Práctica 2: Conceptos Avanzados sobre Bases de Datos Relacionales y Optimización de Consultas

Daniel Birsan & Jorge Paniagua Moreno

Diagrama Entidad-Relación antes de ejecutar “actualiza.sql”

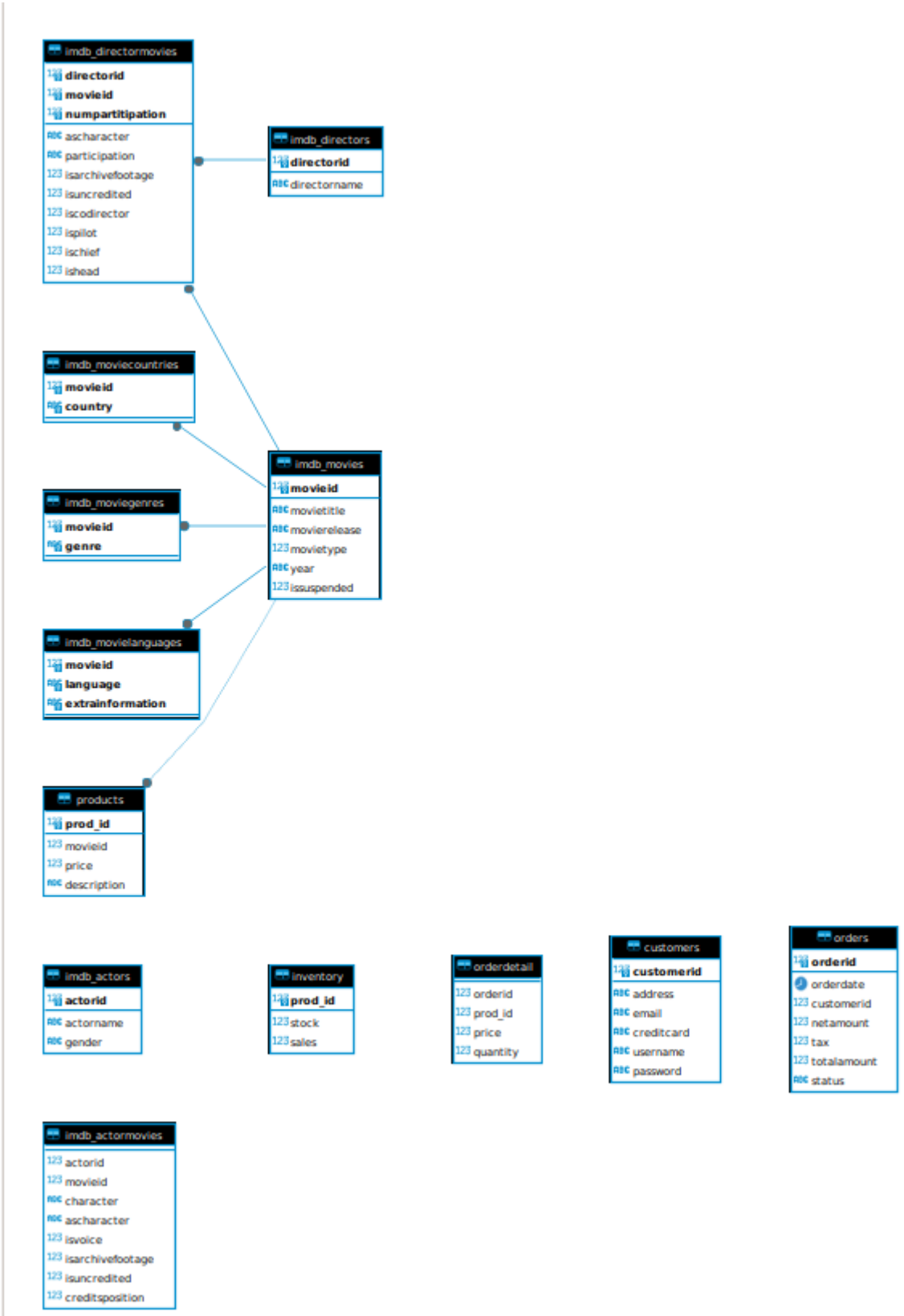
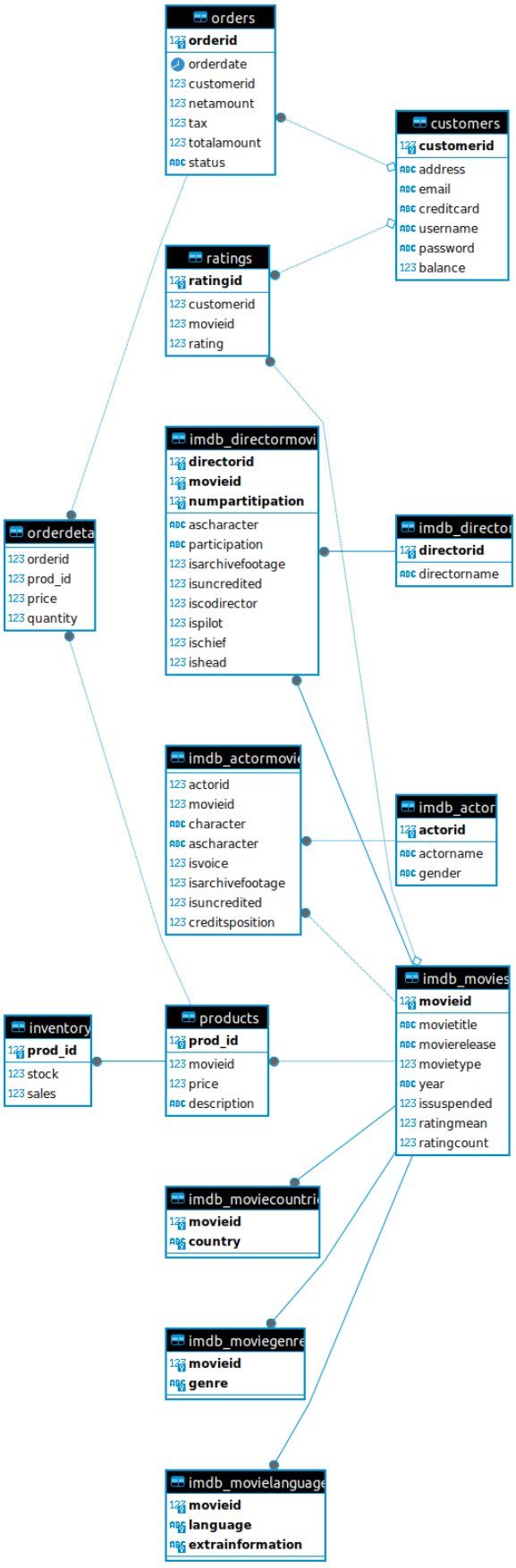


Diagrama Entidad-Relación tras ejecutar “actualiza.sql”



APARTADO A:

Se añaden 6 claves foráneas que faltan aparte de añadir el nuevo campo balance a la tabla “customers”, que será 10 números usando los dos últimos para los decimales.

