Universidad Autónoma de Madrid

Escuela Politécnica Superior

Sistemas Informáticos I

Práctica 2

Iker González Sánchez

Gonzalo Segovia Oblanca

Grupo 1332 - Pareja 8

Noviembre de 2023

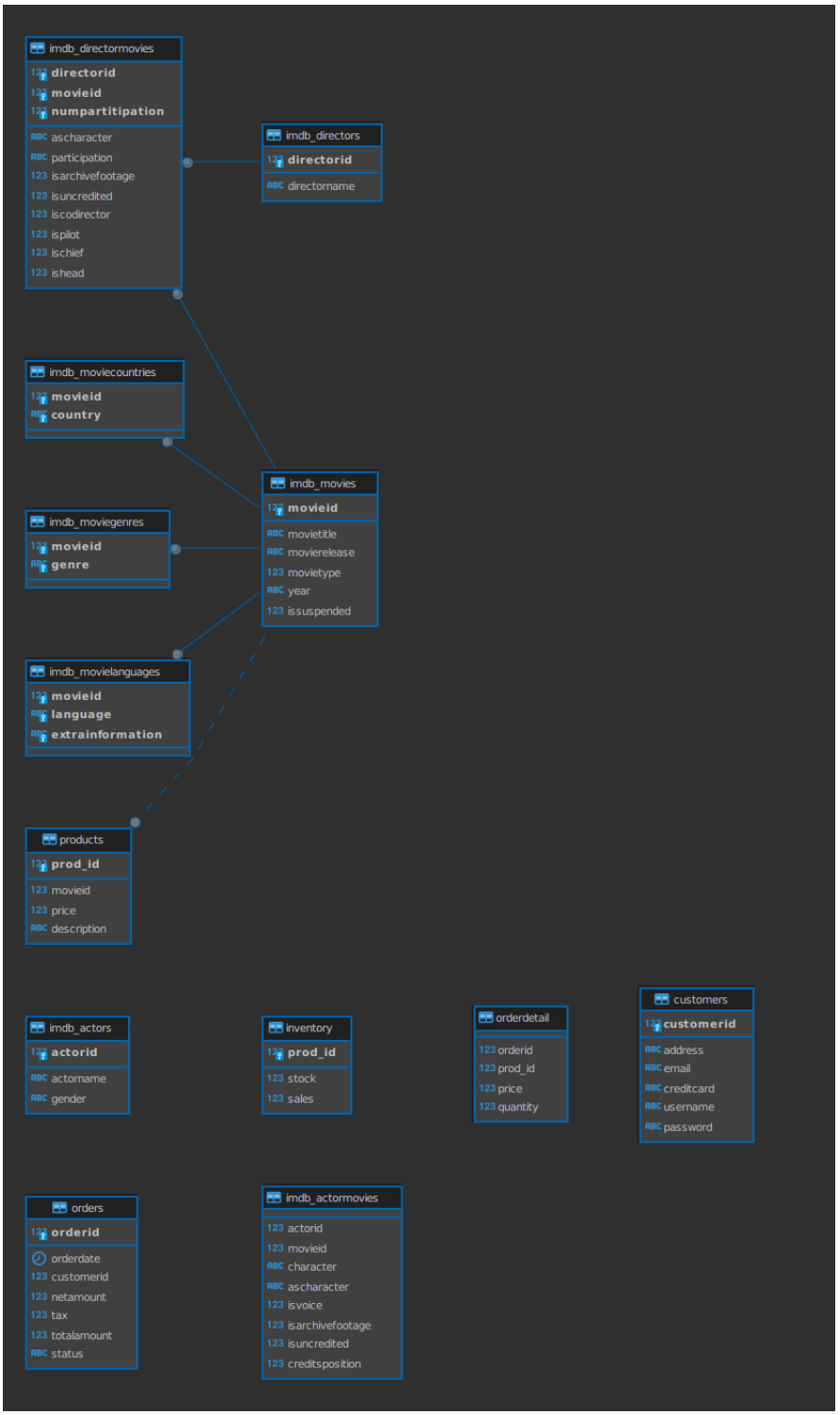
Escuela Politécnica Superior

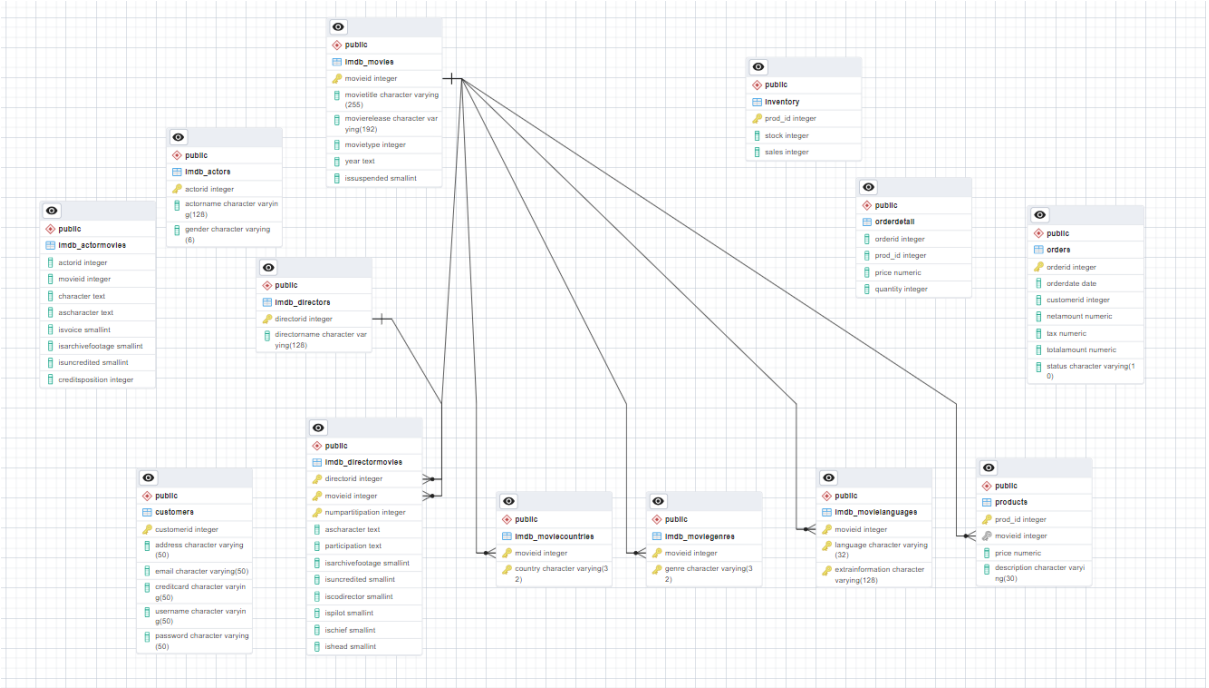
UAM
Universidad Autónoma de Madrid

Diseño de la BD

a) En primer lugar, se deberá efectuar un proceso de ingeniería inversa sobre la base de datos suministrada. En particular, se deberá:

- Obtener el Diagrama Entidad-Relación correspondiente. Se puede realizar manualmente o utilizando las herramientas que sean necesarias.





