Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

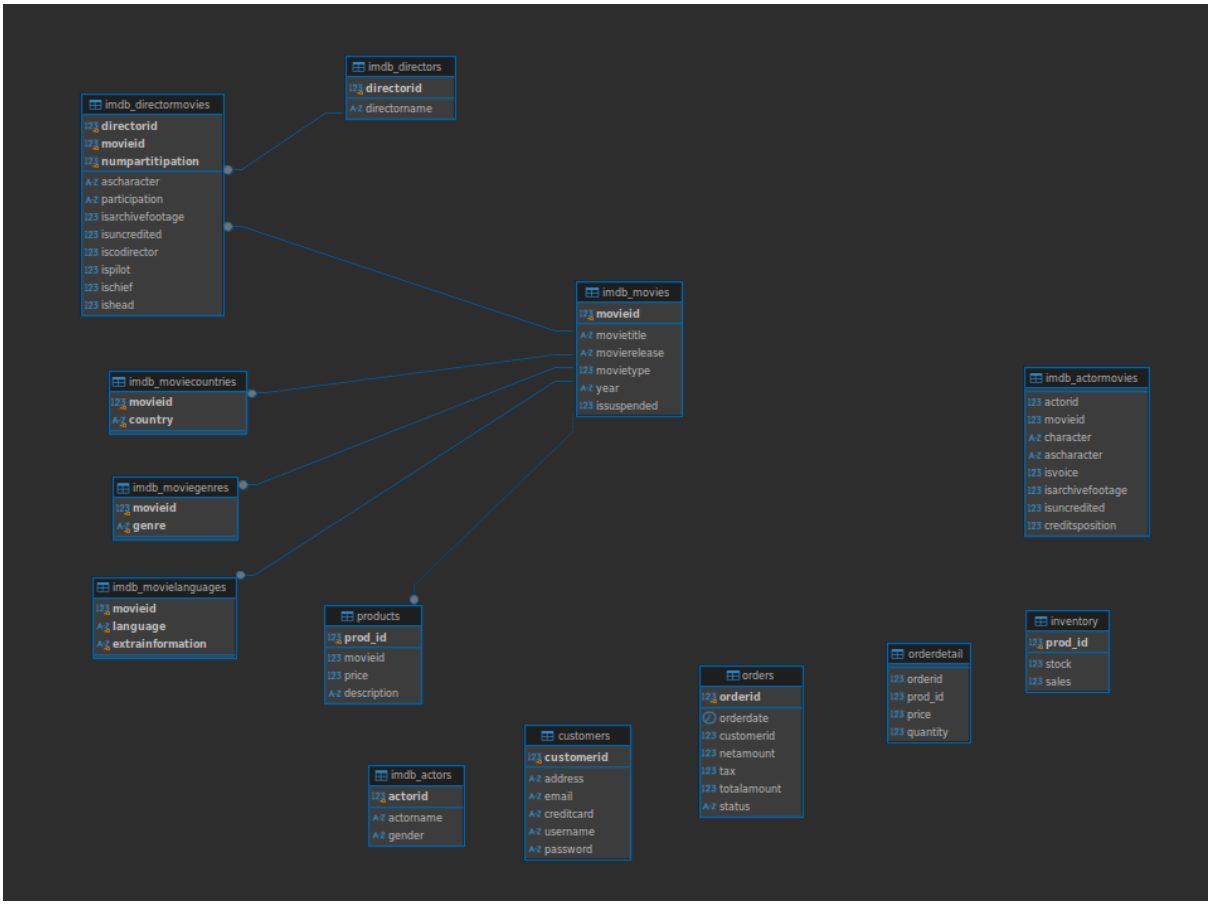
Práctica 2. Conceptos Avanzados sobre Bases de Datos Relacionales y Optimización de Consultas

AVISO:

Hemos hecho un README.md con las sintrucciones para probar esta práctica documentado y formateado que se puede leer correctamente desde nuestro repositorio de GitHub: https://github.com/kunn04/Sistemas-Inform-ticos/tree/main/Practica2

-Al comienzo de la práctica se pide proporcionar el diagrama ER de la base de datos suministrada, tanto antes como después de aplicar los cambios que consideremos necesarios en dicha base en una serie de scripts.

Aquí tenemos una primera captura de dicho diagrama:



(imagen obtenida a través de la app debaver).

De forma resumida, podemos ver que la base de datos combina información sobre películas y un sistema de ventas en línea. La tabla principal, 'imdb_movies', representa las películas disponibles para compra, mientras que `products` almacena los distintos productos asociados

a cada película (como ediciones o formatos) junto con sus precios. La tabla `inventory` rastrea el stock y las ventas de cada producto. Para la gestión de clientes, la tabla `customers` almacena la información de los usuarios, y `orders` organiza los pedidos de cada cliente, que pueden estar finalizados (pagados) o en curso. Finalmente, 'orderdetail' desglosa cada pedido

en productos específicos comprados, proporcionando un historial detallado de las compras realizadas en la web.

-A continuación, se nos pedía completar aquellos aspectos que se considerasen necesarios, tales como restricciones, claves extranjeras, cambios en cascada, etc. Además, todos estos cambios se debían incorporar en un único script, actualiza.sql convenientemente documentado.

La siguiente imagen hace referencia al script en cuestión:

```
-- Elimina los duplicados en orderdetail según orderid y prod_id,
conservando solo el primero
WITH cte AS (
   SELECT ctid, ROW_NUMBER() OVER (PARTITION BY orderid, prod_id ORDER
BY ctid) AS rn
   FROM orderdetail
)
DELETE FROM orderdetail
WHERE ctid IN (
   SELECT ctid FROM cte WHERE rn > 1
);

-- Define la clave primaria en orderdetail para orderid y prod_id
ALTER TABLE orderdetail
ADD CONSTRAINT pk_orderdetail
PRIMARY KEY (orderid, prod_id);

-- Establece una clave foránea en inventory que referencia a prod_id de
products
ALTER TABLE inventory
ADD CONSTRAINT fk_prod_id
FOREIGN KEY (prod_id) REFERENCES products(prod_id);

-- Define la clave primaria en imdb_actormovies para actorid y movieid
ALTER TABLE imdb_actormovies
ADD CONSTRAINT pk_actormovies
PRIMARY KEY (actorid, movieid);
```

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

```sql
-- Añade las claves foráneas en orderdetail para orderid y prod_id
ALTER TABLE orderdetail
ADD CONSTRAINT fk_orderid
FOREIGN KEY (orderid) REFERENCES orders(orderid);

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

-- Añade claves foráneas en imdb_actormovies para actorid y movieid
ALTER TABLE imdb_actormovies
ADD CONSTRAINT fk_actorid
FOREIGN KEY (actorid) REFERENCES imdb_actors(actorid);

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

-- Agrega una clave foránea en orders que referencia customerid de
customers
ALTER TABLE orders
ADD CONSTRAINT fk_customerid
FOREIGN KEY (customerid) REFERENCES customers(customerid);

-- Añade una columna balance en customers con un valor por defecto de
0.00
ALTER TABLE customers
ADD COLUMN balance DECIMAL(10, 2) DEFAULT 0.00;

-- Crea una tabla ratings para almacenar las valoraciones de los
clientes a las películas
CREATE TABLE ratings (
   customerid INT,
   movieid INT,
   likes BOOLEAN,
   rating_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   PRIMARY KEY (customerid, movieid),
   FOREIGN KEY (customerid) REFERENCES customers(customerid),
   FOREIGN KEY (movieid) REFERENCES imdb_movies(movieid)
);

-- Amplía el tamaño de la columna password en customers
ALTER TABLE customers
```

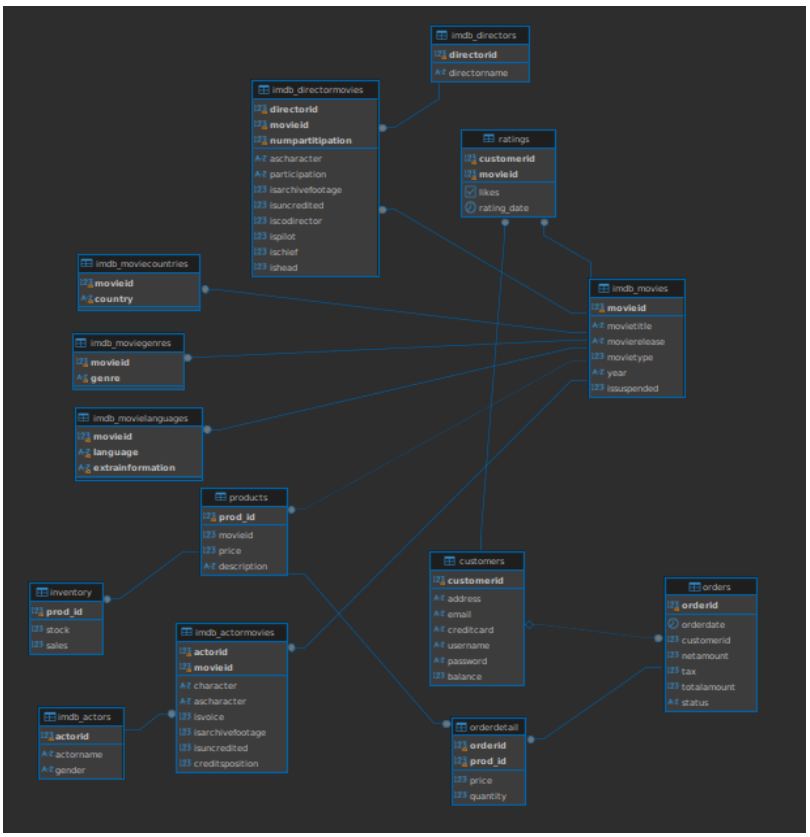
```
ALTER COLUMN password TYPE VARCHAR(128);

-- Asigna a cada cliente un balance aleatorio hasta initialBalance
CREATE OR REPLACE PROCEDURE setCustomersBalance (IN initialBalance
BIGINT)
LANGUAGE plpgsql
AS $$
BEGIN
   UPDATE customers
   SET balance = FLOOR(RANDOM() * initialBalance);
END
$$;;

-- Llama al procedimiento para establecer balances aleatorios con un
máximo de 200
CALL setCustomersBalance(200);
```

(imagen sacada directamente de nuestro código).