También creamos la nueva tabla “ratings” que estará formada por los campos “ratingid”, “customerid” que será una clave foránea conectada con la tabla “customers”, “movieid” que será una clave foránea conectada con la tabla “imdb_movies” y “rating” que será un entero para valorar la película del 00.0 al 10.0. Además, tenemos en cuenta que un usuario solo pueda valorar una película una sola vez.

Añadimos los campos nuevos a la tabla “imdb_movies”: “ratingmean” que es la media de las valoraciones que hacen los usuarios, que es un número del 0 al 9 y con dos números reservados para los decimales, y “rating count”.

Por último, se actualiza el campo “password” de la tabla “customers” a 96 caracteres hexadecimales.

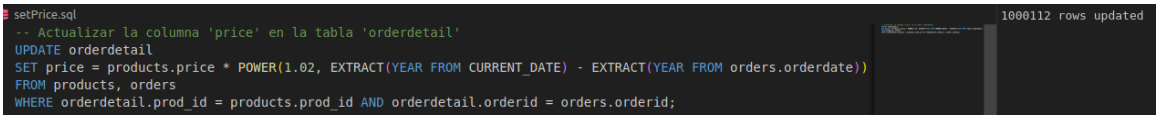
```sql
ALTER TABLE imdb_actormovies
ADD CONSTRAINT "imdb_actormovies_actorid_fkey"
FOREIGN KEY ("actorid")
REFERENCES imdb_actors("actorid");

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

ALTER TABLE orderdetail
ADD CONSTRAINT "imdb_orderdetail_orderid_fkey"
FOREIGN KEY ("orderid")
REFERENCES orders("orderid");

ALTER TABLE orderdetail
ADD CONSTRAINT "imdb_orderdetail_prod_id_fkey"
FOREIGN KEY ("prod_id")
REFERENCES products("prod_id");

ALTER TABLE orders
ADD CONSTRAINT "imdb_orders_customerid_fkey"
FOREIGN KEY ("customerid")
REFERENCES customers("customerid");

ALTER TABLE inventory
ADD CONSTRAINT "imdb_inventory_prod_id_fkey"
FOREIGN KEY ("prod_id")
REFERENCES products("prod_id");

ALTER TABLE customers
ADD COLUMN balance DECIMAL(10, 2);

UPDATE TABLE ratings (
    ratingid SERIAL PRIMARY KEY,
    customerid INTEGER REFERENCES customers("customerid"),
    movieid INTEGER REFERENCES imdb_movies("movieid"),
    rating DECIMAL(3, 1),
    -- Asegurarse de que un usuario no pueda valorar dos veces la misma película
    CONSTRAINT unique_user_movie_rating UNIQUE ("customerid", "movieid")
);

ALTER TABLE imdb_movies
ADD COLUMN ratingmean DECIMAL(3, 2),
ADD COLUMN ratingcount INTEGER;

ALTER TABLE customers
ALTER COLUMN password TYPE VARCHAR(96);
```

Creamos el procedimiento para inicializar el campo “balance” creado previamente entre un número del 0 al N, este último le pasamos tras la llamada siguiente. En este caso hacemos la llamada con un N = 200.

```sql
-- Crear o reemplazar el procedimiento almacenado
CREATE OR REPLACE FUNCTION setCustomersBalance(IN initialBalance bigint)
RETURNS void AS $$
DECLARE
    random_balance bigint;
BEGIN
    -- Generar un número aleatorio entre 0 y N
    random_balance := floor(random() * (initialBalance + 1));

    -- Actualizar el campo balance en la tabla customers con el valor aleatorio
    UPDATE customers
    SET balance = random_balance;

    RAISE NOTICE 'Balances actualizados aleatoriamente.';
END;
$$ LANGUAGE plpgsql;

-- Llamar al procedimiento con un valor específico para initialBalance
SELECT setCustomersBalance(200);
```

APARTADO B:

Con esta consulta estamos actualizando el campo “price” de la tabla “orderdetail” con el precio impuesto en el campo “price” de la tabla “products” + 2%. Como podemos ver, se actualizan las 1000112 que tenemos.



APARTADO C:

Para este caso, hemos creado una view para obtener la suma de los precios de las películas del pedido, para posteriormente usar esta información. Es necesaria para completar las columnas de “netamount” y “totalamount” de la tabla “orders”.

Para las pruebas, hemos hecho la llamada al procedimiento y también una consulta simple para ver si se ha completado correctamente estas columnas.