- Completar aquellos aspectos que se consideren necesarios, tales como restricciones, claves extranjeras, cambios en cascada, etc. Todos estos cambios se incorporarán en un único script, **`actualiza.sql`** convenientemente documentado.

Tras un análisis de la base de datos hemos observado que faltaban algunas claves foráneas:

```sql
-- Agregar una clave foránea a la tabla 'orders' que referencia la tabla 'customers' y configurarla para actualizar en cascada
ALTER TABLE orders
ADD CONSTRAINT fk_customer FOREIGN KEY (customerid) REFERENCES customers(customerid) ON DELETE CASCADE;
```

```sql
-- Agregar claves foráneas en la tabla actormovies
ALTER TABLE imdb_actormovies
ADD CONSTRAINT fk_actormovies_actors
FOREIGN KEY (actorid) REFERENCES imdb_actors(actorid);

ALTER TABLE imdb_actormovies
ADD CONSTRAINT fk_actormovies_movies
FOREIGN KEY (movieid) REFERENCES imdb_movies(movieid);

-- Agregar claves foráneas en la tabla orderdetails
ALTER TABLE orderdetail
ADD CONSTRAINT fk_orderdetail_orders
FOREIGN KEY (orderid) REFERENCES orders(orderid);

-- Agregar claves foráneas en la tabla inventory
ALTER TABLE inventory
ADD CONSTRAINT fk_inventory_products
FOREIGN KEY (prod_id) REFERENCES products(prod_id);
```

Conforme al avance de la práctica se han ido añadiendo más cambios a actualiza.sql, estos cambios se irán exponiendo en su respectivo apartado, conforme nos hemos percatado de

su necesidad.

- Añadir también a **`actualiza.sql`**:
  - Los triggers necesarios.

Los triggers se expondrán en los apartados por los cuales hemos decidido crearlos.

  - Un campo *`balance`* en la tabla *`customers`*, para guardar el saldo de los clientes.

Para ello lo hemos definido como NUMERIC(10,2) que significa que balance puede tener 10 dígitos en total de los cuales 2, son decimales; hemos creído muy apropiado este tipo para un campo que guarda valores monetarios.

```
-- Agregar un campo 'balance' en la tabla 'customers'
ALTER TABLE customers
ADD balance NUMERIC(10, 2),
```

  - Una nueva tabla *`ratings`* para guardar las valoraciones que ha dado cada usuario a cada película, de forma que se evite que un mismo usuario pueda valorar dos veces la misma película.

Para crear la tabla ratings, definimos su clave primaria (su id en este caso), además creamos claves foráneas y campos para el id del usuario y de las películas, y ponemos una restricción para que las valoraciones solo puedan ir de 1 a 5.
Por último para cumplir la restricción de que un mismo usuario pueda valorar 2 veces la misma película utilizamos UNIQUE relacionando el user_id y el movie_id

```
-- Crear una nueva tabla 'ratings' para guardar las valoraciones
CREATE TABLE ratings (
  rating_id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES customers(customerid),
  movie_id INTEGER REFERENCES imdb_movies(movieid),
  rating INTEGER CHECK (rating >= 1 AND rating <= 5),
  UNIQUE (user_id, movie_id) -- Evita que un usuario valore la misma película dos veces
);
```

  - Añadir dos campos a la tabla `imdb_movies`, para contener la valoración media `ratingmean` y el número de valoraciones `ratingcount`, de cada película.

Para ello hemos utilizado el tipo NUMERIC(3,2) para la media, ya que van de 1 a 5 no es necesario utilizar más dígitos; y para la cuenta de valoraciones un INTEGER.

```
-- Agregar dos campos a la tabla 'imdb_movies' para contener la valoración media y el número de valoraciones
ALTER TABLE imdb_movies
ADD ratingmean NUMERIC(3, 2),
ADD ratingcount INTEGER;
```

  - Aumentar el tamaño del campo `password` en la tabla `customers` para poder almacenar las contraseñas con el formato de 96 caracteres hexadecimales (para un futuro cifrado).

Mediante un ALTER COLUMN cambiamos el tipo de datos para poner el formato pedido.

```sql
-- Agregar un campo 'balance' en la tabla 'customers'
-- Aumentar el tamaño del campo 'password' en la tabla 'customers'
ALTER TABLE customers
ADD balance NUMERIC(10, 2),
ALTER COLUMN password TYPE VARCHAR(96);
```

- Crear un procedimiento que inicialice el campo `balance` de la tabla `customers` a un número aleatorio entre `0` y `N`, con signatura:

```
function setCustomersBalance(IN initialBalance bigint);
```

Siguiendo la signatura dada hemos creado la siguiente función:

```sql
-- Crear un procedimiento para inicializar el campo 'balance' de 'customers' con un valor aleatorio
CREATE OR REPLACE FUNCTION setCustomersBalance(IN initialBalance bigint)
RETURNS VOID AS $$
DECLARE
    randomBalance bigint;
BEGIN
    -- Generar un número aleatorio entre 0 y N (en este caso, N = 200)
    randomBalance := floor(random() * (initialBalance + 1));

    -- Actualizar el campo 'balance' para todos los registros en la tabla 'customers' con el valor aleatorio
    UPDATE customers
    SET balance = randomBalance;
END;
$$ LANGUAGE plpgsql;
```

Dado que el dato dado es de tipo bigint y no puede tener decimales, hemos decidido utilizar la función floor() que redondea hacía abajo, para quitar los decimales en esta inicialización.

- Añadir a **`actualiza.sql`** una llamada a dicho procedimiento, con `N = 200`.