Este script, creado para completar aspectos pedidos en el enunciado. Primero, elimina duplicados en la tabla orderdetail para garantizar que cada combinación de orderid y prod_id sea única. Luego, añade claves primarias y foráneas en varias tablas (orderdetail, inventory, imdb_actormovies, orders, y ratings), lo cual refuerza la relación entre las tablas y asegura la coherencia de los datos. También agrega una columna balance en la tabla customers con un valor inicial de 0.00 para manejar el saldo de cada cliente y amplía la longitud de la columna password para mejorar la seguridad de las contraseñas. Finalmente, crea una tabla ratings para almacenar valoraciones de películas por los clientes, y un procedimiento setCustomersBalance que asigna un balance aleatorio a cada cliente hasta un límite definido. Estos cambios fortalecen la integridad de la base de datos y permiten nuevas funcionalidades, como la gestión de balances y valoraciones de usuarios.

-Tras ejecutar dicho script, la base de datos tiene la siguiente forma (diagrama ER):

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024



(captura obtenida de dbeaver)(1).

Podemos observar, que los cambios comentados anteriormente han sido correctamente aplicados.

-De la misma forma, ahora se nos pide hacer scripts del mismo estilo (apartados b, c, d, e) donde tenemos los siguientes:

Por un lado el archivo actualizaPrecios.sql crea procedimientos que calculan y actualizan el total a pagar por cada pedido en orders. El archivo actualizaTablas.sql normaliza la base de datos eliminando atributos multivaluados, creando tablas o relaciones adicionales según sea necesario. Finalmente, actualizaCarrito.sql define un trigger actualizaCarrito que ajusta automáticamente los totales y otros detalles en orders cada vez que se agrega, actualiza o elimina un artículo en el carrito (orderdetail), y pagado.sql que actualiza las tablas en consecuencia al cambiar el estado de un pedido a pagado.

- actualizaPrecios.sql, se trata del siguiente script:

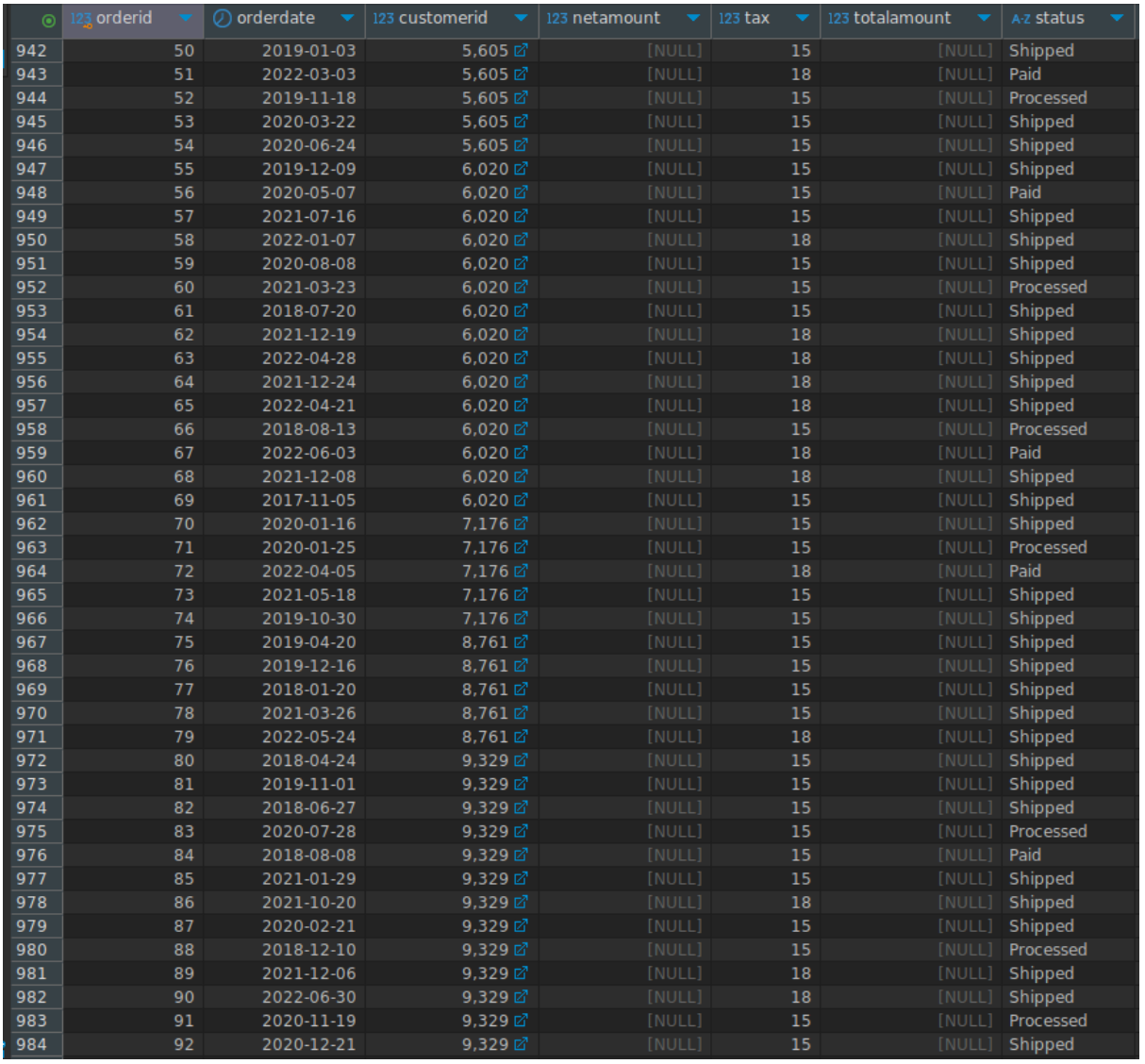
```
CREATE OR REPLACE PROCEDURE calculatePricesOrderdetail()
LANGUAGE plpgsql
AS $$
BEGIN
```

```
    UPDATE orderdetail od
    SET price = p.price * od.quantity
    FROM products p
    WHERE od.prod_id = p.prod_id;
END
$$;;

-- Procedimiento para calcular netamount y totalamount en orders
CREATE OR REPLACE PROCEDURE calculateAmountsOrders()
LANGUAGE plpgsql
AS $$
BEGIN
    -- Calcula el netamount (subtotal sin impuestos) para cada pedido
    UPDATE orders o
    SET netamount = (
        SELECT COALESCE(SUM(od.price), 0)  -- Suma los precios en orderdetail
        FROM orderdetail od
        WHERE od.orderid = o.orderid
    );

    -- Calcula el totalamount (total con impuestos) usando el netamount actualizado
    UPDATE orders o
    SET totalamount = netamount + (netamount * (o.tax / 100));
END
$$;;

-- Llamada a los procedimientos
CALL calculatePricesOrderdetail();
CALL calculateAmountsOrders();
```

(obtenido directamente desde nuestro código).

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

Este script hace cambios directos actualizando las columnas (anteriormente incompletas) de `netamount` y `totalamount` de la tabla de pedidos. De forma que anterior a la ejecución del script la tabla se mostraba de esta forma:



	orderid	orderdate	customerid	netamount	tax	totalamount	status
942	50	2019-01-03	5,605	[NULL]	15	[NULL]	Shipped
943	51	2022-03-03	5,605	[NULL]	18	[NULL]	Paid
944	52	2019-11-18	5,605	[NULL]	15	[NULL]	Processed
945	53	2020-03-22	5,605	[NULL]	15	[NULL]	Shipped
946	54	2020-06-24	5,605	[NULL]	15	[NULL]	Shipped
947	55	2019-12-09	6,020	[NULL]	15	[NULL]	Shipped
948	56	2020-05-07	6,020	[NULL]	15	[NULL]	Paid
949	57	2021-07-16	6,020	[NULL]	15	[NULL]	Shipped
950	58	2022-01-07	6,020	[NULL]	18	[NULL]	Shipped
951	59	2020-08-08	6,020	[NULL]	15	[NULL]	Shipped
952	60	2021-03-23	6,020	[NULL]	15	[NULL]	Processed
953	61	2018-07-20	6,020	[NULL]	15	[NULL]	Shipped
954	62	2021-12-19	6,020	[NULL]	18	[NULL]	Shipped
955	63	2022-04-28	6,020	[NULL]	18	[NULL]	Shipped
956	64	2021-12-24	6,020	[NULL]	18	[NULL]	Shipped
957	65	2022-04-21	6,020	[NULL]	18	[NULL]	Shipped
958	66	2018-08-13	6,020	[NULL]	15	[NULL]	Processed
959	67	2022-06-03	6,020	[NULL]	18	[NULL]	Paid
960	68	2021-12-08	6,020	[NULL]	18	[NULL]	Shipped
961	69	2017-11-05	6,020	[NULL]	15	[NULL]	Shipped
962	70	2020-01-16	7,176	[NULL]	15	[NULL]	Shipped
963	71	2020-01-25	7,176	[NULL]	15	[NULL]	Processed
964	72	2022-04-05	7,176	[NULL]	18	[NULL]	Paid
965	73	2021-05-18	7,176	[NULL]	15	[NULL]	Shipped
966	74	2019-10-30	7,176	[NULL]	15	[NULL]	Shipped
967	75	2019-04-20	8,761	[NULL]	15	[NULL]	Shipped
968	76	2019-12-16	8,761	[NULL]	15	[NULL]	Shipped
969	77	2018-01-20	8,761	[NULL]	15	[NULL]	Shipped
970	78	2021-03-26	8,761	[NULL]	15	[NULL]	Shipped
971	79	2022-05-24	8,761	[NULL]	18	[NULL]	Shipped
972	80	2018-04-24	9,329	[NULL]	15	[NULL]	Shipped
973	81	2019-11-01	9,329	[NULL]	15	[NULL]	Shipped
974	82	2018-06-27	9,329	[NULL]	15	[NULL]	Shipped
975	83	2020-07-28	9,329	[NULL]	15	[NULL]	Processed
976	84	2018-08-08	9,329	[NULL]	15	[NULL]	Paid
977	85	2021-01-29	9,329	[NULL]	15	[NULL]	Shipped
978	86	2021-10-20	9,329	[NULL]	18	[NULL]	Shipped
979	87	2020-02-21	9,329	[NULL]	15	[NULL]	Shipped
980	88	2018-12-10	9,329	[NULL]	15	[NULL]	Processed
981	89	2021-12-06	9,329	[NULL]	18	[NULL]	Shipped
982	90	2022-06-30	9,329	[NULL]	18	[NULL]	Shipped
983	91	2020-11-19	9,329	[NULL]	15	[NULL]	Processed
984	92	2020-12-21	9,329	[NULL]	15	[NULL]	Shipped

(obtenida a través de dbeaver).