setOrderAmount.sql

```sql
CREATE OR REPLACE FUNCTION setOrderAmount()
RETURNS VOID
AS $$
BEGIN
    -- Creamos una view para obtener la suma de los precios de la
    -- película del pedido
    CREATE OR REPLACE VIEW OrderPrice AS
    SELECT
        orderid, SUM(price * quantity) as finalprice
    FROM
        orderdetail
    GROUP BY
        orderid
    ;

    -- Procedimiento que completa las columnas netamount y totalamount
    -- de la tabla orders
    UPDATE orders
    SET netamount = oprice.finalprice, totalamount = ROUND (oprice.finalprice + (opric
    FROM OrderPrice oprice
    WHERE orders.orderid = oprice.orderid;

END;
$$ LANGUAGE plpgsql;

-- Invocación al procedimiento
SELECT setOrderAmount();
SELECT * FROM orders LIMIT 100
```

null rows created

1 row returned

	setorderamount void
1	

100 rows returned

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying
1	76454	2022-06-23	5875	33.048000000000000000	18	39.00	Shipped
2	18889	2021-11-01	1430	135.252000000000000000	18	159.60	Shipped
3	19440	2022-04-20	1478	174.624000000000000000	18	206.06	Shipped
4	19047	2021-07-31	1445	71.37144000000000000	15	82.08	Paid
5	76508	2019-02-01	5879	15.1540502400000000	15	17.43	Paid
6	99173	2020-04-09	7652	73.435593600000000000	15	84.45	Paid
7	7302	2022-03-02	533	72.4200000000000000	18	85.46	Shipped
8	82950	2022-04-23	6373	160.854000000000000000	18	189.81	Shipped
9	62426	2017-02-06	4787	26.12696812692480000	15	30.05	Shipped
10	103052	2021-07-17	7955	94.676400000000000000	15	108.88	Shipped
11	162680	2022-01-05	12640	41.8200000000000000	18	49.35	Shipped
12	150591	2021-09-30	11675	134.731800000000000000	18	158.98	Shipped
13	41781	2019-04-15	3170	176.003469216000000000	15	202.40	Shipped
14	13128	2017-09-25	985	18.0185987082240000	15	20.72	Shipped
15	150344	2022-05-16	11650	187.476000000000000000	18	221.22	Shipped
16	6487	2021-10-15	474	330.847200000000000000	18	390.40	Shipped
17	9490	2021-12-24	685	72.8280000000000000	18	85.94	Shipped
18	102855	2018-09-14	7939	71.10280372608000000	15	81.77	Paid
19	130655	2021-03-11	10143	14.5656000000000000	15	16.75	Shipped
20	128681	2021-07-03	9972	269.463600000000000000	15	309.88	Shipped
21	72603	2021-03-04	5584	44.217000000000000000	15	50.85	Shipped
22	28686	2021-09-16	2197	46.8180000000000000	18	55.25	Shipped
23	15072	2020-06-21	1132	175.417682400000000000	15	201.73	Shipped
24	179255	2022-03-10	13907	13.2600000000000000	18	15.65	Shipped
25	1991	2020-05-12	139	54.758332800000000000	15	62.97	Paid
26	21870	2022-04-22	1665	123.624000000000000000	18	145.88	Shipped
27	131406	2017-06-12	10198	30.406385320128000	15	34.97	Shipped
28	11761	2021-05-03	879	32.876640000000000000	15	37.81	Paid
29	149443	2022-02-11	11582	94.044000000000000000	18	110.97	Shipped
30	77353	2022-05-23	5945	71.4000000000000000	18	84.25	Shipped
31	14314	2022-03-25	1077	76.5000000000000000	18	90.27	Shipped
32	110427	2021-12-16	8527	34.853400000000000000	18	41.13	Shipped
33	174961	2021-07-28	13569	116.004600000000000000	15	133.41	Shipped
34	6959	2021-02-17	510	209.120400000000000000	15	240.49	Shipped
35	71776	2021-07-13	5523	52.0200000000000000	15	59.82	Shipped

APARTADO D:

Como podemos comprobar con esta función, realizamos una consulta para ver las películas con más ventas de los años que indicamos al llamar a la función y pasamos como argumento. Hemos hecho uso de la tupla para la facilitación del código y poder usar mejor estos campos.

getTopSales.sql

```sql
CREATE OR REPLACE FUNCTION getTopSales(year1 INT, year2 INT,
    OUT Year INT, OUT Film CHAR, OUT sales BIGINT)
RETURNS SETOF RECORD
AS $$
DECLARE
    tupla RECORD;
BEGIN
    CREATE TEMPORARY TABLE ventas AS
        (SELECT movieid, sum(q.salesperyear) AS salesperyear,
            q.yearofsale
        FROM (SELECT od.prod_id, sum(quantity) AS salesperyear,
                extract(YEAR FROM orderdate) AS yearOfSale
                FROM orderdetail AS od NATURAL JOIN orders
                GROUP BY yearOfSale, od.prod_id) AS q NATURAL JOIN
            products
        GROUP BY movieid, yearofsale);
    CREATE TEMPORARY TABLE res AS
        (SELECT q.year, m.movietitle, q.maxsales
        FROM (SELECT s2.yearofsale AS year,
                min(s1.movieid) AS movieid, s2.maxsales
                FROM ventas AS s1,
                    (SELECT max(s.salesperyear) AS maxsales,
                        s.yearofsale
                    FROM ventas AS s
                    GROUP BY s.yearofsale) AS s2
                WHERE s2.maxsales = s1.salesperyear AND
                        s2.yearofsale = s1.yearofsale
                GROUP BY year, s2.maxsales) AS q
            JOIN imdb_movies AS m ON m.movieid = q.movieid
        WHERE q.year >= year1 AND q.year <= year2
        ORDER BY q.maxsales DESC);
    FOR tupla IN SELECT * FROM res LOOP
        Year := tupla.year;
        Film := tupla.movietitle;
        sales := tupla.maxsales;
        RETURN NEXT;
    END LOOP;
    DROP TABLE ventas;
    DROP TABLE res;
    RETURN;
END; $$
LANGUAGE plpgsql;

-- Invocación al procedimiento
SELECT * FROM getTopSales(2020, 2021);
```

2 rows returned

	year integer	film bpchar	sales bigint
1	2020	Wizard of Oz, The (1939)	135
2	2021	Stand by Me (1986)	130

APARTADO E:

Esta función devuelve el nombre del actor, el número de películas que hizo del género que recibe, el año del debut del actor, título de la película y director de dicha película. Para su implementación hemos creado una tabla auxiliar (RESULT) en la que guardamos el resultado de la query, después para devolverlo, usamos una tupla y un bucle para asignar cada columna de salida a una columna del resultado.

