

Sistemas informáticos Práctica 2

(La Semana de Javier)

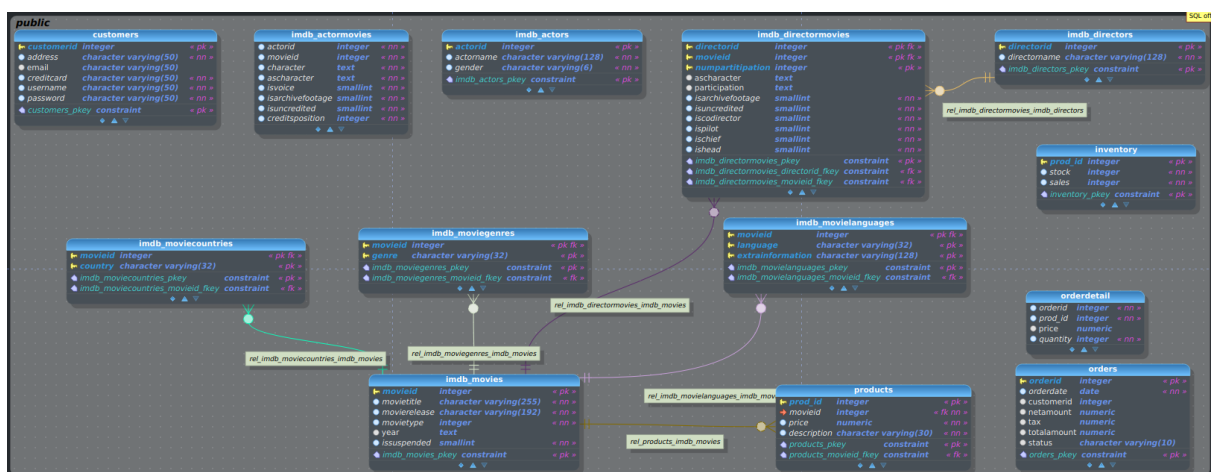
Ignacio Serena y

Marcos Muñoz

Diseño de la BBDD

Apartado A

Para obtener el esquema E-R de la base de datos proporcionada hemos decidido utilizar la herramienta pgmodeler la cual permite efectuar el proceso de ingeniería inversa. Este es el resultado obtenido:



Para actualizar la base de datos con los requisitos solicitados se ha creado un fichero actualiza.sql con el siguiente contenido:

```
-- Añade columna 'balance' a la tabla 'customers' para el sal
ALTER TABLE customers
ADD COLUMN balance NUMERIC(10, 2) DEFAULT 0;

-- Crea tabla 'ratings' para almacenar likes de los usuarios
```

```

CREATE TABLE ratings (
    rating_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES customers(customerid) ON DELETE CASCADE,
    product_id INT REFERENCES products(prod_id) ON DELETE CASCADE,
    is_liked BOOLEAN NOT NULL DEFAULT FALSE,
    UNIQUE (user_id, product_id)
);

-- Modifica la longitud del campo 'password' en la tabla 'customers'
ALTER TABLE customers
ALTER COLUMN password TYPE character varying(100);

-- Procedimiento para asignar saldo aleatorio entre 0 y N
CREATE OR REPLACE FUNCTION setCustomersBalance(IN initialBalance INT)
RETURNS VOID AS $$
BEGIN
    UPDATE customers
    SET balance = floor(random() * (initialBalance + 1))::NUMERIC;
END;
$$ LANGUAGE plpgsql;

-- Llamada al procedimiento con N = 200
SELECT setCustomersBalance(200);

\i actualizaPrecios.sql
\i actualizaTablas.sql
\i actualizaCarrito.sql
\i pagado.sql
\i actualizaPreciosPedidos.sql -- Archivo extra ( no solicitado )
;

```

Antes de ver las ejecuciones realizadas, si usted desea comprobarlas en su maquina personal, puede hacerlo sin problema. Inicie la base de datos

ejecutando el siguiente comando en el directorio donde se encuentre el archivo

```
docker-compose.yml :
```

```
sudo docker-compose up
```

Comprobación de la actualización

Para realizar la comprobación de este apartado vamos a comparar los resultados de las consultas de prueba antes y después de la ejecución de

```
actualiza.sql .
```

Antes de la ejecución de `actualiza.sql` :

```
marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos.M/SI/SI/P2/material_P2/pgmodeler$ sudo docker exec -it material_p2_db_1 bash
psql -U alumnodb -d si1
[sudo] contraseña para marcos1701:
root@230e91596959:/# psql -U alumnodb -d si1
psql (14.8 (Debian 14.8-1.pgdg120+1))
Type "help" for help.

si1=# SELECT customerid, username, balance FROM customers;
ERROR: column "balance" does not exist
LINE 1: SELECT customerid, username, balance FROM customers;
                                     ^
si1=#
SELECT * FROM ratings;

INSERT INTO ratings (user_id, product_id, like) VALUES (1, 1, TRUE);
SELECT * FROM ratings;
ERROR: relation "ratings" does not exist
LINE 1: SELECT * FROM ratings;
               ^
ERROR: syntax error at or near "like"
LINE 1: INSERT INTO ratings (user_id, product_id, like) VALUES (1, 1...
                                     ^
ERROR: relation "ratings" does not exist
LINE 1: SELECT * FROM ratings;
               ^
si1=# SELECT orderid, totalamount FROM orders;
 orderid | totalamount
-----+-----
    316  |
   1025  |
   6260  |
   7223  |
   7779  |
```

Para ejecutar el script usamos el siguiente comando en una nueva terminal:

```
sudo docker exec -it db_db_1 psql -U alumnodb -d si1 -f /actualiza.sql
```

Ahora, puede irse irse al baño si lo necesita, pues, la ejecución durará unos 2 minutos aproximadamente 😊

Pd: No hace falta meter los scripts en el contenedor puesto que esta funcionalidad ya se ha automatizado modificando el archivo docker compose. Tampoco hará falta que ejecute más scripts manualmente a parte del ya ejecutado, puesto que en el código de `actualiza.sql` se ha añadido los comandos de ejecución del resto de scripts para así ofrecer una mejor experiencia de usuario.