Y tras ejecutar el script, se ve de esta forma:

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

	orderid	orderdate	customerid	netamount	tax	totalamount	status
942	52	2019-11-18	5,605	50.5	15	58.075	Processed
943	53	2020-03-22	5,605	81.4	15	93.61	Shipped
944	54	2020-06-24	5,605	34.6	15	39.79	Shipped
945	55	2019-12-09	6,020	151.6	15	174.34	Shipped
946	56	2020-05-07	6,020	82	15	94.3	Paid
947	57	2021-07-16	6,020	91.2	15	104.88	Shipped
948	58	2022-01-07	6,020	134	18	158.12	Shipped
949	59	2020-08-08	6,020	141.6	15	162.84	Shipped
950	60	2021-03-23	6,020	175.1	15	201.365	Processed
951	61	2018-07-20	6,020	156.2	15	179.63	Shipped
952	62	2021-12-19	6,020	56.5	18	66.67	Shipped
953	63	2022-04-28	6,020	176	18	207.68	Shipped
954	64	2021-12-24	6,020	95.3	18	112.454	Shipped
955	65	2022-04-21	6,020	43.8	18	51.684	Shipped
956	66	2018-08-13	6,020	121	15	139.15	Processed
957	67	2022-06-03	6,020	125	18	147.5	Paid
958	68	2021-12-08	6,020	54.2	18	63.956	Shipped
959	69	2017-11-05	6,020	100.7	15	115.805	Shipped
960	70	2020-01-16	7,176	24.2	15	27.83	Shipped
961	71	2020-01-25	7,176	103.4	15	118.91	Processed
962	72	2022-04-05	7,176	155.5	18	183.49	Paid
963	73	2021-05-18	7,176	44	15	50.6	Shipped
964	74	2019-10-30	7,176	120	15	138	Shipped
965	75	2019-04-20	8,761	100.3	15	115.345	Shipped
966	76	2019-12-16	8,761	192.2	15	221.03	Shipped
967	77	2018-01-20	8,761	77.4	15	89.01	Shipped
968	78	2021-03-26	8,761	141	15	162.15	Shipped
969	79	2022-05-24	8,761	82.2	18	96.996	Shipped
970	80	2018-04-24	9,329	107.9	15	124.085	Shipped
971	81	2019-11-01	9,329	111.4	15	128.11	Shipped
972	82	2018-06-27	9,329	78	15	89.7	Shipped
973	83	2020-07-28	9,329	153.8	15	176.87	Processed
974	84	2018-08-08	9,329	99.4	15	114.31	Paid
975	85	2021-01-29	9,329	53.6	15	61.64	Shipped
976	86	2021-10-20	9,329	18	18	21.24	Shipped
977	87	2020-02-21	9,329	43	15	49.45	Shipped
978	88	2018-12-10	9,329	12	15	13.8	Processed
979	89	2021-12-06	9,329	44.5	18	52.51	Shipped
980	90	2022-06-30	9,329	160.4	18	189.272	Shipped
981	91	2020-11-19	9,329	79.2	15	91.08	Processed
982	92	2020-12-21	9,329	79.4	15	91.31	Shipped
983	93	2019-10-12	9,329	117.8	15	135.47	Processed

(obtenida a partir de dbeaver).

Donde podemos observar la correcta actualización de los campos.

- actualizaTablas.sql, es el siguiente script:

```
create table creditcardCustomer (
    customerid INT not null,
    creditcard VARCHAR(50) not null,
    exp_date DATE null,
    cardholder VARCHAR(128) null,
    cvv VARCHAR(3) null,
    constraint pk_creditcardCustomer primary key (customerid,
creditcard),
```

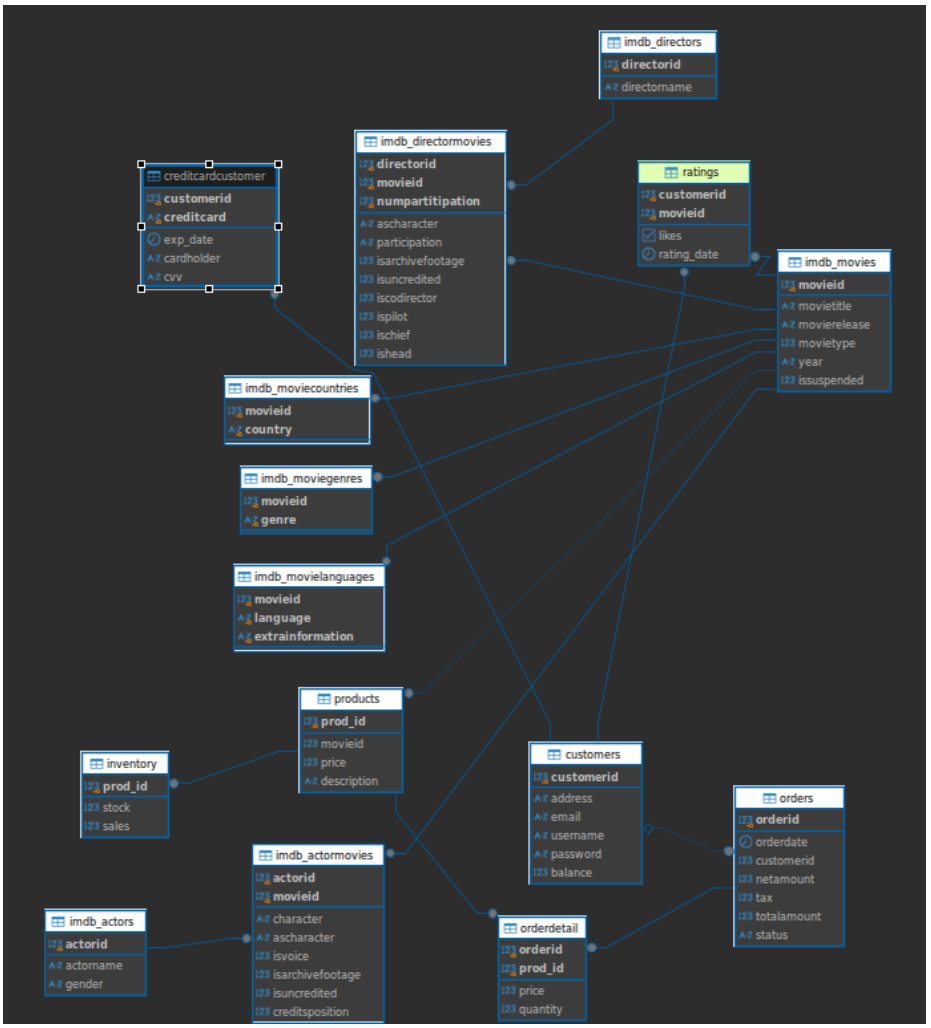
```
    constraint fk_customerid foreign key (customerid) references
customers(customerid)
);

insert into creditcardCustomer(customerid, creditcard)
select customerid, creditcard
from customers;



alter table customers
drop column creditcard;
```

(obtenido directamente desde nuestro código).

Este script crea una nueva tabla llamada creditcardCustomer para almacenar información sobre las tarjetas de crédito de los clientes, incluyendo campos como customerid, creditcard, exp_date, cardholder y cvv. Luego, migra los datos de la columna creditcard de la tabla customers a esta nueva tabla, y finalmente elimina la columna creditcard de customers. Este cambio organiza la información de las tarjetas de crédito de manera más adecuada y elimina un atributo multivaluado en la tabla customers.

De forma que con respecto al primer diagrama ER (1), obtenemos el siguiente diagrama actualizado con los cambios mencionados:

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024



(obtenido de dbeaver)(2).

Donde se puede observar que se crea correctamente la tabla ‘creditcardcostumer’ con lo especificado anteriormente.

- actualizaCarrito.sql, se trata del siguiente script:

```
create or replace function actualizaCarritoFunct()
returns trigger
as $$
begin

    if (tg_op = 'INSERT' or tg_op = 'UPDATE') then

        update orders o
```

```
        set netamount = (
            select SUM(od.price * od.quantity)
            from orderdetail od
            where od.orderid = o.orderid
        );
        UPDATE orders o
        SET totalamount = netamount + (netamount * (o.tax / 100));

    elsif (tg_op = 'DELETE') then

        update orders o
        set netamount = (
            select SUM(od.price * od.quantity)
            from orderdetail od
            where od.orderid = o.orderid
        );
        UPDATE orders o
        SET totalamount = netamount + (netamount * (o.tax / 100));
    end if;

    return null;
end;
$$ language plpgsql;

create or replace trigger actualizaCarrito
after insert or update or delete on orderdetail
for each row
execute function actualizaCarritoFunct();
```

(obtenido directamente del código).