Para probar su correcto funcionamiento hemos implementado una query auxiliar que recibiendo el nombre de un actor y un género, devuelve el número de películas que hizo:

getTopActors.sql

```sql
CREATE OR REPLACE FUNCTION getTopActors(genre CHAR, OUT Actor CHAR,
    OUT Num INT, OUT Debut INT, OUT Film CHAR, OUT Director CHAR)
RETURNS SETOF RECORD
AS $$
DECLARE
    tupla RECORD;
BEGIN
    CREATE TEMPORARY TABLE RESULT
    AS (SELECT q.actorname, q.genremovies, q.genredebut,
            m.movietitle, d.directorname
        FROM (
            SELECT a.actorid, a.actorname, COUNT(a.actorid) AS genremovies,
                MIN(m.year) AS genredebut
            FROM imdb_moviegenres AS g
            JOIN imdb_movies AS m ON m.movieid = g.movieid
            JOIN imdb_actormovies AS am ON am.movieid = m.movieid
            JOIN imdb_actors AS a ON a.actorid = am.actorid
            WHERE g.genre LIKE $1
            GROUP BY a.actorid, a.actorname
            HAVING COUNT(a.actorid) > 4
        ) AS q
        JOIN imdb_actormovies AS am ON am.actorid = q.actorid
        JOIN imdb_movies AS m ON q.genredebut = m.year AND am.movieid = m.movieid
        JOIN imdb_directormovies AS dm ON dm.movieid = m.movieid
        JOIN imdb_directors AS d ON d.directorid = dm.directorid
        ORDER BY q.genremovies DESC);

    FOR tupla IN SELECT * FROM RESULT LOOP
        Actor := tupla.actorname;
        Num := tupla.genremovies;
        Debut := tupla.genredebut;
        Film := tupla.movietitle;
        Director := tupla.directorname;
        RETURN NEXT;
    END LOOP;

    DROP TABLE RESULT;
END;
$$ LANGUAGE plpgsql;
```

2681 rows returned

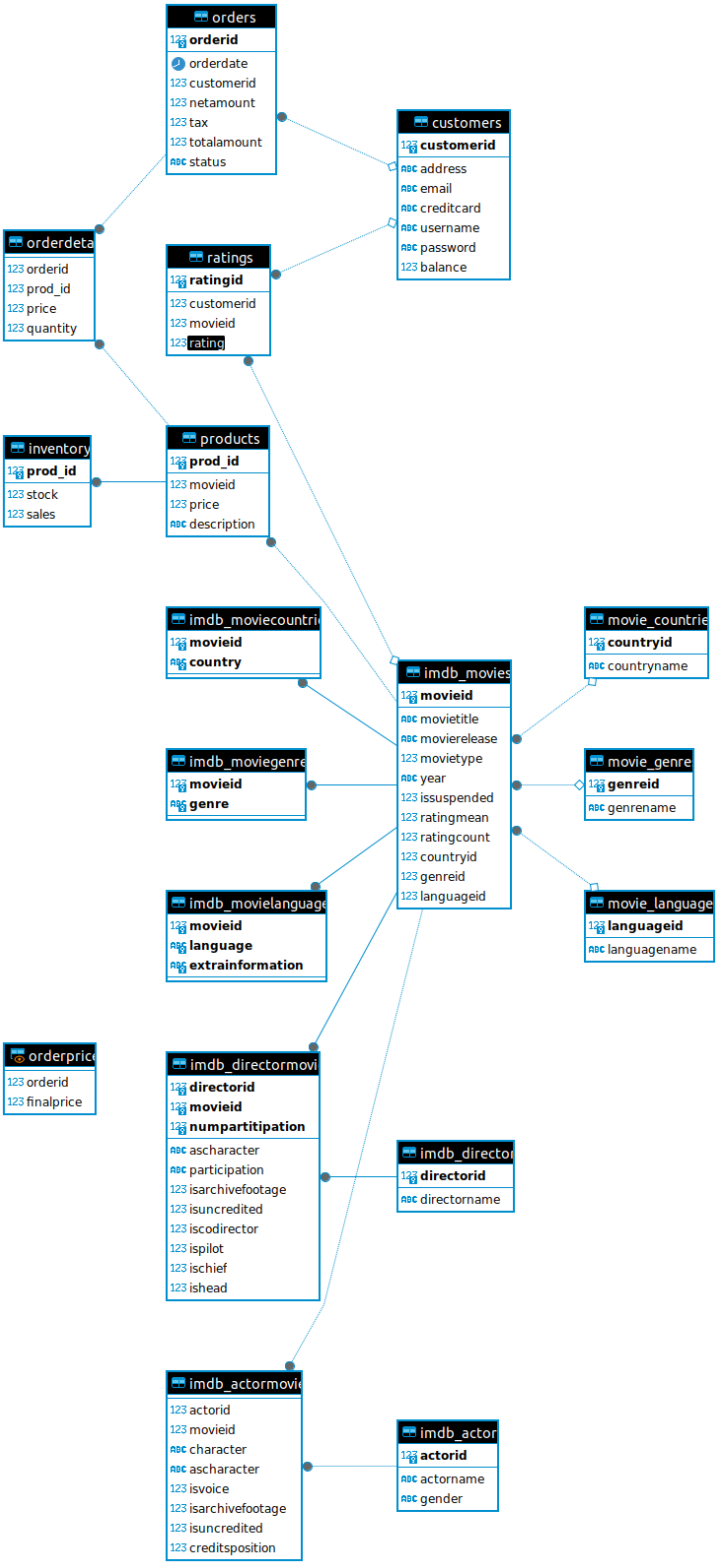
	actor bpchar	num integer	debut integer	film bpchar	director bpchar
1	Jackson, Samuel L.	26	1988	School Daze (1988)	Lee, Spike
2	Duvall, Robert (I)	26	1962	To Kill a Mockingbird (1962)	Mulligan, Robert
3	De Niro, Robert	24	1971	Born to Win (1971)	Passer, Ivan
4	Walsh, M. Emmet	23	1969	Midnight Cowboy (1969)	Schlesinger, John
5	Keitel, Harvey	22	1976	Taxi Driver (1976)	Scorsese, Martin
6	Walsh, J.T.	21	1987	Tin Men (1987)	Levinson, Barry (I)
7	Turturro, John	21	1980	Raging Bull (1980)	Scorsese, Martin
8	Walsh, J.T.	21	1987	Good Morning, Vietnam (1987)	Levinson, Barry (I)
9	Hitchcock, Alfred (I)	21	1927	Lodger, The (1927)	Hitchcock, Alfred (I)
10	Nicholson, Jack	21	1969	Easy Rider (1969)	Hopper, Dennis
11	Costner, Kevin	19	1982	Night Shift (1982)	Howard, Ron
12	Costner, Kevin	19	1982	Frances (1982)	Clifford, Graeme
13	Corrigan, Kevin (I)	18	1990	Goodfellas (1990)	Scorsese, Martin
14	Berkeley, Xander	18	1981	Mommie Dearest (1981)	Perry, Frank (I)
15	Hackman, Gene	18	1967	Bonnie and Clyde (1967)	Penn, Arthur
16	Corrigan, Kevin (I)	18	1990	Men Don't Leave (1990)	Brickman, Paul
17	Walken, Christopher	18	1978	Deer Hunter, The (1978)	Cimino, Michael
18	Macy, William H.	18	1980	Somewhere in Time (1980)	Szwarc, Jeannot
19	Corrigan, Kevin (I)	18	1990	Exorcist III, The (1990)	Blatty, William Peter
20	Pacino, Al	18	1972	Godfather, The (1972)	Coppola, Francis Ford
21	Byrne, Gabriel	18	1986	Gothic (1986)	Russell, Ken (I)
22	Argo, Victor	18	1976	Taxi Driver (1976)	Scorsese, Martin
23	Williams, Robin (I)	17	1987	Good Morning, Vietnam (1987)	Levinson, Barry (I)
24	Woods, James (I)	17	1973	Way We Were, The (1973)	Pollack, Sydney
25	Robards, Jason	17	1968	C'era una volta il West (1968)	Leone, Sergio (I)
26	Tobolowsky, Stephen	17	1990	Grifters, The (1990)	Frears, Stephen
27	Tobolowsky, Stephen	17	1990	Bird on a Wire (1990)	Badham, John
28	Stanton, Harry Dean	17	1959	Pork Chop Hill (1959)	Milestone, Lewis
29	Hedaya, Dan	17	1977	Prince of Central Park, The (1977)	Hart, Harvey
30	Malkovich, John	17	1984	Killing Fields, The (1984)	Joffé, Roland
31	Guilfoyle, Paul (II)	17	1993	Naked in New York (1993)	Algrant, Daniel
32	Guilfoyle, Paul (II)	17	1993	Mrs. Doubtfire (1993)	Columbus, Chris
33	Freeman, Morgan (I)	17	1964	Pawnbroker, The (1964)	Lumet, Sidney
34	Goldberg, Whoopi	17	1985	Color Purple, The (1985)	Spielberg, Steven (I)