Simplemente llamamos a la función:

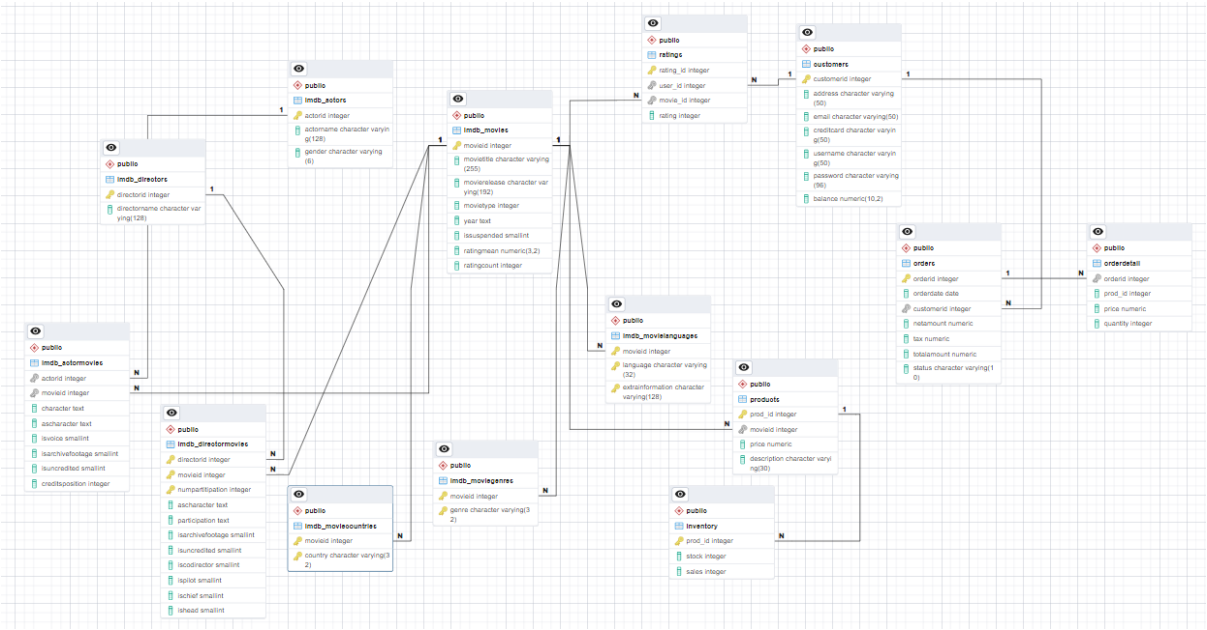
```sql
-- Llamar al procedimiento para inicializar el campo 'balance' de 'customers' con un número aleatorio entre 0 y 200
SELECT setCustomersBalance(200);
```

Además comprobamos que ha funcionado:

	customerid [PK] integer	address character varying (50)	email character varying (50)	creditcard character varying (50)	username character varying (50)	password character varying (96)	balance numeric (10,2)
1	1	plaid throat 139	hawley.bra@jmail.com	4832398859973558	shad	naples	122.00
2	2	bethel scotty 16	claim.portal@jmail.com	4517014291483479	laxity	cesar	122.00
3	3	sylvia gnash 218	trait.ritual@potmail.com	4649262782739587	flax	goat	122.00
4	4	chivas infect 83	clair.rodder@mamoot.com	4834876664936499	share	honor	122.00
5	5	dave rummy 49	skycap.unbred@jmail.com	4369574201483324	dickys	lank	122.00
6	6	vlsi wakeup 281	primal.savior@jmail.com	4963528832263421	gain	ariel	122.00
7	7	sheikh clear 299	havana.opine@potmail.co...	4035822167587742	sender	zodiac	122.00

Se deberá entregar, al menos, el **Diagrama Entidad-Relación final**, tras los cambios del script `actualiza.sql`. A la hora de elaborar este diagrama conviene revisar:

- claves primarias
- claves externas o extranjeras
- qué tablas son entidades, cuáles relaciones y cuáles atributos
- cardinalidad
- secuencias de campos numéricos
- entidades débiles
- atributos multivaluados
- atributos derivados
- participación total



b) Sabiendo que los precios de las películas se han ido incrementando un 2% anualmente, elaborar la consulta **`setPrice.sql`** que complete la columna `price` de la tabla `orderdetail`, sabiendo que el precio actual es el de la tabla `products`.

Para realizar esta consulta se han utilizado diferentes funciones:

- EXTRACT(YEAR FROM CURRENT_DATE): Obtiene el año actual de la fecha actual (CURRENT_DATE).
- EXTRACT(YEAR FROM o.orderdate): Obtiene el año correspondiente de la tabla orders.
- POWER(1.02, ...): Eleva 1.02 a la potencia de la diferencia en años calculada.

```
-- Actualizar la columna 'price' en la tabla 'orderdetail' con el precio actual de 'products' considerando un incremento del 2% anual
UPDATE orderdetail AS od
SET price = p.price * POWER(1.02, EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM o.orderdate))
FROM products AS p, orders AS o
WHERE od.prod_id = p.prod_id AND od.orderid = o.orderid;
```

Dado que tardaba un tiempo considerable, decimos añadir índices:

```
CREATE INDEX idx_orderdetail_prod_id ON orderdetail (orderid, prod_id);
CREATE INDEX idx_products_prod_id ON products (prod_id);
```

c) Una vez se disponga de esta información, realizar un **procedimiento almacenado**, **setOrderAmount**, que complete las columnas `netamount` (suma de los precios de las películas del pedido) y `totalamount` (`netamount` más impuestos) de la tabla `orders` cuando éstas no contengan ningún valor. Invocar de forma manual la consulta de modificación `setPrice` y este procedimiento almacenado que se pide, para realizar una carga inicial. El procedimiento almacenado no lleva parámetros.