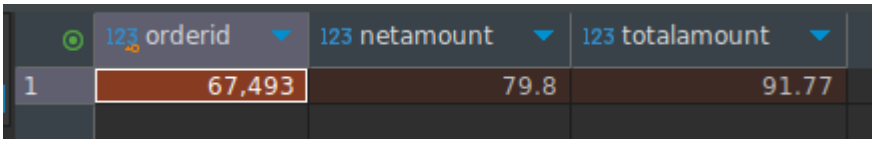
Este script crea una función y un trigger para actualizar automáticamente los totales de los pedidos en la tabla orders cuando se realicen cambios en la tabla orderdetail (añadir, actualizar o eliminar productos). La función recalcula los valores de netamount (total sin impuestos) y totalamount (total con impuestos) en la tabla orders cada vez que se realiza una operación sobre los productos del carrito, asegurando que los totales de los pedidos estén siempre actualizados. El trigger se activa después de

cualquier operación en orderdetail y ejecuta la función para realizar los cálculos correspondientes.

Para comprobar su correcto funcionamiento lo haremos directamente a través de querys sobre la base de datos (inserción, actualización y eliminación):

Actualización:

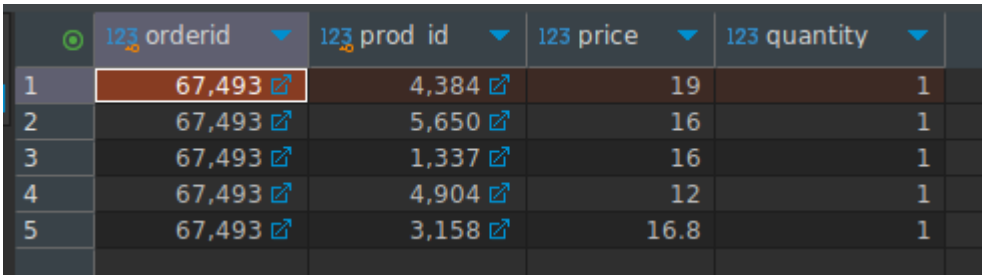
Por ejemplo, teniendo en cuenta el pedido con orederid de 67493:



	orderid	netamount	totalamount
1	67,493	79.8	91.77

(resultado de la query en dbeaver: SELECT orderid, netamount, totalamount FROM orders WHERE orderid = 67493;).

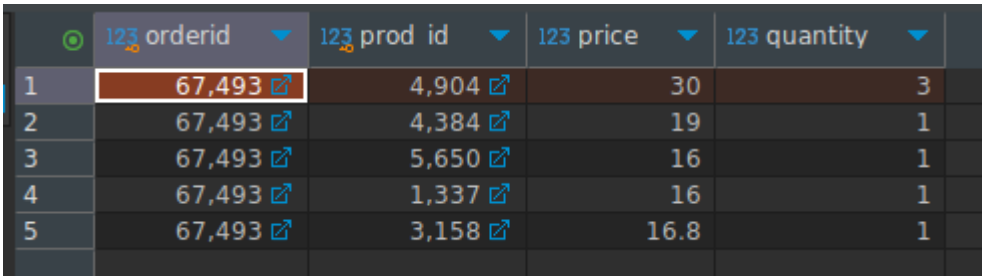
Si tenemos en cuenta que las cantidades y precios que hacen referencia a este pedido son las siguientes:



	orderid	prod_id	price	quantity
1	67,493	4,384	19	1
2	67,493	5,650	16	1
3	67,493	1,337	16	1
4	67,493	4,904	12	1
5	67,493	3,158	16.8	1

*(resultado de la query en dbeaver: SELECT * FROM orderdetail WHERE orderid = 67493;).*

Al modificarlo, por ejemplo con la siguiente query: UPDATE orderdetail
SET quantity = 3, price = 30 WHERE orderid = 67493 AND prod_id = 4904;
(dónde actualizamos el valor del producto 4904 del pedido 67493 de 12 a 30, y la cantidad de 1 a 3), obtenemos como resultado:

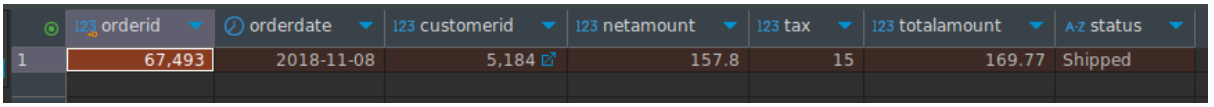


	orderid	prod_id	price	quantity
1	67,493	4,904	30	3
2	67,493	4,384	19	1
3	67,493	5,650	16	1
4	67,493	1,337	16	1
5	67,493	3,158	16.8	1

*(resultado de la query en dbeaver: SELECT * FROM orderdetail WHERE orderid = 67493;).*

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

De forma que con estos nuevos datos, el trigger del script actualizaCarrito.sql debería actualizar los campos netamount y totalamount en función de los nuevos datos. De forma que netamount = suma(quantity*price), es decir 157.8, mientras que totalamount = netamount + (netamount * (o.tax / 100)), (donde tax es 15 en este caso). Veámoslo:



	orderid	orderdate	customerid	netamount	tax	totalamount	status
1	67,493	2018-11-08	5,184	157.8	15	169.77	Shipped

*(resultado de la query en dbeaver: SELECT * FROM orders WHERE orderid = 67493;).*

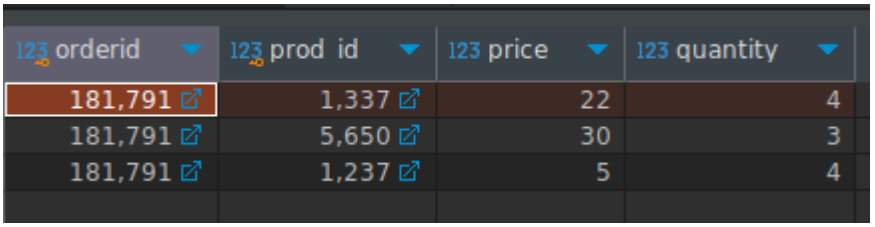
Como se puede observar, la actualización a través del trigger se cumple a la perfección.

Inserción:

Pongamos que nos disponemos a insertar un nuevo pedido: insert into orders o (orderdate, customerid, tax, status) values ('2024-11-12', 72, 15, Processed);

Ahora insertamos una serie de productos en el carrito para dicho pedido (id 181791), y deberíamos ver que se actualiza de forma correcta tanto el netamount como el totalamount:

Query de inserción: insert into orderdetail (orderid, prod_id, price, quantity)
values
(181791, 1337, 22, 4),
(181791, 5650, 30, 3),
(181791, 1237, 5, 4);

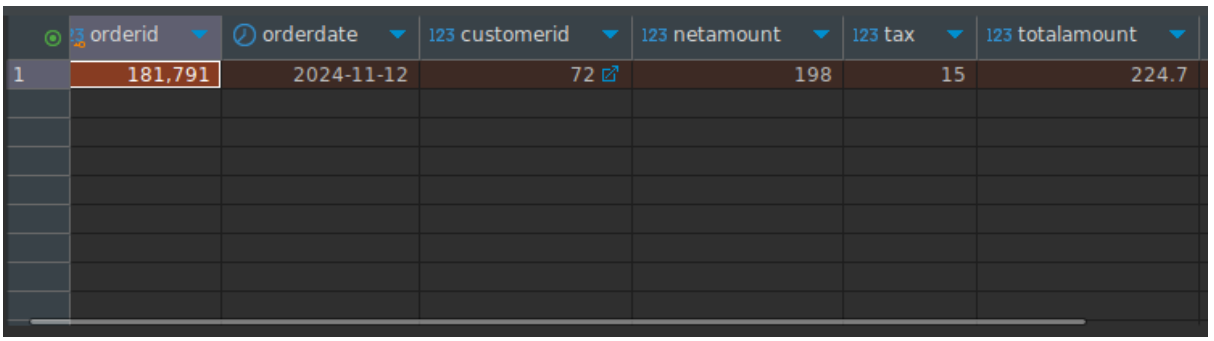


orderid	prod_id	price	quantity
181,791	1,337	22	4
181,791	5,650	30	3
181,791	1,237	5	4

(obetenido de la query anterior en dbeaver).

Ahora, al mirar la tabla de pedidos, deberíamos ver que se ha actualizado correctamente con los valores netamount = suma(quantity*price), es decir 198, mientras que totalamount = netamount + (netamount * (o.tax / 100)), es decir 224.7.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024



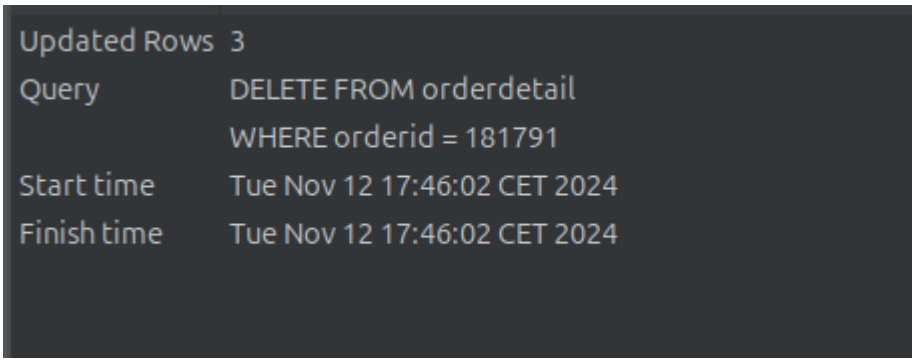
	orderid	orderdate	customerid	netamount	tax	totalamount
1	181,791	2024-11-12	72	198	15	224.7

(obtenido de dbeaver).

Como podemos ver, la actualización a partir de la inserción se hace correctamente.