Query para comprobar el resultado:

```sql
SELECT
    imdb_actors.actorname
FROM
    imdb_actors
JOIN
    imdb_actormovies ON imdb_actors.actorid = imdb_actormovies.actorid
JOIN
    imdb_movies ON imdb_actormovies.movieid = imdb_movies.movieid
JOIN
    imdb_moviegenres ON imdb_movies.movieid = imdb_moviegenres.movieid
WHERE
    imdb_actors.actorname = 'Lee, Spike'
    AND imdb_moviegenres.genre = 'Drama';
```

APARTADO F:

Hemos creado 3 nuevas tablas: “movie_countries”, “movie_genre” y “movie_languages”. Además, hemos establecido las relaciones con la tabla “imdb_movies” con las claves foráneas “countryid”, “genreid” y “languageid”.



```sql
-- Crear las tablas adicionales
CREATE TABLE IF NOT EXISTS movie_countries (
    countryid SERIAL PRIMARY KEY,
    countryname VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS movie_genres (
    genreid SERIAL PRIMARY KEY,
    genrename VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS movie_languages (
    languageid SERIAL PRIMARY KEY,
    languagename VARCHAR(255) NOT NULL
);

-- Añadir las columnas de clave foránea a la tabla movies
ALTER TABLE IF EXISTS imdb_movies
ADD COLUMN IF NOT EXISTS countryid INTEGER REFERENCES movie_countries(countryid),
ADD COLUMN IF NOT EXISTS genreid INTEGER REFERENCES movie_genres(genreid),
ADD COLUMN IF NOT EXISTS languageid INTEGER REFERENCES movie_languages(languageid);

-- Eliminar las columnas antiguas multivaluadas
ALTER TABLE IF EXISTS imdb_movies
DROP COLUMN IF EXISTS moviecountries,
DROP COLUMN IF EXISTS moviegenres,
DROP COLUMN IF EXISTS movielanguages;
```

APARTADO G:

Para este apartado hemos creado un trigger que al detectar un cambio en alguna fila de “orderdetail”, actualizará los valores “netamount” y “totalnetamount” de “orders”. Para ello hemos pensado en como calcular la actualización de los valores anteriormente mencionados, la operación es distinta para cada tipo de actualización, para insertar, hay que calcular el “netamount” y sumarlo al total, para actualizar una fila, hay que sacar la diferencia de precios entre el antiguo “netamount” y el actual, de esta forma si se actualiza a la baja o a la alza funciona de la misma manera, por último, para eliminar, es lo mismo que insertar pero en vez de sumar se restan los valores. Para probar el correcto funcionamiento, hemos tenido que crear una query que nos permita ver que el trigger se ha ejecutado.

updOrders.sql

```
CREATE OR REPLACE FUNCTION updOrders()
RETURNS TRIGGER
AS $$
DECLARE
    price int4;
BEGIN
    IF (TG_OP = 'INSERT') THEN
        price := (select o.netamount from orders o where o.orderid = new.orderid);
        UPDATE orders set netamount = price + (new.price*new.quantity) where orders.orderid = new.orderid;
        UPDATE orders set totalamount = (netamount + (netamount*tax/100)) where orders.orderid = new.orderid;
    ELSEIF (TG_OP = 'UPDATE') THEN
        price := (select o.netamount from orders o where o.orderid = old.orderid);
        price := price - (old.price*old.quantity);
        UPDATE orders set netamount = price + (new.price*new.quantity) where orders.orderid = new.orderid;
        UPDATE orders set totalamount = (netamount + (netamount*tax/100)) where orders.orderid = new.orderid;
    ELSEIF (TG_OP = 'DELETE') THEN
        price := (select o.netamount from orders o where o.orderid = old.orderid);
        UPDATE orders set netamount = price - (old.price*old.quantity) where orders.orderid = old.orderid;
        UPDATE orders set totalamount = (netamount + (netamount*tax/100)) where orders.orderid = old.orderid;
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE 'plpgsql';

CREATE OR REPLACE TRIGGER updOrders
AFTER DELETE OR INSERT OR UPDATE ON orderdetail
FOR EACH ROW EXECUTE PROCEDURE updOrders();
```