En esta consulta utilizamos la función SUM() para sumar todos los precios de orderdetail donde su order id coincida con la de la taba order.

```sql
-- Crear el procedimiento almacenado setOrderAmount
CREATE OR REPLACE FUNCTION setOrderAmount()
RETURNS VOID AS $$
BEGIN
    -- Actualizar las columnas netamount y totalamount en la tabla orders cuando estén en blanco
    UPDATE orders AS o
    SET netamount = (
        SELECT SUM(od.price)
        FROM orderdetail AS od
        WHERE od.orderid = o.orderid
    )
    WHERE netamount IS NULL;

    UPDATE orders as o
    SET totalamount = (
        o.netamount *(1 + (o.tax / 100))
    )
    WHERE totalamount IS NULL;

END;
$$ LANGUAGE plpgsql;


-- Llamar al procedimiento almacenado para realizar la carga inicial en la tabla 'orders'
SELECT setOrderAmount();

--DROP FUNCTION setOrderAmount;
```

Debido al gran número de datos y veces que tiene que recorrer las tablas esta función, en primera instancia tardaba horas en acabar. Por lo tanto decidimos crear índices para acelerar el proceso; por lo que añadimos a actualiza.sql:

```sql
-- Agregar una clave foránea a la tabla 'orders' que referencia la tabla 'customers' y configurarla para actualizar en cascada
ALTER TABLE orders
ADD CONSTRAINT fk_customer FOREIGN KEY (customerid) REFERENCES customers(customerid) ON DELETE CASCADE;
CREATE INDEX idx_orders_orderid ON orders(orderid);
CREATE INDEX idx_orderdetail_orderid ON orderdetail(orderid);
```

Lo que aceleró enormemente el proceso, tardando solo algunos segundos en completarse.

d) Realizar una función, **getTopSales**, que reciba como argumentos dos años diferentes y devuelva las películas que más se han vendido entre esos dos años, una por año, ordenadas de mayor a menor por número de ventas.

Year	Film	Sales
1897	El Gran Dictador	12
1912	Superman	4
...		

(Sólo es un ejemplo, no son resultados reales)

La signatura de la función es:

```
function getTopSales(year1 INT, year2 INT,
                     OUT Year INT, OUT Film CHAR, OUT sales bigint);
```

Mediante un bucle LOOP que recorre el intervalo entre los dos años dados contamos la cantidad de ventas y nos quedamos con la película que más haya vendido de cada año.

```sql
-- Crear la función getTopSales con la firma requerida
CREATE OR REPLACE FUNCTION getTopSales(year1 INT, year2 INT, OUT Year_given INT, OUT Film CHAR, OUT sales bigint)
RETURNS SETOF record AS $$
DECLARE
    year_value INT;
BEGIN
    -- Consulta para obtener las películas más vendidas por año entre los dos años dados
    FOR year_value IN year1..year2 LOOP
        SELECT INTO Film, sales
            m.movietitle, SUM(od.quantity)
            FROM orders o
            JOIN orderdetail od ON o.orderid = od.orderid
            JOIN imdb_movies m ON od.prod_id  = m.movieid
            WHERE EXTRACT(YEAR FROM o.orderdate) = year_value
            GROUP BY m.movietitle
            ORDER BY SUM(od.quantity) DESC
            LIMIT 1;

        Year_given := year_value;
        RETURN NEXT;
    END LOOP;
END;
$$ LANGUAGE plpgsql;


-- Llamar a la función getTopSales con los años deseados (por ejemplo, de 2020 a 2022)
SELECT * FROM getTopSales(2018, 2021);

--DROP FUNCTION IF EXISTS getTopSales;
```

Resultado:

	year_given integer	film character	sales bigint
1	2018	54 (1998)	63
2	2019	8 Heads in a Duffel Bag (1997)	44
3	2020	20 Dates (1998)	57
4	2021	About Adam (2000)	55

e) Realizar una función, **`getTopActors`**, que reciba un parámetro el género y devuelva los actores o actrices que más veces han actuado en dicho género, ordenados de más actuaciones a menos, **siempre que hayan trabajado en más de 4 películas de ese género**, con información de la película en la que debutaron para ese género, el año de esa película y quién (o quiénes) dirigió esa película (pueden aparecer varios registros para el mismo actor porque hizo varias películas su primer año, varios directores para la misma película, etc.)

Actor	Num	Debut	Film	Director
Costner, Kevin	6	1985	American Flyers (1985)	Badham, John
Gibbs, Keith	5	1994	Air Up There, The (1994)	Glaser, Paul Michael
Gibbs, Keith	5	1994	Blue Chips (1994)	Friedkin, William
...				

(Sólo es un ejemplo, no son resultados reales)

La signatura de la función es:

```
function getTopActors(genre CHAR, OUT Actor char, OUT Num INT, OUT Debut INT,
                      OUT Film CHAR, OUT Director CHAR);
```

Concatenando consultas filtramos los actores y después obtenemos el resto.

```
CREATE OR REPLACE FUNCTION getTopActors(Genre_given CHAR, OUT Actor VARCHAR, OUT Num INT, OUT Debut INT, OUT Film VARCHAR, OUT Director VARCHAR)
RETURNS SETOF RECORD AS $$
DECLARE
    rec RECORD;
    cur CURSOR FOR
        WITH gen_actor_stats AS (
            SELECT
                am.actorid,
                a.actorname,
                COUNT(*) AS num_movies,
                MIN(m.year) AS debut_year
            FROM imdb_actormovies am
            JOIN imdb_movies m ON am.movieid = m.movieid
            JOIN imdb_actors a ON am.actorid = a.actorid
            JOIN imdb_moviegenres mg ON m.movieid = mg.movieid
            WHERE mg.genre = Genre_given
            GROUP BY am.actorid, a.actorname
        ),
        filtered_actors AS (
            SELECT
                ga.actorid,
                ga.actorname,
                ga.num_movies,
                ga.debut_year,
                STRING_AGG(CASE WHEN m.year = ga.debut_year THEN m.movietitle END, ', ') AS movie_titles,
                STRING_AGG(CASE WHEN m.year = ga.debut_year THEN d.directorname END, ', ') AS director_names
            FROM gen_actor_stats ga
            JOIN imdb_actormovies am ON ga.actorid = am.actorid
            JOIN imdb_movies m ON am.movieid = m.movieid
            JOIN imdb_directormovies md ON m.movieid = md.movieid
            JOIN imdb_directors d ON md.directorid = d.directorid
            WHERE ga.num_movies > 4
            GROUP BY ga.actorid, ga.actorname, ga.num_movies, ga.debut_year
        )
        SELECT * FROM filtered_actors
        ORDER BY num_movies DESC;
```

```
BEGIN
    OPEN cur;
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND;

        Actor := rec.actorname;
        Num := rec.num_movies;
        Debut := rec.debut_year;
        Film := rec.movie_titles;
        Director := rec.director_names;

        RETURN NEXT;
    END LOOP;
    CLOSE cur;
END;
$$ LANGUAGE plpgsql;


SELECT * FROM getTopActors('Comedy');

--DROP FUNCTION IF EXISTS getTopActors(character);
```

