo de la tabla (empieza el *id* de la primera fila en 1, y con cada inserción el valor insertado es el *id* de la fila anterior más 1

3. Bibliografía

Mostrar la estructura de la tabla, sus restricciones, … :

https://www.postgresqltutorial.com/postgresql-administration/postgresql-describe-table/

Añadir restricción de tamaño a un campo de texto :

https://stackoverflow.com/questions/29560016/how-to-add-a-length-constraint-to-a-text-field

Actualización de campos tipo *tsvector* :

https://www.postgresql.org/docs/current/textsearch-features.html

Tipo de campo tsvector :

https://www.postgresql.org/docs/15/datatype-textsearch.html

Palabra clave SEQUENCE

https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-sequences/

https://www.postgresql.org/docs/current/sql-createsequence.html