Query para probarlo:

```
UPDATE orderdetail SET price = 33 WHERE orderid = 99997;

SELECT * FROM orders o WHERE o.orderid = 99997;
```

Antes:

1 row returned

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying
1	99997	2019-06-16	7716	187.80197976000000000	15	215.97	Processed

Después:

1 row returned

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying
1	99997	2019-06-16	7716	297	15	341.5500000000000000	Processed

APARTADO H:
Para este apartado hemos usado el mismo formato que en el apartado anterior, esta vez no hacía falta guardar en una variable el resultado de una consulta ya que lo tenemos en un campo de la tabla NEW. Esta vez las fórmulas matemáticas para la resolución del ejercicio las hemos planteado lógicamente con nuestros conocimientos.
Para probar que funciona hemos implementado una serie de consultas auxiliares que nos permitían insertar o modificar elementos de las tablas implicadas en el trigger, concretamente, hemos insertado ratings de usuarios aleatorios refiriendo el id de una película que no existía, de ahí, la necesidad de insertar una nueva película.

updRatings.sql
```sql
CREATE OR REPLACE FUNCTION updRatings()
RETURNS TRIGGER
AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        UPDATE imdb_movies SET ratingmean = ((ratingcount * ratingmean) + NEW.rating) / (ratingcount + 1), ratingcount = ratingcount + 1;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE imdb_movies SET ratingmean = ((ratingmean * ratingcount) + (NEW.rating - OLD.rating));
    ELSIF (TG_OP = 'DELETE') THEN
        UPDATE imdb_movies SET ratingmean = ((ratingcount * ratingmean) - OLD.rating) / (ratingcount - 1), ratingcount = ratingcount - 1;
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER updRatings
AFTER DELETE OR INSERT OR UPDATE ON ratings
FOR EACH ROW EXECUTE FUNCTION updRatings();
```

Consultas auxiliares:

```sql
SELECT * FROM ratings
SELECT ratingmean, ratingcount FROM imdb_movies WHERE movieid = 100

INSERT INTO ratings (ratingid, rating, customerid, movieid)
VALUES (13, 10.0, 4, 100);



INSERT INTO imdb_movies (movieid, movietitle, moviereleased,
    movietype, year, issuspended, ratingmean, ratingcount)
VALUES (101, 'wow', 'ayer', 1, 2010, 0, 0.01, 1);
```

APARTADO I:
Este apartado nos propone reducir el stock de un producto cuando el estado de un pedido se cambia a 'Paid', además, hay que aumentar las unidades vendidas de ese producto. También se nos pide que modifiquemos el balance del cliente que ha pagado el pedido restando el "totalamount" del pedido.

Para probar el correcto funcionamiento de nuestra función hemos creado consultas auxiliares que nos permiten obtener el id de determinado producto a través del id de un pedido, también obtenemos el id del cliente, con todos estos datos tenemos acceso al antiguo balance del cliente y a la fila correspondiente al producto de la tabla "inventory". Por tanto, podemos comparar los datos antes y después de ejecutar el trigger, de ahí afirmamos que nuestra consulta es correcta.

updInventoryAndCustomer.sql

```sql
CREATE OR REPLACE FUNCTION updInventoryAndCustomer()
RETURNS TRIGGER
AS $$
DECLARE
    prod record;
BEGIN
    FOR prod IN
        SELECT
            od.prod_id, i.sales, od.quantity, i.stock
        FROM
            public.orderdetail od,
            public.inventory i
        WHERE
            OLD.orderid = od.orderid AND
            i.prod_id = od.prod_id

    LOOP
        UPDATE inventory i
        SET
            stock = prod.stock - prod.quantity,
            sales = prod.sales + prod.quantity
        WHERE
            i.prod_id = prod.prod_id;

        IF (prod.quantity >= prod.stock) THEN
            INSERT INTO alertas VALUES (prod.prod_id, NOW(), prod.stock - prod.quantity);
        END IF;
    END LOOP;
    UPDATE customers SET balance = balance - NEW.totalamount;
    NEW.orderdate = 'NOW()';
    RETURN NEW;
END; $$
LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER updInventoryAndCustomer
BEFORE UPDATE OF STATUS ON orders
FOR EACH ROW
    WHEN (NEW.status = 'Paid')
    EXECUTE PROCEDURE updInventoryAndCustomer();
```

Querys auxiliares usadas para probar el trigger:

```
SELECT * FROM orders o WHERE o.orderid = 95
SELECT * FROM orderdetail o WHERE o.orderid = 95

UPDATE orders SET status = 'Paid' WHERE orderid = 95;
UPDATE orders SET status = 'Processed' WHERE orderid = 95;

UPDATE customers SET balance = 50 where customerid = 9329

SELECT * FROM customers WHERE customerid = 9329
SELECT * FROM inventory WHERE inventory.prod_id = 1072
```

Antes:

1 row returned

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying
1	95	2023-11-16	9329	36.0000000000000000	15	41.40	Processed

1 row returned

	customerid integer	address character varying	email character varying	creditcard character varying	username character varying	password character varying	balance numeric
1	9329	raw lowish 260	hoar.oise@kran.com	4742067881884429	gooier	podium	50.00

1 row returned

	prod_id integer	stock integer	sales integer
1	1072	732	194

Después:

1 row returned

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying
1	95	2023-11-16	9329	36.0000000000000000	15	41.40	Paid

1 row returned

	customerid integer	address character varying	email character varying	creditcard character varying	username character varying	password character varying	balance numeric
1	9329	raw lowish 260	hoar.oise@kran.com	4742067881884429	gooier	podium	8.60