Pruebas realizadas para la comprobación de las actualizaciones:

- Creación e inicialización del campo balance de customers:

```
SELECT customerid, username, balance FROM customers;
```

```
si1=# SELECT customerid, username, balance FROM customers;
customerid | username | balance
-----+-----+-----
          26 | airmen   |    16.00
        14080 | apolar   |    57.00
        14081 | spotty   |   200.00
        14082 | che      |   151.00
        14083 | hamsun   |    39.00
        14084 | sundae   |   157.00
        14085 | motile   |    63.00
        14086 | swing    |   152.00
        14087 | alicia   |    72.00
        14088 | drys     |    44.00
        14089 | bovine   |   146.00
        14090 | hay      |   161.00
        14091 | shucks   |   188.00
        14092 | burro    |   131.00
```

- Verificación del contenido de 'ratings', es decir, si se ha creado la tabla o no.

```
SELECT * FROM ratings;
INSERT INTO ratings (user_id, product_id, like) VALUES (1, 1, TRUE);
SELECT * FROM ratings;
```








```
si1=# -- Verificar contenido de 'ratings'
SELECT * FROM ratings;

-- Insertar un registro de prueba en 'ratings'
INSERT INTO ratings (user_id, product_id, is_liked) VALUES (1, 1, TRUE);
SELECT * FROM ratings;
 rating_id | user_id | product_id | is_liked
-----+-----+-----+-----
(0 rows)








INSERT 0 1
 rating_id | user_id | product_id | is_liked
-----+-----+-----+-----
          1 |         1 |           1 | t
(1 row)
```

- Ampliación del tamaño del campo password:

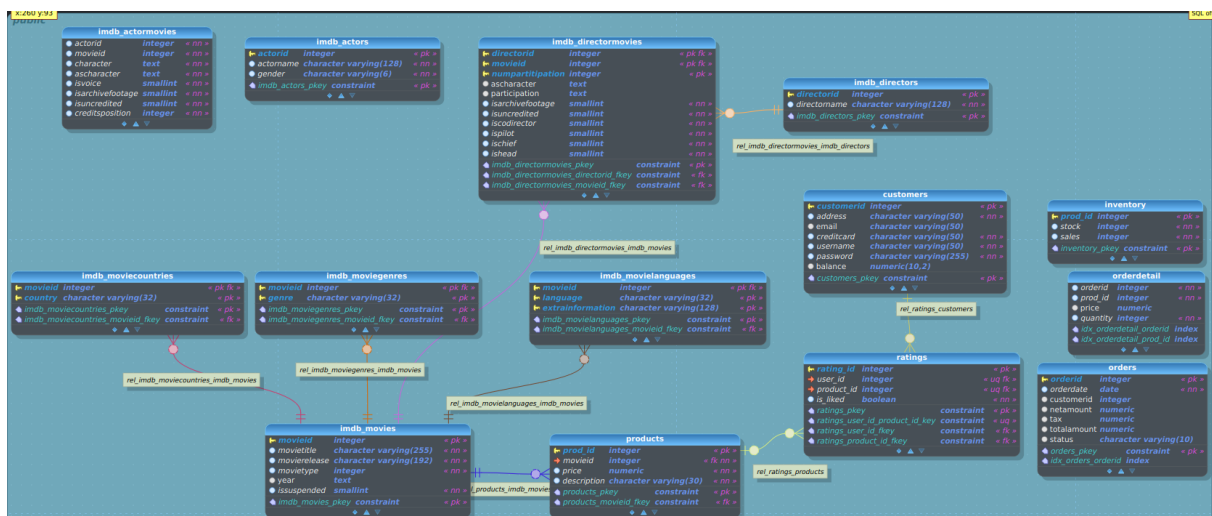
Antes de la ejecución del script:

customers		
 customerid	integer	« pk »
 address	character varying(50)	« nn »
 email	character varying(50)	
 creditcard	character varying(50)	« nn »
 username	character varying(50)	« nn »
 password	character varying(50)	« nn »
 customers_pkey	constraint	« pk »

Tras la ejecución del script:

customers		
 customerid	integer	« pk »
 address	character varying(50)	« nn »
 email	character varying(50)	
 creditcard	character varying(50)	« nn »
 username	character varying(50)	« nn »
 password	character varying(100)	« nn »
 balance	numeric(10,2)	
 customers_pkey	constraint	« pk »

Tras realizar estos cambios, el esquema E-R toma la siguiente forma:



Apartado B

En este apartado, se describe el diseño y la implementación de dos funciones que permiten calcular y actualizar el precio de los pedidos en la base de datos. La primera función,

`calcularTotalPedido`, se encarga de calcular el total de un pedido específico teniendo en cuenta el precio de los productos asociados y aplicando impuestos (tax). La segunda función, `actualizarTodosLosPedidos`, actualiza todos los pedidos de la base de datos en bloques de 10000, asegurando que el proceso sea eficiente y no sobrecargue la máquina, que ha sido un problema recurrente durante la realización de este ejercicio.