Resultado:

	actor character varying	num integer	debut integer	film character varying	director character varying
1	Welker, Frank	34	1969	Computer Wore Tennis Shoes, The (1969)	Butler, Robert (I)
2	Goldberg, Whoopi	26	1986	Jumpin' Jack Flash (1986)	Marshall, Penny (I)
3	Aykroyd, Dan	23	1980	Blues Brothers, The (1980)	Landis, John (I)
4	Cusack, Joan	22	1980	My Bodyguard (1980)	Bill, Tony
5	Williams, Robin (I)	20	1980	Popeye (1980)	Altman, Robert (I)
6	Flowers, Bess	19	1934	It Happened One Night (1934)	Capra, Frank
7	Kane, Carol (I)	19	1971	Carnal Knowledge (1971)	Nichols, Mike (I)
8	DeVito, Danny	19	1971	Bananas (1971)	Allen, Woody
9	Garofalo, Janeane	19	1994	Reality Bites (1994)	Stiller, Ben
10	McKean, Michael	19	1982	Young Doctors in Love (1982)	Marshall, Garry

f) Crear las tablas correspondientes y convertir los atributos multivaluados `moviecountries`, `moviegenres` y `movielanguages` en relaciones entre la tabla `movies` y las tablas creadas. Estos cambios también se incorporarán al script **`actualiza.sql`**.

Ya están implementadas correctamente en la base de datos original por lo que no ha sido necesario hacer ningún cambio.

g) Realizar un trigger, **`updOrders`**, que actualice la información de la tabla `orders` cuando se añada, actualice o elimine un artículo del carrito.

Para este trigger creamos la función con las 3 posibilidades, es decir, actualizar, añadir y borrar y después el trigger que la llama para cada columna.

```
--Función para cambios en el pedido
CREATE OR REPLACE FUNCTION updOrdersFunction()
RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'UPDATE') THEN
      UPDATE orders
      SET netamount = netamount + (NEW.price * (NEW.quantity - OLD.quantity)), totalamount = netamount *(1 + (tax / 100))
      WHERE orderid = OLD.orderid;
      RETURN NEW;
  ELSIF (TG_OP = 'INSERT') THEN
      UPDATE orders
      SET netamount = netamount + (NEW.price * NEW.quantity), totalamount = netamount *(1 + (tax / 100))
      WHERE orderid = OLD.orderid;
      RETURN NEW;
  ELSIF (TG_OP = 'DELETE') THEN
      UPDATE orders
      SET netamount = netamount - (OLD.price * OLD.quantity), totalamount = netamount *(1 + (tax / 100))
      WHERE orderid = OLD.orderid;
      RETURN OLD;
  END IF;
END;
$$ LANGUAGE plpgsql;

--Trigger para cambios en el pedido
CREATE OR REPLACE TRIGGER updOrders
AFTER INSERT OR DELETE OR UPDATE ON orderdetail
FOR EACH ROW
EXECUTE FUNCTION updOrdersFunction();
```

Se han aplicado algunas pruebas como borrar por ejemplo:

```
SELECT * FROM orderdetail od
WHERE od.orderid = 3

SELECT orderid, SUM(price) FROM orderdetail od
WHERE od.orderid = 3
GROUP BY orderid

DELETE FROM orderdetail od
WHERE od.prod_id=1467 AND od.orderid = 3
```

h) Realizar un trigger, **updRatings**, que actualice la información de la tabla `imdb_movies` cuando se añada, actualice o elimine una valoración, modificando los campos `ratingmean` y `ratingcount`.

Mismo procedimiento que el anterior, para la media se ha utilizado la función AVG() y para contar las valoraciones COUNT().

```sql
--Función para cambios en la valoración
CREATE OR REPLACE FUNCTION updRatingsFunction()
RETURNS TRIGGER AS $$
DECLARE
    mean_value numeric(3,2);
    count_value integer;
BEGIN
    IF (TG_OP = 'UPDATE' OR TG_OP = 'INSERT') THEN
        SELECT COUNT(rating_id) INTO count_value FROM ratings
        WHERE movie_id = NEW.movie_id;
        SELECT AVG(rating) INTO mean_value FROM ratings
        WHERE movie_id = NEW.movie_id;

        UPDATE imdb_movies
        SET ratingmean = mean_value, ratingcount = count_value
        WHERE movieid = NEW.movie_id;
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        SELECT COUNT(rating_id) INTO count_value FROM ratings
        WHERE movie_id = OLD.movie_id;
        SELECT AVG(rating) INTO mean_value FROM ratings
        WHERE movie_id = OLD.movie_id;

        UPDATE imdb_movies
        SET ratingmean = mean_value, ratingcount = count_value
        WHERE movieid = OLD.movie_id;
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE plpgsql;

--Trigger para cambios en la valoración
CREATE OR REPLACE TRIGGER updRatings
AFTER INSERT OR DELETE OR UPDATE ON ratings
FOR EACH ROW
EXECUTE FUNCTION updRatingsFunction();
```

Para probar este trigger se ha creado una función que permita al cliente valorar una película:

```sql
-- Crear un procedimiento para añadir una valoración a una película
CREATE OR REPLACE FUNCTION addRating(IN r_rating integer, IN r_movie_id integer, IN r_user_id integer)
RETURNS VOID AS $$
BEGIN
    INSERT INTO ratings (rating, movie_id, user_id) VALUES (r_rating, r_movie_id, r_user_id);
END;
$$ LANGUAGE plpgsql;
```

Además de utilizar esta función para las pruebas se han hecho otras, como borrar por ejemplo:

```sql
SELECT addRating(3, 103, 4);
DELETE FROM ratings WHERE movie_id = 103 AND user_id = 4;
```

i) Crear un trigger, **updInventoryAndCustomer**, que realice las siguientes tareas cuando un pedido pase a estado `Paid`:

- actualice la tabla `inventory`.
- descuente en la tabla `customers` el precio total de la compra.