1 row returned

	prod_id integer	stock integer	sales integer
1	1072	731	195

APARTADO J:
Para el uso de sqlalchemy debemos de instalar antes los paquetes necesarios por terminal usando el comando “pip install sqlalchemy” y después, hay que importar en el fichero “mostrarTabla.py” lo necesario. En este caso, usamos el create_engine para establecer la conexión con la base de datos. Tras esto, ejecutamos nuestra función del apartado d) para realizar la consulta y sacar los datos de los dos últimos años (2021 y 2022).

```python
from sqlalchemy import create_engine, text

# Establecer la conexión a la base de datos PostgreSQL
engine = create_engine('postgresql://alumnodb:1234@localhost:5432/si1')

# Ejecutar la función getTopSales con SQLAlchemy
with engine.connect() as con:
    query = text("SELECT * FROM getTopSales(:year1, :year2)")
    result = con.execute(query.bindparams(year1=2021, year2=2022))

    # Mostrar los resultados
    for row in result.fetchmany(10):  # Limitar a 10 filas
        print(row)
```

```
jorge@jorge-virtual-machine:~/Desktop/SI/P2$ python3 mostrarTabla.py
(2021, 'Stand by Me (1986)', 130)
(2022, 'Jerk, The (1979)', 52)
jorge@jorge-virtual-machine:~/Desktop/SI/P2$
```

2 rows returned

	year integer	film bpchar	sales bigint
1	2021	Stand by Me (1986)	130
2	2022	Jerk, The (1979)	52

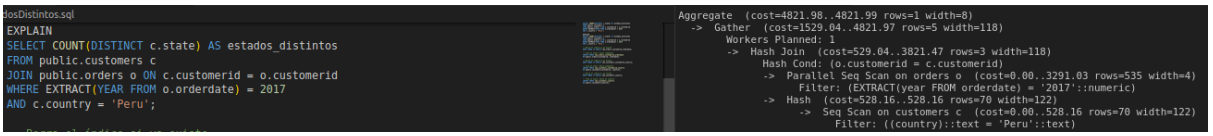
APARTADO K:

Con esta query mostramos el número de estados distintos (DISTINT) con clientes que tienen pedidos en un año dado y que además pertenecen a Perú.

```
dosDistintos.sql
SELECT COUNT(DISTINCT c.state) AS estados_distintos
FROM public.customers c
JOIN public.orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = 2017
AND c.country = 'Peru';
```

```
1 row returned
 estados_distintos
 bigint
1 185
```

El plan indica que se está haciendo un escaneo secuencial en paralelo en la tabla “orders” con un filtro para el año 2017 y un hash sobre la tabla “customers” con un filtro para el país Perú. Estas operaciones se unen utilizando el customerid.

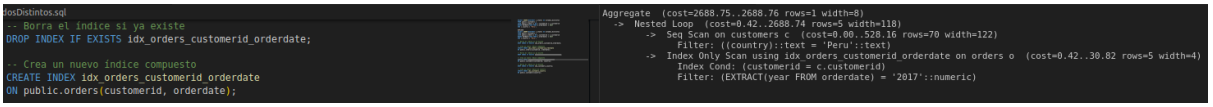


```
dosDistintos.sql
EXPLAIN
SELECT COUNT(DISTINCT c.state) AS estados_distintos
FROM public.customers c
JOIN public.orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = 2017
AND c.country = 'Peru';
```

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

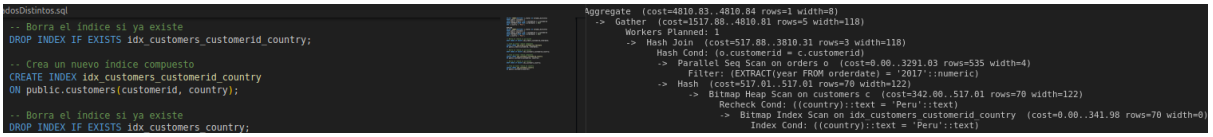
Con este índice, mejoramos mucho el rendimiento de la consulta. Bajamos de un coste de 4821,98 a uno de 2688,75. Para el resto también bajamos algo el coste pero no tan notorio como el primero.

Este nuevo plan muestra mejoras ya que utiliza un escaneo de índice para las operaciones en la tabla orders, lo que reduce significativamente el costo en comparación con el escaneo secuencial sin ningún índice.



```
dosDistintos.sql
-- Borra el índice si ya existe
DROP INDEX IF EXISTS idx_orders_customerid_orderdate;

-- Crea un nuevo índice compuesto
CREATE INDEX idx_orders_customerid_orderdate
ON public.orders(customerid, orderdate);
```

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



```
dosDistintos.sql
-- Borra el índice si ya existe
DROP INDEX IF EXISTS idx_customers_customerid_country;

-- Crea un nuevo índice compuesto
CREATE INDEX idx_customers_customerid_country
ON public.customers(customerid, country);

-- Borra el índice si ya existe
DROP INDEX IF EXISTS idx_customers_country;
```

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

```
dosDistintos.sql
-- Borra el índice si ya existe
DROP INDEX IF EXISTS idx_customers_country;

-- Crea un nuevo índice en country
CREATE INDEX idx_customers_country
ON public.customers(country);
```

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

APARTADO L:

Consulta 1:

Utiliza una subconsulta para encontrar los “customerid” en la tabla “customers” que no están presentes en la tabla orders con estado 'Paid'.

Plan de Ejecución: Utiliza un escaneo secuencial en la tabla orders con un filtro para encontrar aquellos con estado 'Paid'. Luego, realiza un escaneo índice solo en la clave primaria de la tabla customers y aplica un filtro con la subconsulta anterior.

Consulta 2:

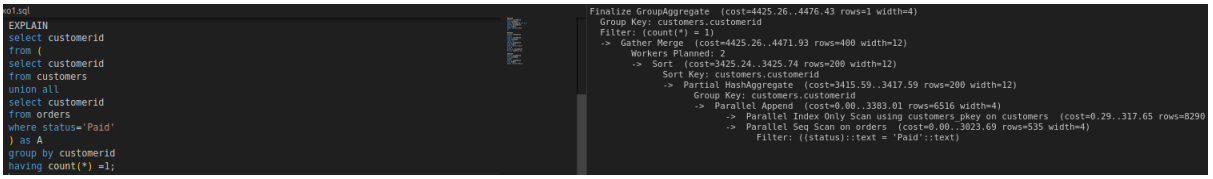
Emplea una unión de “customerid” de las tablas “customers" y “orders" con estado 'Paid', luego agrupa por “customerid” y filtra aquellos que tienen un recuento de 1.