A continuación se detalla el código de ambas funciones, su propósito.

```
CREATE INDEX idx_orderdetail_orderid ON orderdetail(orderid);
CREATE INDEX idx_orderdetail_prod_id ON orderdetail(prod_id);
CREATE INDEX idx_orders_orderid ON orders(orderid);

-- Función para calcular el total de un pedido específico
CREATE OR REPLACE FUNCTION calcularTotalPedido(order_id INT)
RETURNS VOID AS $$
BEGIN
    -- 1. Actualiza el campo price en orderdetail con el precio
    UPDATE orderdetail od
    SET price = p.price
    FROM products p
    WHERE od.prod_id = p.prod_id AND od.orderid = order_id;

    -- 2. Calcula el subtotal neto y actualizar netamount en orders
    UPDATE orders
    SET netamount = (
        SELECT SUM(od.price * od.quantity)
        FROM orderdetail od
        WHERE od.orderid = orders.orderid
    )
)
```

```

WHERE orderid = order_id;

-- 3. Calcula el total con impuestos y actualizar totalamount
UPDATE orders
SET totalamount = netamount + (netamount * tax / 100)
WHERE orderid = order_id;
END;
$$ LANGUAGE plpgsql;

-- Función para actualizar todos los pedidos en bloques de 50
CREATE OR REPLACE FUNCTION actualizarTodosLosPedidos()
RETURNS VOID AS $$
DECLARE
    pedidos CURSOR FOR SELECT orderid FROM orders;
    pedido_id INT;
    contador INT := 0;
BEGIN
    OPEN pedidos;
    LOOP
        FETCH pedidos INTO pedido_id;
        EXIT WHEN NOT FOUND;

        -- Llama a calcularTotalPedido para el pedido actual
        PERFORM calcularTotalPedido(pedido_id);

        -- Incrementa el contador y verificar si ha llegado a 50
        contador := contador + 1;
        IF contador >= 10000 THEN
            contador := 0;
            -- Hace una pausa breve para evitar la sobrecarga
            PERFORM pg_sleep(0.2); -- pausa de 0.2 segundos,
        END IF;
    END LOOP;
    CLOSE pedidos;
END;
$$ LANGUAGE plpgsql;

```

```
-- Llama a la función para actualizar todos los pedidos
SELECT actualizarTodosLosPedidos();
```

Explicación de cada función:

1. `calcularTotalPedido(order_id INT)`:

- Esta función calcula el total de un pedido específico. Primero, actualiza los precios de los productos en el pedido (`orderdetail`) tomando los valores de la tabla `products` . Luego, calcula el subtotal neto del pedido y lo actualiza en el campo `netamount` de la tabla `orders` . Finalmente, calcula el total con impuestos (tax) y actualiza el campo `totalamount` en la misma tabla `orders` .

2. `actualizarTodosLosPedidos()`:

- Esta función actualiza los totales de todos los pedidos en la base de datos en bloques de 10000 pedidos. Utiliza un cursor para recorrer todos los `orderid` de la tabla `orders` , y por cada uno, llama a la función `calcularTotalPedido` para actualizar los totales. Después de procesar 10000 pedidos, realiza una breve pausa de 0.2 segundos para evitar sobrecargar el servidor. El proceso continúa hasta que todos los pedidos han sido actualizados.

Pruebas realizadas para la comprobación del resultados:

Previo a la ejecución:

```
sql=# SELECT orderid, totalamount FROM orders;
orderid | totalamount
-----+-----
 316    |
1025    |
6260    |
7223    |
7779    |
8327    |
8748    |
9148    |
9838    |
12470   |
12891   |
14011   |
14815   |
```

Tras la ejecución del script:


```

s11=# SELECT orderid, totalamount FROM orders;
 orderid |      totalamount
-----+-----
    1041 | 119.94500000000000
    70613 | 89.70000000000000
    70614 | 80.15500000000000
    70615 | 20.70000000000000
    70616 | 32.20000000000000
    70617 | 212.75000000000000
    70618 | 130.18000000000000
    70619 | 132.02000000000000
    70620 | 35.88000000000000

```

Como podemos comprobar, los valores se han actualizado correctamente en todos los pedidos, por lo que podemos asegurar que la ejecución ha sido un completo éxito.

Apartado C:

Para abordar la creación de las tablas y la modificación de atributos multivaluados, hemos creado tablas intermedias para almacenar las relaciones entre las entidades y así solucionar el problema de la multievaluación de atributos.

Se ha implementado el siguiente código:

```

-- 1. Crea tabla temporal para los países asociados a cada pe.
CREATE TABLE imdb_moviecountries_temp (
    country_id SERIAL PRIMARY KEY,
    movieid INTEGER NOT NULL,
    country CHARACTER VARYING(32) NOT NULL,
    FOREIGN KEY (movieid) REFERENCES imdb_movies(movieid) ON
);

-- Inserta los datos existentes en la nueva tabla temporal
INSERT INTO imdb_moviecountries_temp (movieid, country)
SELECT movieid, country FROM imdb_moviecountries;

-- Elimina la tabla antigua
DROP TABLE IF EXISTS imdb_moviecountries;

-- Renombra la tabla temporal para que tenga el nombre de la
ALTER TABLE imdb_moviecountries_temp RENAME TO imdb_moviecoun

```

```

-- 2. Crea tabla temporal para los géneros asociados a cada p
CREATE TABLE imdb_moviegenres_temp (
    genre_id SERIAL PRIMARY KEY,
    movieid INTEGER NOT NULL,
    genre CHARACTER VARYING(32) NOT NULL,
    FOREIGN KEY (movieid) REFERENCES imdb_movies(movieid) ON I
);

-- Inserta los datos existentes en la nueva tabla temporal
INSERT INTO imdb_moviegenres_temp (movieid, genre)
SELECT movieid, genre FROM imdb_moviegenres;

-- Elimina la tabla antigua
DROP TABLE IF EXISTS imdb_moviegenres;

-- Renombra la tabla temporal para que tenga el nombre de la
ALTER TABLE imdb_moviegenres_temp RENAME TO imdb_moviegenres;

-- 3. Crea tabla temporal para los idiomas asociados a cada p
CREATE TABLE imdb_movi_languages_temp (
    language_id SERIAL PRIMARY KEY,
    movieid INTEGER NOT NULL,
    language CHARACTER VARYING(32) NOT NULL,
    FOREIGN KEY (movieid) REFERENCES imdb_movies(movieid) ON I
);

-- Inserta los datos existentes en la nueva tabla temporal
INSERT INTO imdb_movi_languages_temp (movieid, language)
SELECT movieid, language FROM imdb_movi_languages;

-- Elimina la tabla antigua
DROP TABLE IF EXISTS imdb_movi_languages;

-- Renombra la tabla temporal para que tenga el nombre de la
ALTER TABLE imdb_movi_languages_temp RENAME TO imdb_movi_lang