Eliminación:

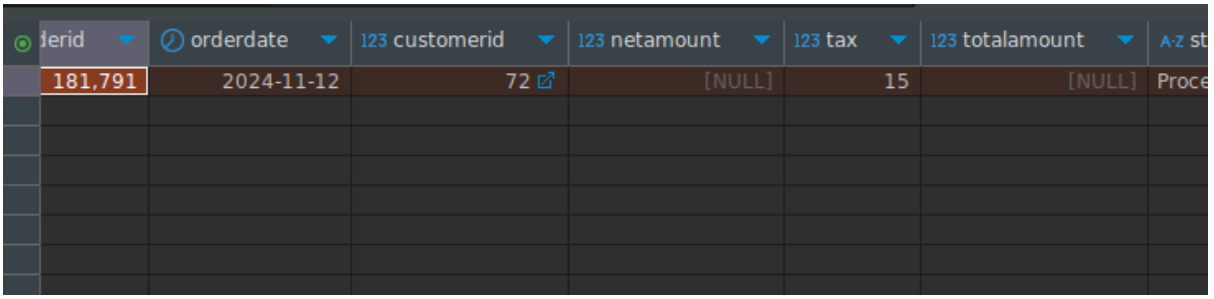
Ahora nos disponemos a eliminar un pedido de orderdetail:
DELETE FROM orders
WHERE orderid = 181791;



Updated Rows	3
Query	DELETE FROM orderdetail WHERE orderid = 181791
Start time	Tue Nov 12 17:46:02 CET 2024
Finish time	Tue Nov 12 17:46:02 CET 2024

(output dberaver de la consulta anterior).

Ahora, mirando la tabla de pedidos, debería haberse actualizado correctamente:



	derid	orderdate	customerid	netamount	tax	totalamount	st
	181,791	2024-11-12	72	[NULL]	15	[NULL]	Proce

(obtenido de dbeaver).

Como podemos observar funciona correctamente.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

- pagado.sql: el script es el siguiente:

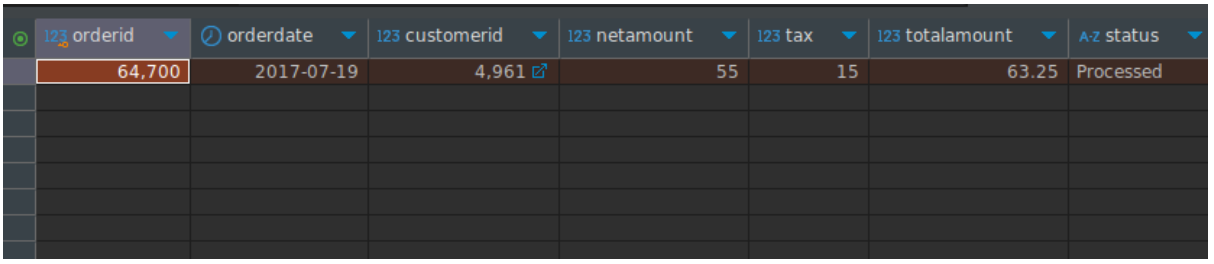
```
create or replace function pagadoFunct()
returns trigger
as $$
begin

   if (new.status = 'Paid') then

       update inventory i
       set
           stock = i.stock - od.quantity,
           sales = i.sales + od.quantity
       from orderdetail od
       where od.prod_id = i.prod_id and od.orderid = new.orderid;

   update customers c
   set balance = c.balance - o.totalamount
   from orders o
   where o.orderid = new.orderid and c.customerid = o.customerid;

   end if;

   return null;
end;
$$ language plpgsql;

create or replace trigger pagado
after update of status on orders
for each row
when (new.status = 'Paid')
execute function pagadoFunct();
```

(obtenido directamente desde nuestro código).

Este script crea una función de trigger llamada pagadoFunct() que se ejecuta automáticamente después de que se actualice el campo status en la tabla orders, pero solo cuando el nuevo estado es 'Paid'. Al activarse, la función realiza dos acciones: primero, actualiza la tabla inventory reduciendo el stock y aumentando las ventas de

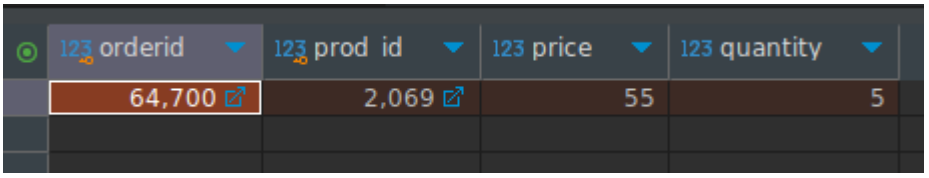
los productos correspondientes a los artículos del pedido que se ha marcado como pagado; y segundo, actualiza la tabla customers descontando el dinero total del pedido (totalamount) del saldo del cliente que realizó la compra.

Vamos por ejemplo a actualizar el status del pedido 67400:



orderid	orderdate	customerid	netamount	tax	totalamount	status
64,700	2017-07-19	4,961	55	15	63.25	Processed

(pedido en la tabla de pedidos).

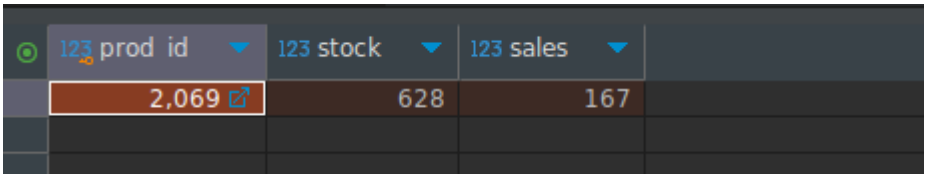


orderid	prod_id	price	quantity
64,700	2,069	55	5

(detalles del pedido).

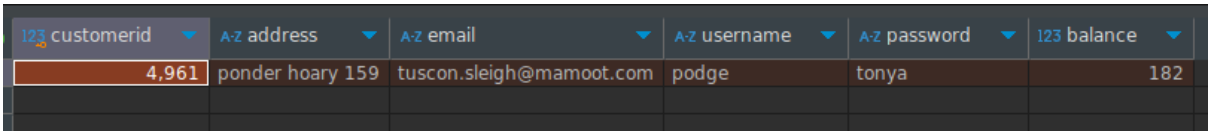
Con estos datos, si ponemos el pedido en paid, debería de actualizar la tabla inventory reduciendo el stock de los productos pagados, y aumentar las ventas de los productos correspondientes. Por último debe actualizar la tabla customers descontando el total del pedido al cliente que realizó la compra. Veámoslo:

Primero mostraremos los estados iniciales de dichas tablas:



prod_id	stock	sales
2,069	628	167

(estado de la tabla inventory para el producto en cuestión antes de cambiar el estado del pedido).



customerid	address	email	username	password	balance
4,961	ponder hoary 159	tuscon.sleigh@mamoot.com	podge	tonya	182

(tabla customers del cliente asociado al pedido 67400).

Veamos ahora el estado de las tablas tras actualizar el status del pedido a paid:

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

orderid	orderdate	customerid	netamount	tax	totalamount	status
64,700	2017-07-19	4,961	55	15	63.25	Paid

(status del pedido actualizado a paid).

prod_id	stock	sales
2,069	623	172

(tabla inventory actualizada).

Podemos ver que la tabla se ha actualizado correctamente, ya que aumentó las ventas del producto vendido.

customerid	address	email	username	password	balance
4,961	ponder hoary 159	tuscon.sleigh@mamoot.com	podge	tonya	118.75

(tabla customers actualizada).

De la misma forma, la tabla de clientes se ha actualizado de forma correcta, ya que disminuyó el saldo del cliente en consecuencia del pedido pagado. Por lo que el trigger funciona correctamente.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

Para trabajar con la base de datos utilizamos una api python que utiliza Quart y SQLAlchemy para poder hacer de intermediario entre cliente y BBDD y cumplir la función de procesamiento en caso de ser necesario.

Las funciones implementadas son las siguientes:

Register:

- **ruta:** host:port/register
- **input:**
  - email
  - password
  - address
  - username
- **resumen:**

  Esta función verifica si existe el email en la base de datos y si no existe crea un nuevo customer asignando automáticamente un customerid a éste
- **métodos:**
  - POST
- **código:**

```python
@app.route('/register', methods = ['POST'])
async def register():
    datos = await request.get_json()
    email = datos.get("email")
    pwd = datos.get("password")
    username = datos.get("username")
    address = datos.get("address")

    if not email or not pwd or not username or not address:
        return jsonify({"error": "Missing some data"}), 401

    session = Session()
    try:
        user = session.query(Customer).filter_by(email = email).first()

        if user:
            return jsonify({'message' : "Email already in use"}), 401

        maxid = session.query(func.max(Customer.customerid)).scalar()

        new_user = Customer(customerid = maxid+1, email = email, password = pwd,
username = username, address = address)
        session.add(new_user)
        session.commit()

        return jsonify({'message' : "User registered successfully"}), 20

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

Login:

- **ruta:** host:port/login
- **input:**
  - email
  - password
- **resumen:**

  Esta función verifica si existe la combinación (email, password) en la base de datos, devuelve OK (code:200) en caso positivo
- **métodos:**
  - POST
- **código:**

```
@app.route('/login', methods = ['POST'])
async def login():
   datos = await request.get_json()
   email = datos.get("email")
   pwd = datos.get("password")

   if not email or not pwd:
       return jsonify({"error": "Missing email or password"}), 400

   session = Session()
   try:
       user = session.query(Customer).filter_by(email = email, password =
pwd).first()

       if user:
           return jsonify({'message' : "Login Successful", 'customerid' :
user.customerid}), 200
       else:
           return jsonify({'message' : "Invalid username or password"}), 401
   except SQLAlchemyError as e:
       return jsonify({"error": str(e)}), 500
   finally:
       session.close()
```

Add Creditcard:

- **ruta:** host:port/add_creditcard
- **input:**
  - email
  - password
  - creditcard
  - expiration date
  - cvv
  - cardholder
- **resumen:**

  Esta función verifica la integridad de los datos y si existe la combinación (email, password) en la base de datos, en caso positivo saca el customerid y busca una combinación (customerid, creditcard) en creditcardCustomer, en caso negativo añade la tarjeta de crédito a la base de datos
- **métodos:**

    - POST
- **código:**