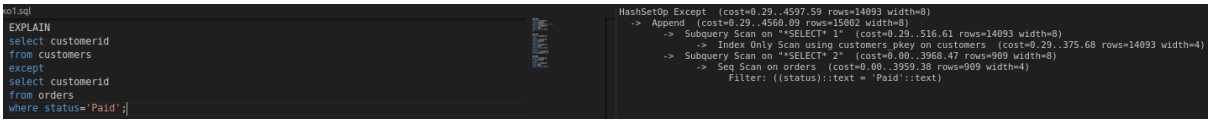
Plan de Ejecución: Utiliza una combinación de operaciones, primero realiza una unión de las tablas, luego una agrupación y finalmente un filtrado de los resultados.

Consulta 3:

Utiliza la operación EXCEPT entre los conjuntos de “customerid" de las tablas “customers” y “orders” con estado 'Paid'.

Plan de Ejecución: Realiza un escaneo índice solo en la clave primaria de “customers” y un escaneo secuencial en “orders” con un filtro para el estado 'Paid'. Luego, aplica la operación EXCEPT para encontrar las diferencias entre los conjuntos.





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

La tercera consulta parece ser la que devuelve algún resultado nada más comenzar su ejecución. Esto se debe a que utiliza un escaneo índice solo en la tabla `customers` y un escaneo secuencial en la tabla `orders` con un filtro para el estado 'Paid', lo que implica menos operaciones y filtros en comparación con las otras consultas.

Al realizar un EXCEPT entre los conjuntos de “customerid” de las tablas “customers” y “orders” con estado 'Paid', puede empezar a devolver resultados a medida que encuentra las diferencias entre los conjuntos. Esto significa que en cuanto se encuentre un “customerid” en la tabla “customers” que no esté presente en la tabla “orders” con estado 'Paid', se retornará como resultado.

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

La segunda consulta es la que más se adecua a la ejecución en paralelo, implica operaciones como unión, agrupación y filtrado sobre conjuntos de datos. Estas operaciones suelen ser susceptibles de ejecución en paralelo, ya que pueden dividirse en tareas más pequeñas que pueden ser procesadas por múltiples núcleos o procesadores de manera simultánea.

APARTADO M:
En este apartado simplemente nos hemos documentado sobre el uso de las sentencias que se nos proponen (EXPLAIN, ANALYZE Y CREATE INDEX). Realmente, los apartados c, e y g, es ejecutar la misma consulta pero después de agregar el índice o usar ANALYZE.
El resultado de las consultas está en el orden propuesto por el enunciado (c, e, g, h) de arriba abajo.

```
--c
EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status IS NULL;

EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status ='Shipped';

--d
CREATE INDEX idx_status ON orders(status);

--e
EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status IS NULL;

EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status ='Shipped';

--f
ANALYZE orders;

--g
EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status IS NULL;

EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status ='Shipped';

--h
EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status ='Paid';

EXPLAIN SELECT COUNT(*)
FROM orders
WHERE status ='Processed';
```

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

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

```
Aggregate  (cost=1488.40..1488.41 rows=1 width=8)
  ->  Bitmap Heap Scan on orders  (cost=11.34..1486.12 rows=909 width=0)
        Recheck Cond: (status IS NULL)
        ->  Bitmap Index Scan on idx_status  (cost=0.00..11.11 rows=909 width=0)
              Index Cond: (status IS NULL)

Aggregate  (cost=1490.67..1490.68 rows=1 width=8)
  ->  Bitmap Heap Scan on orders  (cost=11.34..1488.40 rows=909 width=0)
        Recheck Cond: ((status)::text = 'Shipped'::text)
        ->  Bitmap Index Scan on idx_status  (cost=0.00..11.11 rows=909 width=0)
              Index Cond: ((status)::text = 'Shipped'::text)
```

```
Aggregate  (cost=7.16..7.17 rows=1 width=8)
  ->  Index Only Scan using idx_status on orders  (cost=0.29..7.16 rows=1 width=0)
        Index Cond: (status IS NULL)

Finalize Aggregate  (cost=4211.64..4211.65 rows=1 width=8)
  ->  Gather  (cost=4211.53..4211.64 rows=1 width=8)
        Workers Planned: 1
        ->  Partial Aggregate  (cost=3211.53..3211.54 rows=1 width=8)
              ->  Parallel Seq Scan on orders  (cost=0.00..3023.69 rows=75136 width=0)
                    Filter: ((status)::text = 'Shipped'::text)
```

```
Aggregate  (cost=2151.35..2151.36 rows=1 width=8)
  ->  Bitmap Heap Scan on orders  (cost=200.58..2107.39 rows=17585 width=0)
        Recheck Cond: ((status)::text = 'Paid'::text)
        ->  Bitmap Index Scan on idx_status  (cost=0.00..196.18 rows=17585 width=0)
              Index Cond: ((status)::text = 'Paid'::text)

Aggregate  (cost=2645.06..2645.07 rows=1 width=8)
  ->  Bitmap Heap Scan on orders  (cost=410.96..2553.87 rows=36473 width=0)
        Recheck Cond: ((status)::text = 'Processed'::text)
        ->  Bitmap Index Scan on idx_status  (cost=0.00..401.84 rows=36473 width=0)
              Index Cond: ((status)::text = 'Processed'::text)
```

¿Qué hace el generador de estadísticas?
El generador de estadísticas recopila información sobre la distribución de datos en las tablas, lo que ayuda al optimizador de consultas a tomar decisiones más informadas sobre cómo ejecutar consultas de manera eficiente.

¿Por qué la planificación de las dos consultas es la misma hasta que se generan las estadísticas?
Hasta que se generan estadísticas, el optimizador de consultas puede tener información limitada sobre la distribución de los datos. La planificación puede ser más genérica.
Después de generar estadísticas, el optimizador tiene información más detallada, lo que puede influir en la elección del plan de ejecución más eficiente.