```

El procedimiento seguido para la resolución de este apartado es muy sencillo. Hemos aplicado la siguiente metodología en todas las tablas que creaban problemas de atributos multivaluados:

1. Creamos una tabla temporal igual a la ya existente pero añadiéndole un id del atributo multivaluado para solucionar el problema.
2. Insertamos los datos existentes en la nueva tabla temporal
3. Eliminamos la tabla antigua
4. Renombramos la tabla temporal para que tenga el nombre de la tabla original

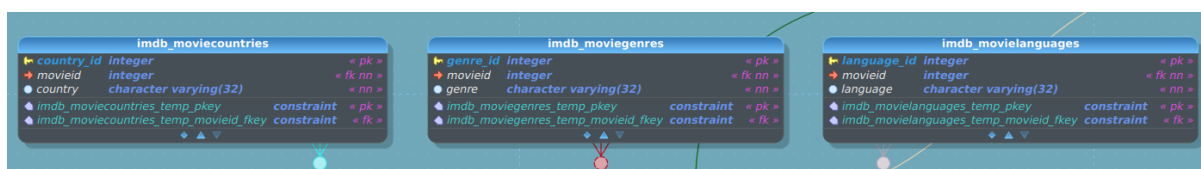
Pruebas realizadas para la comprobación del resultados:

Para realizar la comprobación de si se han realizado los cambios necesarios para eliminar los atributos multievaluados vamos a comparar los esquemas E-R antes y después de la ejecución del script.

Previo a la ejecucion:



Tras la ejecución de actualizaTablas.sql:



```

s1=# SELECT * FROM imdb_moviegenres WHERE movieid = 103;
 genre_id | movieid | genre
-----+-----+-----
        1 |      103 | Drama
        2 |      103 | History
        3 |      103 | War
(3 rows)

```

Como podemos observar, al ejecutar el script, se han reemplazado las tablas que causaban problemas de atributos multievaluados por otras nuevas, con un nuevo atributo identificativo que elimina el problema.

Apartado D

Para realizar este apartado se ha implementado el siguiente código:

```
-- Paso 1: Crea la función que recalculé el total de un pedido
CREATE OR REPLACE FUNCTION actualiza_carrito_func()
RETURNS TRIGGER AS $$
BEGIN

    IF TG_OP = 'INSERT' THEN
        -- En un INSERT, siempre recalculamos el total
        PERFORM calcularTotalPedido(NEW.orderid);

    ELSIF TG_OP = 'UPDATE' THEN
        -- Solo recalculamos si las columnas 'price' o 'quantity' han cambiado
        IF NEW.price IS DISTINCT FROM OLD.price OR NEW.quantity IS DISTINCT FROM OLD.quantity THEN
            -- Recalculamos si los valores de price o quantity han cambiado
            PERFORM calcularTotalPedido(NEW.orderid);
        END IF;

    ELSIF TG_OP = 'DELETE' THEN
        -- En un DELETE, recalculamos los totales para el `orderid` eliminado
        PERFORM calcularTotalPedido(OLD.orderid);
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Paso 2: Crea el trigger que llame a la función después de cada operación
CREATE OR REPLACE TRIGGER actualizaCarrito
AFTER INSERT OR UPDATE OR DELETE ON orderdetail
```

```
FOR EACH ROW
EXECUTE FUNCTION actualiza_carrito_func();
```

Pruebas específicas para la comprobación del trigger

1. Prueba con **INSERT**

Datos previos a la nueva inserción:

```
si1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
       1 |    1938 |     11 |         1
       1 |    1014 |     11 |         1
       1 |    1288 |     11 |         1
(3 rows)

si1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
       1 |         33 |    15 | 37.9500000000000000
(1 row)
```

Comando de inserción ejecutado :

```
INSERT INTO orderdetail (orderid, prod_id, quantity, price) VALUES (1, 101, 2, 50);
```

Datos posteriores a la nueva inserción:

```
si1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
       1 |     101 |     24 |         2
       1 |    1938 |     11 |         1
       1 |    1014 |     11 |         1
       1 |    1288 |     11 |         1
(4 rows)

si1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
       1 |         81 |    15 | 93.1500000000000000
(1 row)
```

2. Prueba con **UPDATE**

Datos previos a la actualización:

```

s1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      1 |    101 |    24 |         2
      1 |   1938 |    11 |         1
      1 |   1014 |    11 |         1
      1 |   1288 |    11 |         1
(4 rows)

s1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
      1 |      81 |  15 | 93.1500000000000000
(1 row)

```

Comando de actualización ejecutado :

```

UPDATE orderdetail
SET quantity = 3
WHERE orderid = 1 AND prod_id = 101;

```

Datos posteriores a la actualización:

```

s1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      1 |    101 |    24 |         3
      1 |   1938 |    11 |         1
      1 |   1014 |    11 |         1
      1 |   1288 |    11 |         1
(4 rows)

s1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
      1 |     105 |  15 | 120.7500000000000000
(1 row)

```

3. Prueba de DELETE

Datos previos a la eliminación:

```

s1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      1 |    101 |    24 |         3
      1 |   1938 |    11 |         1
      1 |   1014 |    11 |         1
      1 |   1288 |    11 |         1
(4 rows)

s1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
      1 |     105 |  15 | 120.7500000000000000
(1 row)