Para esta función comprobamos cuando el pedido cambia su estado a “Paid”, además de reducir el stock del producto en inventory y bajar el dinero del cliente:

```sql
--Función para pagos de pedidos
CREATE OR REPLACE FUNCTION updInventoryAndCustomerFunction()
RETURNS TRIGGER AS $$
DECLARE
    total_price_paid numeric(10, 2);
BEGIN
    -- Verificar si el pedido cambió al estado "Paid"
    IF NEW.status = 'Paid' AND OLD.status <> 'Paid' THEN
        -- Actualizar la tabla inventory
        UPDATE inventory
        SET stock = stock - (SELECT SUM(quantity) FROM orderdetail WHERE orderid = NEW.orderid)
        WHERE prod_id IN (SELECT prod_id FROM orderdetail WHERE orderid = NEW.orderid);


        -- Descuentar el precio total en la tabla customers
        SELECT totalamount INTO total_price_paid FROM orders
        WHERE orderid = NEW.orderid;

        UPDATE customers
        SET balance = balance - total_price_paid
        WHERE customerid = NEW.customerid;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger para pagos de pedidos
CREATE OR REPLACE TRIGGER updInventoryAndCustomer
AFTER UPDATE ON orders
FOR EACH ROW
EXECUTE FUNCTION updInventoryAndCustomerFunction();
```

Se realizaron múltiples pruebas para comprobar su correcto funcionamiento, como por ejemplo:

```sql
-- Ver el estado antes de la actualización
SELECT * FROM inventory WHERE prod_id=1;
SELECT * FROM customers WHERE customerid = 2132;
UPDATE customers SET balance = 300 WHERE customerid = 2132;

SELECT *
FROM orders
WHERE orderid IN (SELECT orderid FROM orderdetail WHERE prod_id = 1);

SELECT *
FROM orders o
JOIN orderdetail od ON o.orderid = od.orderid
JOIN inventory i ON od.prod_id = i.prod_id
WHERE o.customerid = 2132 AND od.prod_id = 1;

-- Actualizar el estado del pedido a 'Paid' para probar el trigger
UPDATE orders SET status = 'Paid' WHERE orderid = 27872;
```

Integración con Python

j) Utiliza SQLAlchemy (https://www.sqlalchemy.org/) para mostrar la tabla resultante en el apartado d) anterior, para los dos últimos años, y limitado a 10 filas. Crea para ello un fichero Python (**`mostrarTabla.py`**) cuya ejecución muestre el contenido de la tabla.

Para realizar este archivo lo primero es usar create_engine al que le pasamos una cadena de texto en la que especificamos todos los datos de la base de datos como el usuario, la contraseña, el host, el puerto y el nombre de la base de datos.
Después nos conectamos y mediante execute(), text() y una cadena de texto con nuestra orden obtenemos los resultados, que mostramos con un bucle.
Por último cerramos tanto la conexión como la máquina.

```python
from sqlalchemy import create_engine, text, func


def main():
    # engine is an instance of AsyncEngine
    engine = create_engine(
        "postgresql://alumnodb:password@localhost:5432/si1",
        echo=True,
    )

    conection = engine.connect()

    result = conection.execute(text("SELECT getTopSales(2018, 2021) AS resultado LIMIT 10;"))
    resultados = result.fetchall()

    # Imprime todos los resultados
    for resultado in resultados:
        print(resultado)

    conection.close()
    engine.dispose()


if __name__ == "__main__":
    main()
```

```
e462135@LAPTOP-995AQRBR:~/SI1_PR/p2/python_files$ python3 mostrarTabla.py
2023-11-17 12:52:35,892 INFO sqlalchemy.engine.Engine select pg_catalog.version()
2023-11-17 12:52:35,892 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-11-17 12:52:35,894 INFO sqlalchemy.engine.Engine select current_schema()
2023-11-17 12:52:35,895 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-11-17 12:52:35,901 INFO sqlalchemy.engine.Engine show standard_conforming_strings
2023-11-17 12:52:35,901 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-11-17 12:52:35,903 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2023-11-17 12:52:35,903 INFO sqlalchemy.engine.Engine SELECT getTopSales(2018, 2021) AS resultado;
2023-11-17 12:52:35,903 INFO sqlalchemy.engine.Engine [generated in 0.00016s] {}
('(2018,"54 (1998)",63)',)
('(2019,"8 Heads in a Duffel Bag (1997)",44)',)
('(2020,"20 Dates (1998)",57)',)
('(2021,"About Adam (2000)",55)',)
2023-11-17 12:52:36,423 INFO sqlalchemy.engine.Engine ROLLBACK
```

	year_given integer	film character	sales bigint
1	2018	54 (1998)	63
2	2019	8 Heads in a Duffel Bag (1997)	44
3	2020	20 Dates (1998)	57
4	2021	About Adam (2000)	55

Como se puede observar el resultado es el mismo.

Optimización

Para la parte de Optimización de esta práctica, se suministra una variante de la Base de Datos de la Práctica anterior (`dump_p2_v2.sql.zip`). En dicha Base de Datos se ha hecho uso de sentencias:

```
ALTER TABLE table_name SET (autovacuum_vacuum_threshold=100000000,
autovacuum_analyze_threshold=100000000);
```

para evitar la generación automática de estadísticas, y se ha realizado algo de limpieza sobre la BD de la práctica anterior, además de cargarse los importes de los pedidos.

k) Estudio del impacto de un índice:

a. Crear una consulta, **`estadosDistintos.sql`**, que muestre el número de estados (columna `state`) distintos con clientes que tienen pedidos en un año dado usando el formato YYYY (por ejemplo 2017) y que además pertenecen a un país (`country`) determinado, por ejemplo, `Peru`.

Tras realizar la query descrita en el enunciado:

```
SELECT COUNT(DISTINCT customers.state) AS num_states
FROM orders
JOIN customers ON orders.customerid = customers.customerid
WHERE EXTRACT(YEAR FROM orders.orderDate) = 2017
AND customers.country = 'Peru';
```

Obtenemos lo siguiente:

	123 num_states
1	185

b. Mediante la sentencia `EXPLAIN` estudiar el plan de ejecución de la consulta.

	ABC QUERY PLAN
1	Aggregate (cost=4821.98..4821.99 rows=1 width=8)
2	-> Gather (cost=1529.04..4821.97 rows=5 width=118)
3	Workers Planned: 1
4	-> Hash Join (cost=529.04..3821.47 rows=3 width=118)
5	Hash Cond: (orders.customerid = customers.customerid)
6	-> Parallel Seq Scan on orders (cost=0.00..3291.03 rows=535 width=4)
7	Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
8	-> Hash (cost=528.16..528.16 rows=70 width=122)
9	-> Seq Scan on customers (cost=0.00..528.16 rows=70 width=122)
10	Filter: ((country)::text = 'Peru'::text)

Podemos observar que comienza iterando orders y customers para emparejar los customerids, en paralelo con la iteración del escaneo de fechas en orders para filtrar aquellas que corresponden a *2017*. Finalmente, sobre el resultado de estos escaneos previos, filtra aquellas cuya columna country equivale a la cadena *'Peru'*.

c. Identificar un índice que mejore el rendimiento de la consulta y crearlo (si ya existía, borrarlo).