```
@app.route('/add_creditcard', methods = ['POST'])
async def add_creditcard():
    datos = await request.get_json()
    email = datos.get("email")
    pwd = datos.get("password")
    creditcard = datos.get("creditcard")
    exp_date_str = datos.get("exp_date")
    cvv = datos.get("cvv")
    cardholder = datos.get("cardholder")

    try:
        exp_date = datetime.strptime(exp_date_str, '%Y-%m-%d').date()
    except ValueError:
        return jsonify({"error": "Invalid date format"}), 400

    if not email or not pwd or not creditcard or not exp_date or not cvv or not cardholder:
        return jsonify({"error": "Missing some data"}), 401

    session = Session()
    try:
        user = session.query(Customer).filter_by(email = email, password = pwd).first()

        if not user:
            return jsonify({'message' : "Invalid email or password"}), 401

        creditcard_user = session.query(CreditcardCustomer).filter_by(customerid =
user.customerid, creditcard=creditcard).first()

        if creditcard_user:
            return jsonify({'message' : "Credit card already registered by the user"}), 401

        new_creditcard = CreditcardCustomer(customerid = user.customerid, creditcard =
creditcard, exp_date = exp_date, cvv = cvv, cardholder = cardholder)
        session.add(new_creditcard)
        session.commit()

        return jsonify({'message' : "Credit card added successfully"}), 200

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

List Creditcards:

- **ruta:** host:port/list_creditcards
- **input:**
    - email
    - password
- **resumen:**

  Esta función verifica si existe la combinación (email, password) en la base de datos, en caso positivo saca todas las ocurrencias de creditcard(creditcard, customerid)

- **métodos:**
    - POST
- **código:**

```
@app.route('/list_creditcards', methods = ['POST'])
async def list_creditcards():
   datos = await request.get_json()
   email = datos.get("email")
   pwd = datos.get("password")

   if not email or not pwd:
       return jsonify({"error": "Missing some data"}), 401

   session = Session()
   try:
       user = session.query(Customer).filter_by(email = email, password = pwd).first()

       if not user:
           return jsonify({'message' : "Invalid email or password"}), 401

       creditcards_user = session.query(CreditcardCustomer.creditcard).filter_by(customerid =
user.customerid).all() #Lista de tuplas
       creditcards_list = [creditcard[0] for creditcard in creditcards_user] #Primer elemento de
cada tupla

       if not creditcards_user:
           return jsonify({'message' : "No creditcards registered by the user"}), 401

       session.commit()

       return jsonify({'message' : creditcards_list}), 200

   except SQLAlchemyError as e:
       return jsonify({"error": str(e)}), 500
   finally:
       session.close()
```

Delete Creditcard:

- **ruta:** host:port/delete_creditcard
- **input:**
    - email
    - password
    - creditcard
- **resumen:**

  Esta función verifica si existe la combinación (email, password) en la base de datos, en caso positivo verifica si existe la combinación (customerid, creditcard) en la tabla creditcardCustomer y borra dicha entrada
- **métodos:**
    - POST
- **código:**

```
@app.route('/delete_creditcard', methods = ['POST'])
async def delete_creditcard():
```

```
    datos = await request.get_json()
    email = datos.get("email")
    pwd = datos.get("password")
    creditcard = datos.get("creditcard")

    if not email or not pwd or not creditcard:
        return jsonify({"error": "Missing some data"}), 401

    session = Session()
    try:
        user = session.query(Customer).filter_by(email = email, password =
pwd).first()

        if not user:
            return jsonify({'message' : "Invalid email or password"}), 401

        creditcard_user = session.query(CreditcardCustomer).filter_by(customerid =
user.customerid, creditcard=creditcard).first()

        if not creditcard_user:
            return jsonify({'message' : "Invalid creditcard"}), 401

        session.delete(creditcard_user)
        session.commit()

        return jsonify({'message' : "Credit card deleted successfully"}), 200

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

Add Balance:

- **ruta:** host:port/add_balance
- **input:**
    - email
    - password
    - creditcard
    - balance
- **resumen:**

  Esta función verifica la integridad de los datos y si existe la combinación (email, password) en la base de datos, en caso positivo verifica si existe la combinación (customerid, creditcard) en la tabla creditcardCustomer y añade el saldo dado a la entrada correspondiente en la tabla Customers
- **métodos:**
    - POST
- **código:**

```
@app.route('/add_balance', methods = ['POST'])
async def add_balance():
    datos = await request.get_json()
    email = datos.get("email")
    pwd = datos.get("password")
    balance = datos.get("balance")
    creditcard = datos.get("creditcard")
```

```
    try:
        intBalance = int(balance)
    except ValueError:
        return jsonify({"error": "Balance must be a number"}), 400

    if intBalance < 0:
        return jsonify({"error": "Balance can't be negative"}), 400

    if not email or not pwd or not balance or not creditcard:
        return jsonify({"error": "Missing some data"}), 401

    session = Session()
    try:
        user = session.query(Customer).filter_by(email = email, password =
pwd).first()

        if not user:
            return jsonify({'message' : "Invalid email or password"}), 401

        creditcard_user = session.query(CreditcardCustomer).filter_by(customerid =
user.customerid, creditcard=creditcard).first()

        if not creditcard_user:
            return jsonify({'message' : "Invalid credit card"}), 401

        user.balance += intBalance
        session.commit()

        return {'message' : "Balance added successfully", 'balance' : user.balance},
200

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

Add to Cart:

- **ruta:** host:port/add_to_cart
- **input:**
    - email
    - password
    - product ID
    - quantity
- **resumen:**

  Esta función verifica la integridad de los datos y si existe la combinación (email, password) en la base de datos, en caso positivo verifica si existe alguna entrada en Products con (prod_id=product ID) y si hay stock suficiente para la cantidad seleccionada.
  Después comprueba si existe alguna entrada en la tabla Orders con (customerid = customerid(email, password), status = ‘Processed’), en caso negativo lo crea.

Comprueba si existe alguna entrada en OrderDetails con (prod_id = product ID), en caso negativo la añade y los triggers de la BBDD se encargan de actualizar automáticamente, en caso positivo modifica el campo *quantity* de dicha entrada y los triggers se encargan del resto.

- **métodos:**
  - POST
- **código:**

```
@app.route('/add_to_cart', methods = ['POST'])
async def add_to_cart():
   datos = await request.get_json()
   email = datos.get("email")
   pwd = datos.get("password")
   productid = datos.get("productid")
   quantity_str = datos.get("quantity")


   try:
       quantity = int(quantity_str)
   except ValueError:
       return jsonify({"error": "Quantity must be a number"}), 400


   if not email or not pwd or not productid or not quantity:
       return jsonify({"error": "Missing some data"}), 401

   session = Session()
   try:
       user = session.query(Customer).filter_by(email = email, password = pwd).first()

       if not user:
           return jsonify({'message' : "Invalid email or password"}), 401

       product = session.query(Product).filter_by(prod_id = productid).first()

       if not product:
           return jsonify({'message' : "Invalid product id"}), 401

       inventory = session.query(Inventory).filter_by(prod_id = productid).first()

       if not inventory or inventory.stock < quantity:
           return jsonify({'message' : "Not enough stock for the quantity required"}), 401

       order = session.query(Order).filter_by(customerid = user.customerid, status =
'Processed').first()

       if not order:
           maxid = session.query(func.max(Order.orderid)).scalar()
           order = Order(orderid = maxid+1, customerid = user.customerid, orderdate =
datetime.now(), tax = 15, status = 'Processed')
           session.add(order)

       orderDetail = session.query(OrderDetail).filter_by(orderid = order.orderid, prod_id =
productid).first()

       if not orderDetail:
```

```
            newOrderDetail = OrderDetail(orderid = order.orderid, prod_id = productid,
quantity = quantity, price = product.price)
            session.add(newOrderDetail)
        else:
            orderDetail.quantity += quantity


        session.commit()

        return jsonify({'message' : "Product added to cart successfully"}), 200

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

Delete from Cart:

- **ruta:** host:port/delete_from_cart
- **input:**
    - email
    - password
    - product ID
    - quantity
- **resumen:**

  Esta función verifica la integridad de los datos y si existe la combinación (email, password) en la base de datos.
  Después comprueba si existe alguna entrada en la tabla Orders con (customerid = customerid(email, password), status = 'Processed'), en caso negativo devuelve error.
  Tras esto comprueba si existe alguna entrada en OrderDetails con (prod_id = product ID), en caso negativo devuelve error, por otro lado, en caso positivo modifica el campo *quantity* de dicha entrada si hay suficiente para restar y en caso de que *quantity* resulte ser 0 después de la modificación, borra la entrada correspondiente en OrderDetails. Los triggers de la BBDD se encargan de actualizar la tabla Orders.
- **métodos:**
    - POST
- **código:**

```
@app.route('/delete_from_cart', methods = ['POST'])
async def delete_from_cart():
    datos = await request.get_json()
    email = datos.get("email")
    pwd = datos.get("password")
    productid = datos.get("productid")
    quantity_str = datos.get("quantity")

    try:
        quantity = int(quantity_str)
    except ValueError:
        return jsonify({"error": "Quantity must be a number"}), 400
```

```
    if not email or not pwd or not productid or not quantity:
        return jsonify({"error": "Missing some data"}), 401

    session = Session()
    try:
        user = session.query(Customer).filter_by(email = email, password = pwd).first()

        if not user:
            return jsonify({'message' : "Invalid email or password"}), 401

        order = session.query(Order).filter_by(customerid = user.customerid, status =
'Processed', ).first()

        if not order:
            return jsonify({'message' : "No cart for the user"}), 401

        orderDetail = session.query(OrderDetail).filter_by(orderid = order.orderid, prod_id =
productid).first()

        if not orderDetail:
            return jsonify({'message' : "Product not in the cart"}), 401
        else:
            orderDetail.quantity -= quantity
            if orderDetail.quantity == 0:
                session.delete(orderDetail)

        session.commit()

        return jsonify({'message' : "Product deleted from the cart successfully"}), 200

    except SQLAlchemyError as e:
        return jsonify({"error": str(e)}), 500
    finally:
        session.close()
```

Pay Cart:

- **ruta:** host:port/pay _cart
- **input:**
    - email
    - password
    - product ID
    - quantity
- **resumen:**

    Esta función verifica si existe la combinación (email, password) en la base de datos.
    Después comprueba si existe alguna entrada en la tabla Orders con (customerid = customerid(email, password), status = ‘Processed’), en caso negativo devuelve error.
    Tras esto comprueba si el customer de la consulta anterior posee suficiente saldo en su cuenta para completar la transacción, en caso positivo modifica el estado de la entrada correspondiente en la tabla Orders a ‘Paid’.

Los triggers de la BBDD se encargan automáticamente de restar el saldo a la cuenta del customer.

- **métodos:**
  - POST

- **código:**