```

Comando de eliminación ejecutado :

```
DELETE FROM orderdetail WHERE orderid = 1 AND prod_id = 101;
```

Datos posteriores a la eliminación:

```
si1=# DELETE FROM orderdetail WHERE orderid = 1 AND prod_id = 101;
DELETE 1
si1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      1 |    1938 |    11 |         1
      1 |    1014 |    11 |         1
      1 |    1288 |    11 |         1
(3 rows)

si1=# SELECT orderid, netamount, tax, totalamount FROM orders WHERE orderid = 1;
 orderid | netamount | tax | totalamount
-----+-----+-----+-----
      1 |        33 |   15 | 37.9500000000000000
(1 row)
```

Apartado E:

Para realizar este apartado vamos a implementar un trigger, que cuando cambie el valor status de un pedido a 'Paid' llame a la función pedido_pagado(), la cual restará la cantidad comprada de cada producto en la tabla `inventory`, aumentará el número de ventas y reducirá el saldo en la tabla `customers` en función del `totalamount` del pedido.

```
-- Actualiza el inventario y el saldo del cliente cuando un p
CREATE OR REPLACE FUNCTION pedido_pagado()
RETURNS TRIGGER AS $$
BEGIN
    -- Verifica que el nuevo estado es "Paid"
    IF NEW.status = 'Paid' THEN
        -- Actualiza inventario: descuenta la cantidad de cada
        UPDATE inventory
        SET stock = stock - od.quantity, sales = sales + od.q
        FROM orderdetail od
        WHERE od.orderid = NEW.orderid AND inventory.prod_id =
```

```

        -- Descuenta el saldo del cliente en función del total
UPDATE customers
SET balance = balance - NEW.totalamount
WHERE customerid = NEW.customerid;
END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Crea el trigger para actualizar el inventario y el saldo del cliente
CREATE or REPLACE TRIGGER pagado
AFTER UPDATE OF status ON orders
FOR EACH ROW
WHEN (NEW.status = 'Paid')
EXECUTE FUNCTION pedido_pagado();

```

Pruebas realizadas para la comprobación del trigger:

En esta ocasión vamos a realizar la prueba usando el pedido número 1 de ejemplo, y comprobar si el trigger desempeña su función correctamente.

Datos antes de la actualización del estado del pedido:

```

s1=# SELECT * FROM orders WHERE orderid = 1;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      1 | 2020-06-21 |      693 |      33 | 15 | 37.950000000000000 | Procesing
(1 row)

s1=# SELECT * FROM inventory WHERE prod_id IN (SELECT prod_id FROM orderdetail WHERE orderid = 1);
 prod_id | stock | sales
-----+-----+-----
    1938 |   963 |   154
    1288 |   161 |   172
    1014 |   202 |   185
(3 rows)

s1=# SELECT balance FROM customers WHERE customerid = (SELECT customerid FROM orders WHERE orderid = 1);
 balance
-----
    95.00
(1 row)

```


Comando para emular el pago de un pedido:

```
UPDATE orders
SET status = 'Paid'
WHERE orderid = 1;
```

Datos después de actualizar el estado del pedido:

```
sql=# SELECT * FROM inventory WHERE prod_id IN (SELECT prod_id FROM orderdetail WHERE orderid = 1);
 prod_id | stock | sales
-----+-----+-----
    1938 |    962 |    155
    1288 |    160 |    173
    1014 |    201 |    186
(3 rows)

sql=# SELECT balance FROM customers WHERE customerid = (SELECT customerid FROM orders WHERE orderid = 1);
 balance
-----
    57.05
(1 row)
```

Integración con Python

Apartado A:

Introducción General

La API que hemos diseñado se enfoca en el ciclo de compra de una plataforma de comercio electrónico. Los usuarios pueden autenticarse, gestionar sus fondos, agregar productos a su carrito y finalizar el proceso de compra realizando un pago. Para la realización de este apartado, hemos optado por dividir la funcionalidad en varios ficheros para garantizar que cada aspecto del sistema esté bien organizado y sea fácil de mantener y ampliar.

Estructura de Ficheros

1. `api.py` : Este es el fichero principal que contiene la definición de la API en sí. Aquí definimos los endpoints y controladores para gestionar las funcionalidades clave: autenticación de usuarios, gestión del saldo, manipulación del carrito y procesamiento de pagos. La separación de la lógica en este fichero permite una estructura más limpia y facilita el uso de un framework web (como Quart) para gestionar las solicitudes HTTP.

2. `models.py` : En este archivo se definen las clases de datos y los esquemas de base de datos, utilizando SQLAlchemy como ORM. Hemos optado por colocar los modelos en un fichero separado para centralizar y aislar la lógica de datos, lo que facilita la reutilización y modificación de los modelos sin afectar a otros componentes de la API. `models.py` incluye las definiciones de usuarios, productos, carritos, y órdenes de compra, asegurando que toda la lógica de la base de datos esté unificada.
3. `config.py` : Aquí configuramos la conexión a la base de datos y las sesiones, declarando la URI de la base de datos. Al externalizar esta funcionalidad, garantizamos que el acceso y la manipulación de datos sean uniformes en toda la API.

Argumento del Diseño elegido

La decisión de dividir la API en varios módulos específicos, en lugar de colocar toda la funcionalidad en un solo archivo, está motivada por varias razones:

- **Escalabilidad:** Dividir cada aspecto en módulos separados hace que el sistema sea más fácil de ampliar. Por ejemplo, si necesitasemos añadir nuevas funcionalidades solo habría que añadir la función correspondiente al fichero `api.py`, sin modificar el resto.
- **Mantenimiento:** La separación modular facilita la resolución de errores. Si surge un problema en la conexión a la base de datos, por ejemplo, puedes localizarlo rápidamente en `config.py` sin tener que revisar la lógica de cada endpoint. Esto ha sido el mayor punto de ayuda sin duda, ya que la resolución de errores ha sido mucho más eficiente.
- **Responsabilidad Única:** Cada archivo tiene una responsabilidad clara, siguiendo el principio de responsabilidad única. Esto mejora la claridad del código y permite que cada parte sea reutilizable en otros proyectos o en futuras versiones de la API.