Basándonos en el estudio anterior de la consulta del apartado *a)*, concluimos que un índice compuesto en *orders.customerid* y *orders.orderDate*.

d. Estudiar el nuevo plan de ejecución y compararlo con el anterior.

```
1  Aggregate  (cost=2688.75..2688.76 rows=1 width=8)
2    ->  Nested Loop  (cost=0.42..2688.74 rows=5 width=118)
3          ->  Seq Scan on customers  (cost=0.00..528.16 rows=70 width=122)
4                Filter: ((country)::text = 'Peru'::text)
5          ->  Index Only Scan using idx_orders_customerid_orderdate on orders  (cost=0.42..30.82 rows=5 width=4)
6                Index Cond: (customerid = customers.customerid)
7                Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
```

e. Probad distintos índices y discutid los resultados.

Hemos probado los siguientes índices:

customerid y country (compuesto en customers)

- Con esta combinación de índices obtenemos un coste de ~4810, que supone una mejora despreciable en comparación a una ejecución sin ella. En concreto, este índice cambia el plan de ejecución de la query, filtrando antes el año del pedido que los clientes peruanos:

```
1  Aggregate  (cost=4810.83..4810.84 rows=1 width=8)
2    ->  Gather  (cost=1517.88..4810.81 rows=5 width=118)
3          Workers Planned: 1
4          ->  Hash Join  (cost=517.88..3810.31 rows=3 width=118)
5                Hash Cond: (orders.customerid = customers.customerid)
6                ->  Parallel Seq Scan on orders  (cost=0.00..3291.03 rows=535 width=4)
7                      Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
8                ->  Hash  (cost=517.01..517.01 rows=70 width=122)
9                      ->  Bitmap Heap Scan on customers  (cost=342.00..517.01 rows=70 width=122)
10                           Recheck Cond: ((country)::text = 'Peru'::text)
11                           ->  Bitmap Index Scan on idx_customers_customerid_country  (cost=0.00..341.98 rows=70 width=0)
12                                 Index Cond: ((country)::text = 'Peru'::text)
```

orders.customerid

No afecta al rendimiento.

customers.customerid

No afecta al rendimiento.

orders.orderdate

No afecta al rendimiento.

customers.country

Obtenemos el siguiente plan de ejecución con una ligera mejora frente al original:

```
1  Aggregate  (cost=4473.65..4473.66 rows=1 width=8)
2    ->  Gather  (cost=1180.71..4473.64 rows=5 width=118)
3          Workers Planned: 1
4          ->  Hash Join  (cost=180.71..3473.14 rows=3 width=118)
5                Hash Cond: (orders.customerid = customers.customerid)
6                ->  Parallel Seq Scan on orders  (cost=0.00..3291.03 rows=535 width=4)
7                      Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
8                ->  Hash  (cost=179.83..179.83 rows=70 width=122)
9                      ->  Bitmap Heap Scan on customers  (cost=4.83..179.83 rows=70 width=122)
10                           Recheck Cond: ((country)::text = 'Peru'::text)
11                           ->  Bitmap Index Scan on idx_customers_country  (cost=0.00..4.81 rows=70 width=0)
12                                 Index Cond: ((country)::text = 'Peru'::text)
```

Como vemos, el plan de ejecución se ordena de manera idéntica al compuesto por *customerid* y *country*.

customers.state

No afecta al rendimiento.

Conclusiones:
Creando un índice en *country* y otro compuesto en *orderdate* y *customerid* mejora ligeramente el rendimiento de nuestra observación inicial a expensas del rendimiento en las sentencias insert/delete/update. Por tanto, concluimos que la mejor opción para la implementación de la base de datos es crear sólo el índice compuesto en *orderdate* y *customerid*. Sin embargo, para la ejecución de nuestra query, la mejor opción es la combinación de los dos índices, como se puede apreciar en el plan de ejecución:

```
1  Aggregate  (cost=2340.42..2340.43 rows=1 width=8)
2    ->  Nested Loop  (cost=5.25..2340.41 rows=5 width=118)
3          ->  Bitmap Heap Scan on customers  (cost=4.83..179.83 rows=70 width=122)
4                Recheck Cond: ((country)::text = 'Peru'::text)
5                ->  Bitmap Index Scan on idx_customers_country  (cost=0.00..4.81 rows=70 width=0)
6                      Index Cond: ((country)::text = 'Peru'::text)
7          ->  Index Only Scan using idx_orders_customerid_orderdate on orders  (cost=0.42..30.82 rows=5 width=4)
8                Index Cond: (customerid = customers.customerid)
9                Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
```

l) Estudio del impacto de cambiar la forma de realizar una consulta:

a. Estudiar los planes de ejecución de las consultas alternativas mostradas en el Anexo 1 y compararlos.

i. ¿Qué consulta devuelve algún resultado nada más comenzar su ejecución?

La tercera consulta, ya que no está utilizando un subconjunto de datos específico de la tabla *orders*, a diferencia del resto.

ii. ¿Qué consulta se puede beneficiar de la ejecución en paralelo?

El rendimiento de la segunda consulta podría verse beneficiado ejecutando en paralelo las operaciones de *UNION ALL* y *GROUP BY*, ya que se trata de operaciones independientes.

m) Estudio del impacto de la generación de estadísticas:

b. Partir de la base de datos suministrada para este apartado (recién creada y cargada de datos).

Para obtener una BDD nueva, borramos el volumen haciendo uso de *Docker Desktop* y ejecutamos *$ docker compose down* y *$ docker compose up* con el yml proporcionado (cambiando el fichero de datos del primer apartado por el del siguiente).

c. Estudiar con la sentencia `EXPLAIN` el coste de ejecución de las dos consultas indicadas en el Anexo 2.

Para la primera consulta:

```
   QUERY PLAN
1  Aggregate  (cost=3507.17..3507.18 rows=1 width=8)
2    ->  Seq Scan on orders  (cost=0.00..3504.90 rows=909 width=0)
3          Filter: (status IS NULL)
```

Para la segunda:

```
   QUERY PLAN
1  Aggregate  (cost=3961.65..3961.66 rows=1 width=8)
2    ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=0)
3          Filter: ((status)::text = 'Shipped'::text)
```

Podemos observar que el plan de ejecución es muy similar y que sólo cambia el filtro. Asimismo, el coste de la segunda query es ligeramente mayor que el de la primera. Probablemente debido a que sea más costoso comparar un texto que NULL.