```
@app.route('/pay_cart', methods = ['POST'])
async def pay_cart():
   datos = await request.get_json()
   email = datos.get("email")
   pwd = datos.get("password")
   creditcard = datos.get("creditcard")

   if not email or not pwd:
       return jsonify({"error": "Missing some data"}), 401

   session = Session()
   try:
       user = session.query(Customer).filter_by(email = email, password =
pwd).first()

       if not user:
           return jsonify({'message' : "Invalid email or password"}), 401

       creditcard_user = session.query(CreditcardCustomer).filter_by(customerid =
user.customerid, creditcard=creditcard).first()
       if not creditcard_user:
           return jsonify({'message' : "Invalid credit card"}), 401

       order = session.query(Order).filter_by(customerid = user.customerid, status =
'Processed').first()

       if not order:
           return jsonify({'message' : "No cart for the user"}), 401

       if user.balance < order.totalamount:
           return jsonify({'message' : "Insufficient balance"}), 401

       order.status = 'Paid'
       session.commit()

       return jsonify({'message' : "Cart paid successfully", 'balance' :
user.balance}), 200

   except SQLAlchemyError as e:
       return jsonify({"error": str(e)}), 500
   finally:
       session.close()
```

OPTIMIZACIÓN:

a.) La consulta pedida es la siguiente:

```
-- Estudio del plan de ejecución de la consulta
EXPLAIN
SELECT COUNT(DISTINCT state) AS num_estados
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE o.orderdate >= '2017-01-01' AND o.orderdate < '2018-01-01'
AND c.country = 'Spain';
```

(obtenido directamente de nuestro código).

b.) Al añadir la sentencia EXPLAIN, y usando por ejemplo como país determinado "Spain", obtenemos el siguiente output:

```
                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------
 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: ((orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
               ->  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 = 'Spain'::text)
(10 rows)
```

(output obtenido directamente desde terminal).

El plan de ejecución proporcionado muestra cómo PostgreSQL planea ejecutar la consulta para contar el número de estados distintos de los clientes en España que realizaron pedidos en el año 2017. Aquí está la interpretación de cada parte del plan:

1. Aggregate (cost=4821.98..4821.99 rows=1 width=8)
- Esta operación realiza la agregación final para contar los estados distintos.
- El costo estimado de esta operación es 4821.98.

2. Gather (cost=1529.04..4821.97 rows=5 width=118)
- Esta operación reúne los resultados de los trabajadores paralelos.
- Se planea utilizar 1 trabajador paralelo.

3. Hash Join (cost=529.04..3821.47 rows=3 width=118)
- Esta operación realiza una unión hash entre las tablas \`orders\` y \`customers\`.
- La condición de hash es que \`o.customerid\` debe ser igual a \`c.customerid\`.

4. Parallel Seq Scan on orders o (cost=0.00..3291.03 rows=535 width=4)
- Esta operación realiza un escaneo secuencial paralelo en la tabla \`orders\`.

- Se aplica un filtro para seleccionar los pedidos realizados entre el 1 de enero de 2017 y el 31 de diciembre de 2017.

5. Hash (cost=528.16..528.16 rows=70 width=122)
- Esta operación crea una tabla hash a partir de los resultados del escaneo secuencial en la tabla `customers`.

6. Seq Scan on customers c (cost=0.00..528.16 rows=70 width=122)
- Esta operación realiza un escaneo secuencial en la tabla `customers`.
- Se aplica un filtro para seleccionar los clientes cuyo país es España.

c.) Para mejorar el rendimiento de la consulta, podemos crear índices en las columnas que se utilizan en las condiciones de unión y filtrado. En este caso, las columnas son: customerid (en ambas tablas), orderdate (en la tabla orders), y country (en la tabla customers).

Primero, eliminamos cualquier índice existente en estas columnas (si existen), y luego creamos los nuevos índices.

```
-- Eliminar índices existentes (si existen)
DROP INDEX IF EXISTS idx_orders_customerid;
DROP INDEX IF EXISTS idx_orders_orderdate;
DROP INDEX IF EXISTS idx_customers_customerid;
DROP INDEX IF EXISTS idx_customers_country;

-- Crear nuevos índices
CREATE INDEX idx_orders_customerid ON orders (customerid);
CREATE INDEX idx_orders_orderdate ON orders (orderdate);
CREATE INDEX idx_customers_customerid ON customers (customerid);
CREATE INDEX idx_customers_country ON customers (country);
```

(obtenido directamente de nuestro código).

d.) Con estos cambios, veamos si ha surgido alguna mejora en la ejecución de la consulta anterior. Ejecutando de nuevo la query con la sentencia EXPLAIN, obtenemos el siguiente output:

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

```
si2=# \i estadosDistintos.sql
DROP INDEX
DROP INDEX
DROP INDEX
DROP INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1676.05..1676.06 rows=1 width=8)
   ->  Hash Join  (cost=194.32..1676.04 rows=5 width=118)
         Hash Cond: (o.customerid = c.customerid)
         ->  Bitmap Heap Scan on orders o  (cost=13.61..1492.94 rows=909 width=4)
               Recheck Cond: ((orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
               ->  Bitmap Index Scan on idx_orders_orderdate  (cost=0.00..13.38 rows=909 width=0)
                     Index Cond: ((orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
         ->  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 = 'Spain'::text)
                     ->  Bitmap Index Scan on idx_customers_country  (cost=0.00..4.81 rows=70 width=0)
                           Index Cond: ((country)::text = 'Spain'::text)
(12 rows)
```

(output obtenido directamente desde terminal).

De forma resumida podemos ver que:

-Aggregate: Realiza la agregación final para contar los estados distintos.

-Hash Join: Realiza una unión hash entre las tablas orders y customers.

-Bitmap Heap Scan on orders: Utiliza un escaneo de heap con un índice de mapa de bits en la tabla orders.

-Bitmap Index Scan on idx_orders_orderdate: Utiliza el índice idx_orders_orderdate para encontrar las filas relevantes en la tabla orders.

-Hash: Crea una tabla hash a partir de los resultados del escaneo de heap en la tabla customers.

-Bitmap Heap Scan on customers: Utiliza un escaneo de heap con un índice de mapa de bits en la tabla customers.

-Bitmap Index Scan on idx_customers_country: Utiliza el índice idx_customers_country para encontrar las filas relevantes en la tabla customers.

Este plan de ejecución muestra que los índices están siendo utilizados de manera efectiva, lo que ha reducido el costo total de la consulta y mejorado el rendimiento. Pasando de un coste aproximado de 4821.98 a 1676.05 en este caso.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

e.) Los distintos índices a probar serán índices individuales y compuestos. Es decir índices que se crean en una sola columna de una tabla (individuales). Los cuales, son útiles para mejorar el rendimiento de las consultas que filtran, ordenan o buscan datos basados en una sola columna. Y los compuestos, que se crean en dos o más columnas de una tabla. Estos últimos, son útiles para mejorar el rendimiento de las consultas que filtran, ordenan o buscan datos basados en múltiples columnas.

Sin ejecutar nada, podemos más o menos predecir que en nuestro caso, los compuestos serán más eficientes en nuestra base de datos, ya que se ordenan y buscan datos basándose en varias columnas mayoritariamente. Veámoslo, éste es el script actualizado para que nos ofrezca el output para comparar los dos tipos de índices:

```
-- Eliminar índices existentes (si existen)
DROP INDEX IF EXISTS idx_orders_customerid;
DROP INDEX IF EXISTS idx_orders_orderdate;
DROP INDEX IF EXISTS idx_customers_customerid;
DROP INDEX IF EXISTS idx_customers_country;
DROP INDEX IF EXISTS idx_orders_customerid_orderdate;
DROP INDEX IF EXISTS idx_customers_customerid_country;

-- Crear nuevos índices individuales necesarios
CREATE INDEX idx_orders_customerid ON orders (customerid);
CREATE INDEX idx_orders_orderdate ON orders (orderdate);
CREATE INDEX idx_customers_customerid ON customers (customerid);
CREATE INDEX idx_customers_country ON customers (country);

-- Estudio del plan de ejecución de la consulta con índices
individuales
EXPLAIN
SELECT COUNT(DISTINCT state) AS num_estados
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE o.orderdate >= '2017-01-01' AND o.orderdate < '2018-01-01'
AND c.country = 'Spain';

-- Eliminar índices individuales
DROP INDEX IF EXISTS idx_orders_customerid;
DROP INDEX IF EXISTS idx_orders_orderdate;
DROP INDEX IF EXISTS idx_customers_customerid;
DROP INDEX IF EXISTS idx_customers_country;

-- Crear nuevos índices compuestos
```

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

```sql
CREATE INDEX idx_orders_customerid_orderdate ON orders
(customerid, orderdate);
CREATE INDEX idx_customers_customerid_country ON customers
(customerid, country);

-- Estudio del plan de ejecución de la consulta con índices
compuestos
EXPLAIN
SELECT COUNT(DISTINCT state) AS num_estados
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE o.orderdate >= '2017-01-01' AND o.orderdate < '2018-01-01'
AND c.country = 'Spain';
```

(obtenido directamente desde nuestro código).

El output es el siguiente:

```
DROP INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1676.05..1676.06 rows=1 width=8)
   ->  Hash Join  (cost=194.32..1676.04 rows=5 width=118)
         Hash Cond: (o.customerid = c.customerid)
         ->  Bitmap Heap Scan on orders o  (cost=13.61..1492.94 rows=909 width=4)
               Recheck Cond: ((orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
               ->  Bitmap Index Scan on idx_orders_orderdate  (cost=0.00..13.38 rows=909 width=0)
                     Index Cond: ((orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
         ->  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 = 'Spain'::text)
                     ->  Bitmap Index Scan on idx_customers_country  (cost=0.00..4.81 rows=70 width=0)
                           Index Cond: ((country)::text = 'Spain'::text)
(12 rows)