En los siguientes apartados, podemos repasar cada función clave en cada uno de estos archivos y analizar los motivos detrás de su implementación, así como su relación con el flujo de la API.

`models.py`

```

from sqlalchemy import Column, Integer, String, Float, Numeric
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'customers'
    customerid = Column(Integer, primary_key=True)
    address = Column(String(50), nullable=False)
    email = Column(String(50), nullable=False, unique=True)
    creditcard = Column(String(50), nullable=False)
    username = Column(String(50), nullable=False, unique=True)
    password = Column(String(100), nullable=False)
    balance = Column(Numeric(10, 2), default=0.00, nullable=False)

    # Relación uno a muchos: Un usuario puede tener muchos pedidos
    orders = relationship('Order', back_populates='user', cascade=True)

class Product(Base):
    __tablename__ = 'products'
    prod_id = Column(Integer, primary_key=True)
    movieid = Column(Integer, ForeignKey('imdb_movies.movieid'))
    price = Column(Numeric, nullable=False)
    description = Column(String(30), nullable=False)
    inventory = relationship("Inventory", back_populates="product")

class Inventory(Base):
    __tablename__ = 'inventory'
    prod_id = Column(Integer, ForeignKey('products.prod_id'), primary_key=True)
    stock = Column(Integer, nullable=False)
    sales = Column(Integer, nullable=False)
    product = relationship("Product", back_populates="inventory")

class Order(Base):
    __tablename__ = 'orders'
    orderid = Column(Integer, primary_key=True)
    orderdate = Column(Date)

```

```

    customerid = Column(Integer, ForeignKey('customers.customerid'))
    netamount = Column(Numeric)
    tax = Column(Numeric)
    totalamount = Column(Numeric)
    status = Column(String(10), nullable=False)

    # Relación inversa, los pedidos pertenecen a un usuario
    user = relationship('User', back_populates='orders') #Par

class OrderDetail(Base):
    __tablename__ = 'orderdetail'
    orderid = Column(Integer, ForeignKey('orders.orderid'), primary_key=True)
    prod_id = Column(Integer, ForeignKey('inventory.prod_id'), primary_key=True)
    quantity = Column(Integer, nullable=False)
    price = Column(Float, nullable=False)

```

En este código hemos definido un esquema de base de datos usando SQLAlchemy, un ORM en Python. Está estructurado en varias clases que representan tablas en la base de datos:

1. **User (Tabla: customers):**

- Representa a los usuarios de la base de datos.
- Incluye columnas como `customerid` (ID del cliente), `email`, `username`, `password`, y `balance`.
- Cada usuario tiene un identificador único (`customerid`) y se asegura que tanto el correo electrónico como el nombre de usuario sean únicos.

2. **Product (Tabla: products):**

- Representa los productos que están disponibles para compra.
- Incluye columnas como `prod_id` (ID del producto), `movieid` (vincula el producto a una película externa a través de `imdb_movies`), `price`, y `description`.
- Usa la relación `inventory` para asociarse con la tabla `Inventory`, lo que permite saber el stock y las ventas del producto.

3. **Inventory (Tabla: inventory):**

- Almacena la información sobre el stock y las ventas de cada producto.

- Incluye `prod_id` (relacionado con `products.prod_id`), `stock` (cantidad disponible) y `sales` (número de ventas).
- La relación con `Product` se define con `back_populates`, lo que permite una navegación bidireccional entre productos e inventario.

4. `Order` (Tabla: `orders`):

- Representa las órdenes o pedidos realizados por los usuarios.
- Incluye `orderid` (ID del pedido), `orderdate` (fecha del pedido), `customerid` (cliente que realiza el pedido), `netamount`, `tax`, `totalamount`, y `status` (estado del pedido).
- Cada pedido está vinculado a un usuario específico y almacena el monto total con impuestos y sin impuestos.

5. `OrderDetail` (Tabla: `orderdetail`):

- Contiene los detalles de cada producto en un pedido específico.
- Incluye `orderid` (ID del pedido), `prod_id` (ID del producto en inventario), `quantity` (cantidad del producto), y `price` (precio unitario en el pedido).
- Se usa una clave primaria compuesta por `orderid` y `prod_id` para manejar pedidos con múltiples productos.

`config.py`

```
import os

SQLALCHEMY_DATABASE_URI = 'postgresql+asyncpg://alumnodb:1234@localhost:5432/ecommerce'
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Este código es la configuración básica para que la aplicación utilice **SQLAlchemy** para interactuar con una base de datos en PostgreSQL.

SQLALCHEMY_TRACK_MODIFICATIONS: Hemos decidido desactivar el seguimiento de las modificaciones en los objetos de SQLAlchemy, con el fin de mejorar el rendimiento. Si se deja activado, SQLAlchemy mantendría un registro de cada cambio en los objetos, lo que no es necesario actualmente.

api.py

```
# Permite construir consultas SQL utilizando SQLAlchemy
from sqlalchemy import select
# Importa los elementos principales de Quart
from quart import Quart, request, jsonify
# Importa herramientas para manejar conexiones y sesiones de
from sqlalchemy.ext.asyncio import AsyncSession, create_async
# Facilita la creación de sesiones para interactuar con la ba
from sqlalchemy.orm import sessionmaker
# Importa los modelos de la base de datos, que representan la
from models import User, Product, Order, OrderDetail, Invento
# Importa la cadena de conexión a la base de datos.
from config import SQLALCHEMY_DATABASE_URI
# Se utiliza para verificar las contraseñas de los usuarios d
from werkzeug.security import check_password_hash
#Se importa para manejar la programación asincrónica en Pytho
import asyncio
# Se importa para manejar números decimales con precisión.
from decimal import Decimal
# Se importa para manejar excepciones de integridad en la bas
from sqlalchemy.exc import IntegrityError
# Se importa para usar funciones SQL como max.
from sqlalchemy.sql import func
# Se importa para manejar fechas y horas.
from datetime import datetime

app = Quart(__name__)