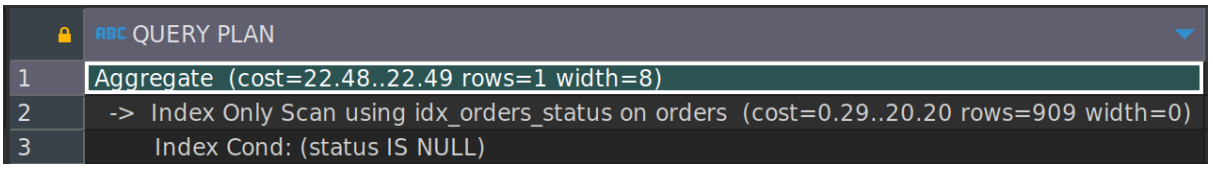
d. Crear un índice en la tabla *orders* por la columna *status*.

Lo creamos mediante la siguiente sentencia:

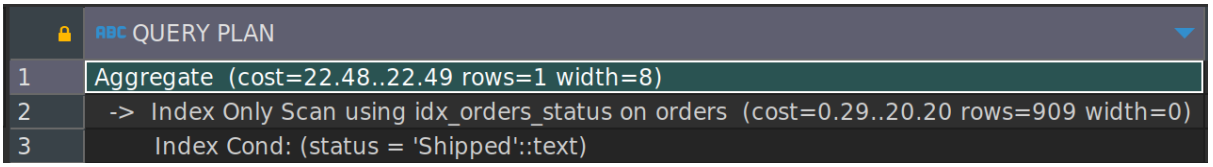
```
create index idx_orders_status on orders(status);
```

e. Estudiar de nuevo la planificación de las mismas consultas.

Para la primera consulta:



Para la segunda:



Podemos observar que ambos costes son ahora iguales y que se han reducido mucho (de ~4000 a ~20). El plan de ejecución sigue siendo el mismo, cambiando tan sólo la condición del índice.

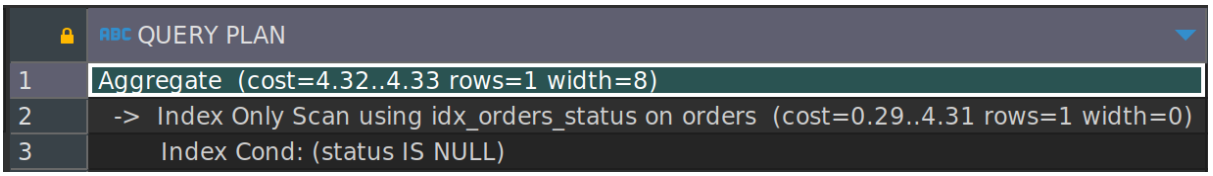
f. Ejecutar la sentencia `ANALYZE` para generar las estadísticas sobre la tabla *orders*.

Ejecutamos la siguiente sentencia:

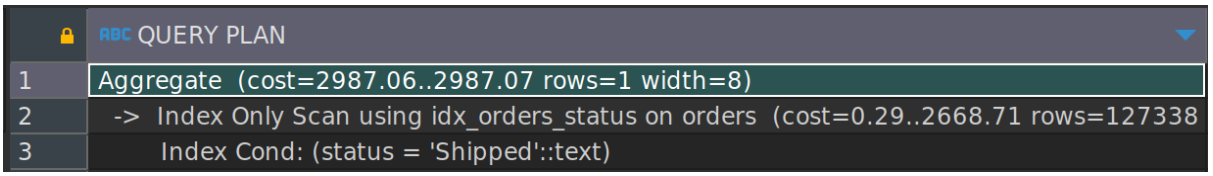
```
analyze orders;
```

g. Estudiar de nuevo el coste de las consultas y comparar con el coste anterior a la generación de estadísticas. Comparar el plan de ejecución y discutir el resultado.

Para la primera consulta:



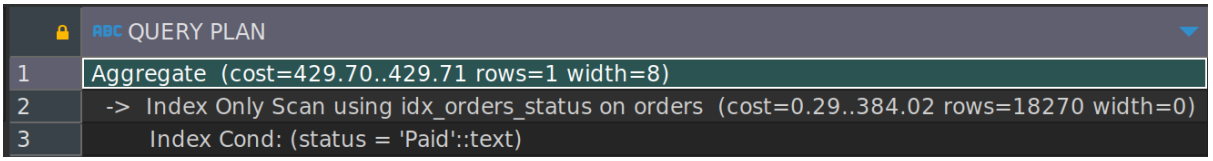
Para la segunda:



Como se puede ver, tras ejecutar analyze, la segunda query resulta ~750 veces más costosa que la primera. Para comprender esto hay que entender que la sentencia "*analyze orders;*" recopila estadísticas de la distribución de datos en la tabla orders. Uno de estas estadísticas es la frecuencia de valores distintos y, puesto que hay 0 valores *NULL* y 127.323 'Shipped', obtenemos costes tan dispares. Esto se puede ver en la segunda fila del plan de ejecución: explora 1 fila en la primera query y 127.338 en la segunda.

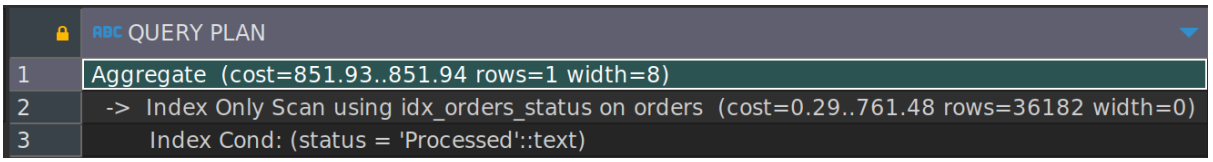
h. Comparar con la planificación de las otras dos consultas proporcionadas y comentar los resultados.

Para la tercera consulta:



	QUERY PLAN
1	Aggregate (cost=429.70..429.71 rows=1 width=8)
2	-> Index Only Scan using idx_orders_status on orders (cost=0.29..384.02 rows=18270 width=0)
3	Index Cond: (status = 'Paid'::text)

Para la cuarta:



	QUERY PLAN
1	Aggregate (cost=851.93..851.94 rows=1 width=8)
2	-> Index Only Scan using idx_orders_status on orders (cost=0.29..761.48 rows=36182 width=0)
3	Index Cond: (status = 'Processed'::text)

Como podemos observar, en el plan de ejecución de cada una solo cambia la condición del índice con respecto a las demás. De nuevo, se puede contemplar que tras analizar las estadísticas el coste de las querys queda enlazado a la frecuencia de los datos que se piden en las mismas.

i. Crear un *script* **`countStatus.sql`**, conteniendo las consultas, la creación de índices y las sentencias `ANALYZE`.

Disponible en los archivos entregados.