DROP INDEX
DROP INDEX
DROP INDEX
DROP INDEX
CREATE INDEX
CREATE INDEX
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=821.79..821.80 rows=1 width=8)
   ->  Nested Loop  (cost=342.42..821.78 rows=5 width=118)
         ->  Bitmap Heap Scan on customers c  (cost=342.00..517.01 rows=70 width=122)
               Recheck Cond: ((country)::text = 'Spain'::text)
               ->  Bitmap Index Scan on idx_customers_customerid_country  (cost=0.00..341.98 rows=70 width=0)
                     Index Cond: ((country)::text = 'Spain'::text)
         ->  Index Only Scan using idx_orders_customerid_orderdate on orders o  (cost=0.42..4.30 rows=5 width=4)
               Index Cond: ((customerid = c.customerid) AND (orderdate >= '2017-01-01'::date) AND (orderdate < '2018-01-01'::date))
(8 rows)
```

(output de terminal del anterior script).

Como se había mencionado, se puede ver que al usar índices compuestos se obtiene un mejor coste, 821.80 frente a 1676.8. En conclusión, la consulta obtiene un mejor rendimiento utilizando índices compuestos porque el planificador de consultas puede acceder a los datos de manera más eficiente (ya que se maneja usando varias columnas en la búsqueda o filtrado de datos), reduciendo el costo total de la consulta y mejorando el tiempo de respuesta.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

ANEXO1:

A. Estudiar los planes de ejecución de las consultas alternativas mostradas en el Anexo 1 y compararlos.
   a. ¿Qué consulta devuelve algún resultado nada más comenzar su ejecución?
   b. ¿Qué consulta se puede beneficiar de la ejecución en paralelo?

Primero analicemos resumidamente las consultas:

- Todas las consultas hacen lo mismo en conjunto, seleccionar los customerid de los clientes que no tienen pedidos con el estado 'Paid'. Sin embargo, utilizan diferentes enfoques para lograrlo.

1. Por un lado tenemos la consulta que usa NOT IN. La cual necesita completar la subconsulta antes de comenzar a devolver resultados, ya que compara cada customerid con el conjunto completo de customerid devuelto por la subconsulta.

2. Luego tenemos la consulta que usa EXCEPT. La cual necesita completar ambas subconsultas antes de devolver resultados, ya que debe calcular la diferencia entre ambos subconjuntos.

3. Por último, tenemos la consulta que usa UNION ALL. Esta, puede devolver resultados tan pronto como se procesan las primeras filas de cada subconsulta, ya que la agrupación GROUP BY permite que los resultados se acumulen y se devuelvan a medida que se procesan.

-De forma que, la consulta que devuelve algún resultado nada más comenzar su ejecución es la que usa UNION ALL.

-En este caso, la tercera consulta, construida con EXCEPT, es la más adecuada para beneficiarse de la paralelización. Esta consulta funciona comparando dos conjuntos de resultados: los customerid de customers y los customerid de orders con status='Paid'. Como ambas tablas pueden escanearse independientemente antes de comparar resultados, esta estructura se adapta bien a la ejecución en paralelo, ya que permite que cada subconsulta (customers y orders) se escanee de forma concurrente.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

ANEXO 2:

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

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

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

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

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

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

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

c.) el output a la sentencia EXPLAIN de las respectivas querys es:

```
si2=# \i countStatus.sql
                          QUERY PLAN
---------------------------------------------------------------
 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)
(3 rows)

 count
--------
 127323
(1 row)

                          QUERY PLAN
---------------------------------------------------------------
 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 = 'Paid'::text)
(3 rows)

 count
-------
 36304
(1 row)
```

(output de terminal).

Podemos ver que ambas consultas son muy parecidas, ambas consultas realizan un escaneo secuencial de la tabla orders y aplican un filtro para contar las filas que cumplen con la

condición especificada (status IS NULL y status = 'Paid'). Podemos ver que la primera consulta es ligeramente menos costosa.

d.) el índice pedido:

```
-- Crea un índice en la columna 'status' de la tabla 'orders'
CREATE INDEX idx_orders_status ON orders (status);

-- Estudio del plan de ejecución de la consulta con índice en la
columna 'status'
EXPLAIN
select count(*)
from orders
where status is null;
select count(*)
from orders
where status ='Shipped';

-- Estudio del plan de ejecución de la segunda consulta con índice en
la columna 'status'
EXPLAIN
select count(*)
from orders
where status ='Paid';
select count(*)
from orders
where status ='Processed';
```

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

e.) El nuevo output es el siguiente:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=22.48..22.49 rows=1 width=8)
   ->  Index Only Scan using idx_orders_status on orders  (cost=0.29..20.20 rows=909 width=0)
         Index Cond: (status IS NULL)
(3 rows)

 count
--------
 127323
(1 row)

                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=22.48..22.49 rows=1 width=8)
   ->  Index Only Scan using idx_orders_status on orders  (cost=0.29..20.20 rows=909 width=0)
         Index Cond: (status = 'Paid'::text)
(3 rows)

 count
-------
 36304
(1 row)
```

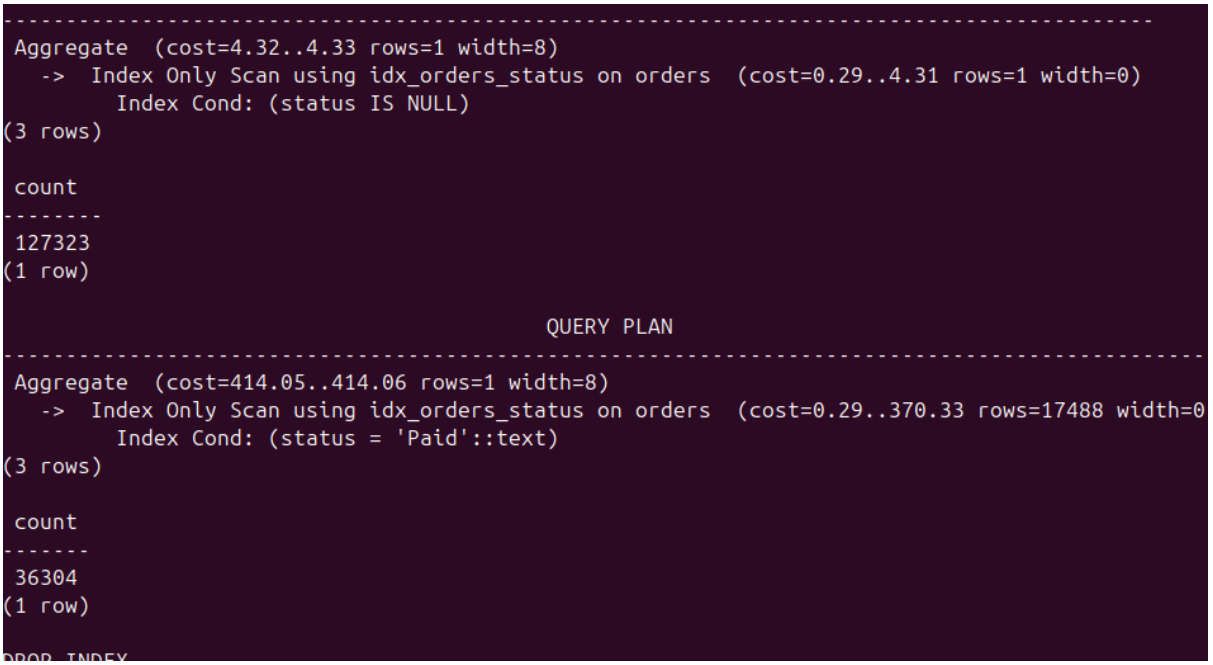
(output de terminal).

Podemos observar que la creación del índice idx_orders_status ha reducido significativamente los costos estimados de las consultas, pasando de más de 3500 a aproximadamente 22. Esto se debe a que el índice permite un acceso más rápido y eficiente a las filas que cumplen con las condiciones de filtro, evitando la necesidad de un escaneo secuencial completo de la tabla.

f,g,h,i.) Lo primero de todo es explicar qué ha hecho la sentencia ANALYZE (la cual hemos ejecutado como: ANALYZE orders;). La sentencia ANALYZE ha recolectado estadísticas detalladas sobre la distribución de los datos en la tabla orders, incluyendo la columna status. Estas estadísticas se almacenan en el catálogo del sistema de la base de datos y son utilizadas por el optimizador de consultas para generar planes de ejecución más eficientes. En resumen, ANALYZE ha permitido al optimizador de consultas tener una mejor comprensión de cuántas filas cumplen con ciertas condiciones (como campos a NULL), lo que influye en la elección de los planes de ejecución.

Nombre: Luis Núñez y Javier Agüero
Pareja: 6
Fecha: 12/11/2024

De esta forma, aquí tenemos el nuevo output tras ejecutar la sentencia:



(output de terminal).

En cuanto a la primera consulta, el costo total estimado es bajo (4.32), lo que indica que el índice es muy eficiente para esta consulta. Esto se debe a que el optimizador ahora sabe que hay relativamente pocas filas con status IS NULL, permitiendo un acceso rápido a través del índice.

En cuanto a la segunda, el costo total estimado es mayor (414.05) en comparación con la primera consulta. Esto se debe a que hay muchas más filas con status = 'Paid', lo que significa que el índice tiene que procesar más datos. Aunque el costo es mayor, el uso del índice sigue siendo más eficiente que un escaneo secuencial completo de la tabla.

En conclusión, ANALYZE ha permitido al optimizador de consultas generar planes de ejecución más precisos y eficientes (sobre cómo usar el índice creado). La primera consulta tiene un costo muy bajo debido al bajo uso del filtro status IS NULL, mientras que la segunda consulta tiene un costo mayor debido al alto uso del filtro status = 'Paid'. En ambos casos, el uso del índice mejora la eficiencia en comparación con un escaneo secuencial completo de la tabla.