# Configuración de la base de datos
engine = create_async_engine(SQLALCHEMY_DATABASE_URI, echo=Tr
SessionLocal = sessionmaker(engine, class_=AsyncSession, expi

# Crea las tablas en la base de datos al iniciar el servidor,
@app.before_serving
async def startup():
    # Crear las tablas en la base de datos
    async with engine.begin() as conn:
```

```

        await conn.run_sync(Base.metadata.create_all)

# Genera una sesión de base de datos asincrónica
async def get_db():
    async with SessionLocal() as session:
        yield session

@app.route('/login', methods=['POST'])
async def login():
    data = await request.get_json()
    username = data.get('username')
    password = data.get('password')

    # Crea una sesión asincrónica con la base de datos usando
    async with SessionLocal() as session:
        # Ejecuta una consulta usando SQLAlchemy
        result = await session.execute(select(User).filter_by(
            username=username))
        user = result.scalar_one_or_none() # Devuelve un sol

    # Verifica si el usuario existe y si la contraseña es cor
    if user and user.password == password:
        customer_id = user.customerid
        return jsonify({"message": "Login successful", "custo
    else:
        return jsonify({"message": "Invalid username or passw

@app.route('/add_balance', methods=['POST'])
async def add_balance():
    data = await request.get_json()
    customer_id = data.get('customerid')
    amount = data.get('amount')

    # Crea una sesión asincrónica con la base de datos usando
    async with SessionLocal() as session:
        user = await session.get(User, customer_id)
        if user: # Si el usuario con ese id existe
            if user.balance is None:
                user.balance = Decimal('0.00')

```

```

        user.balance = user.balance + Decimal(amount)
        await session.commit()
        return jsonify({"message": "Balance added successfully"})
    else:
        return jsonify({"message": "User not found"}), 404

@app.route('/add_to_cart', methods=['POST'])
async def add_to_cart():
    data = await request.get_json()
    customer_id = data.get('customerid')
    prod_id = data.get('prod_id')
    quantity = data.get('quantity')

    # Crea una sesión asincrónica con la base de datos usando
    # async with SessionLocal() as session:

    product = await session.get(Product, prod_id)
    inventory = await session.get(Inventory, prod_id)

    if not product or inventory.stock < quantity:
        return jsonify({"message": "Product not available"}), 404

    # Busca el carrito pendiente del usuario
    order = await session.execute(select(Order).filter_by(customer_id=customer_id))
    order = order.scalar_one_or_none()

    # Creamos un carrito si no existe
    if not order:
        # Obtiene el siguiente valor disponible para orderid
        result = await session.execute(select(func.max(Order.orderid)))
        max_orderid = result.scalar() or 0 # Si no hay resultado, max_orderid es 0
        new_orderid = max_orderid + 1
        # Obtiene la fecha actual en formato 'YYYY-MM-DD'
        fecha_actual = func.current_date()

        order = Order(orderid = new_orderid, orderdate = fecha_actual, customer_id=customer_id, prod_id=prod_id, quantity=quantity)
        session.add(order)
        await session.commit()

```



```

        order_detail = OrderDetail(orderid=order.orderid, pro
        session.add(order_detail)
        await session.commit()

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

@app.route('/pay_cart', methods=['POST'])
async def pay_cart():
    data = await request.get_json()
    customer_id = data.get('customerid')

    async with SessionLocal() as session:
        order = await session.execute(select(Order).filter_by
        order = order.scalar_one_or_none()

        if not order:
            return jsonify({"message": "No pending order foun

        user = await session.get(User, customer_id)
        if order.totalamount > user.balance:
            return jsonify({"message": "Insufficient balance"

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

    return jsonify({"message": "Payment successful"}), 200

```

Este código implementa una aplicación web utilizando **Quart** (un framework asincrónico utilizado en la primera práctica) y **SQLAlchemy** para manejar bases de datos asincrónicas. Aquí está el resumen de las principales funciones:

- **Conexión a la base de datos:** Configura la conexión asincrónica a una base de datos PostgreSQL utilizando SQLAlchemy, lo que permite ejecutar consultas de manera eficiente en aplicaciones asincrónicas.
- **Rutas de la API:**
 - `/login` : Permite a los usuarios iniciar sesión verificando su nombre de usuario y contraseña.

- `/add_balance` : Permite a un usuario agregar saldo a su cuenta.
- `/add_to_cart` : Permite a un usuario agregar productos a su carrito de compras, validando la disponibilidad del producto y creando un pedido si es necesario.
- `/pay_cart` : Permite a un usuario pagar el carrito de compras, verificando que tenga suficiente saldo para cubrir el total del pedido.
- **Manejo de la base de datos:** Utiliza sesiones asincrónicas de SQLAlchemy para interactuar con las tablas de usuarios, productos, pedidos y detalles de pedidos. Las funciones permiten la creación, consulta y actualización de registros en la base de datos de manera eficiente.

Para ejecutarla la API, use el siguiente comando en la terminal:

```
python3 api.py
```

Pruebas realizadas para la comprobación de la api

1. Endpoint de Login (`/login`)

```
sql=# SELECT * FROM customers;
customerid | address | email | creditcard | username | password | balance
-----+-----+-----+-----+-----+-----+-----
31 | fixate flurry 77 | primed.tut@mamoot.com | 4399554125265813 | benton | boil | 121.00
```

Vamos a iniciar sesión con el usuario benton:

```
curl -X POST http://127.0.0.1:5000/login -H "Content-Type: application/json" -d '{"username": "benton", "password": "boil"}'
```

```
marcos1701@marcos1701-VirtualBox: /Desktop/marcos1701/PruebasAPI$ curl -X POST http://127.0.0.1:5000/login -H "Content-Type: application/json" -d '{"username": "benton", "password": "boil"}'
{"message": "Login successful"}
```

2. Agregar Balance (`/add_balance`)

```
sql=# SELECT * FROM customers WHERE customerid = 31;
customerid | address | email | creditcard | username | password | balance
-----+-----+-----+-----+-----+-----+-----
31 | fixate flurry 77 | primed.tut@mamoot.com | 4399554125265813 | benton | boil | 121.00
(1 row)
```

Vamos a añadirle 50\$ a benton:

```
curl -X POST http://127.0.0.1:5000/add_balance -H "Content-Type: application/json" -d '{"customerid": 31, "amount": 50.0}'
```

```
marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos/H/51/51/P2/src/API$ curl -X POST http://127.0.0.1:5000/add_balance -H "Content-Type: application/json" -d '{"customerid": 31, "amount": 50.0}'
{"message": "Balance added successfully"}
```

```
si1=# SELECT * FROM customers WHERE customerid = 31;
 customerid | address | email | creditcard | username | password | balance
-----+-----+-----+-----+-----+-----+-----
31 | fixate flurry 77 | primed.tut@mamoot.com | 4399554125265813 | benton | boil | 171.00
(1 row)
```

3. Agregar Producto al Carrito (/add_to_cart)

```
si1=# SELECT * FROM orders WHERE orderid = 316;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
316 | 2021-06-30 | 17 | 153.1 | 15 | 176.0650000000000000 | Pending
(1 row)

si1=# SELECT * FROM orderdetail WHERE orderid = 316;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
316 | 4339 | 18 | 1
316 | 4787 | 11 | 1
316 | 2624 | 22.8 | 1
316 | 2041 | 28.5 | 1
316 | 4378 | 20.4 | 1
316 | 2737 | 15 | 1
316 | 1738 | 20.4 | 1
316 | 3563 | 17 | 1
(8 rows)
```

Vamos a añadirle 6 unidades del producto con id 101 al carrito del usuario con id 17:

```
curl -X POST http://127.0.0.1:5000/add_to_cart -H "Content-Type: application/json" -d '{"customerid": 17, "prod_id": 101, "quantity": 6}'
```

```
marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos/H/51/51/P2/src/API$ curl -X POST http://127.0.0.1:5000/add_to_cart -H "Content-Type: application/json" -d '{"customerid": 17, "prod_id": 101, "quantity": 6}'
{"message": "Product added to cart"}
```

```
si1=# SELECT * FROM orders WHERE orderid = 316;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
316 | 2021-06-30 | 17 | 297.1 | 15 | 341.6650000000000000 | Pending
(1 row)

si1=# SELECT * FROM orderdetail WHERE orderid = 316;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
316 | 101 | 24 | 6
316 | 2624 | 22.8 | 1
316 | 2041 | 28.5 | 1
316 | 4378 | 20.4 | 1
316 | 2737 | 15 | 1
316 | 1738 | 20.4 | 1
316 | 3563 | 17 | 1
316 | 4339 | 18 | 1
316 | 4787 | 11 | 1
(9 rows)
```

4. Pagar el Carrito (/pay_cart)

Esta prueba sea probablemente la más compleja de realizar, ya que depende también del trigger de pagado.sql y hay que comprobar que estos 3 campos se actualizan correctamente:

- Que el balance del usuario disminuya de acuerdo al totalamount del pedido.
- Que el stock del inventory de los productos asociados a ese pedido se decremente de acuerdo a la cantidad que se encuentran en el pedido pagado.
- Que las sales de los productos aumente en función de la cantidad de cada producto vendido.

```
si1=# SELECT * FROM orders WHERE orderid = 1;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      1 | 2020-06-21 |      693 |      33 |  15 | 37.9500000000000000 | Pending
(1 row)

si1=# SELECT * FROM orderdetail WHERE orderid = 1;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      1 |   1938 |    11 |         1
      1 |   1014 |    11 |         1
      1 |   1288 |    11 |         1
(3 rows)

si1=# SELECT * FROM inventory WHERE prod_id = 1938 OR prod_id = 1014 OR prod_id = 1288;
 prod_id | stock | sales
-----+-----+-----
   1014 |   202 |   185
   1288 |   161 |   172
   1938 |   963 |   154
(3 rows)

si1=# SELECT customerid, balance FROM customers WHERE customerid = 693;
 customerid | balance
-----+-----
        693 |   182.00
(1 row)
```

Vamos a pasar el pedido pendiente del cliente con id 693 a 'Paid':

```
curl -X POST http://127.0.0.1:5000/pay_cart -H "Content-Type: application/json" -d '{"customerid": 693}'
```

```
marcosi701@marcosi701-VirtualBox:~/Escritorio/marcos-h/si/si/p2/src/API$ curl -X POST http://127.0.0.1:5000/pay_cart -H "Content-Type: application/json" -d '{"customerid": 693}'
{"message": "Payment successful"}
```

```

si1=# SELECT * FROM orders WHERE orderid = 1;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      1 | 2020-06-21 |      693 |      33 | 15 | 37.950000000000000 | Paid
(1 row)

si1=# SELECT * FROM inventory WHERE prod_id = 1938 OR prod_id = 1014 OR prod_id = 1288;
 prod_id | stock | sales
-----+-----+-----
     1014 |   201 |   186
     1288 |   160 |   173
     1938 |   962 |   155
(3 rows)

si1=# SELECT customerid, balance FROM customers WHERE customerid = 693;
 customerid | balance
-----+-----
          693 | 144.05
(1 row)

```

Como podemos observar, todos los datos han sido actualizados perfectamente. Por lo que podemos concluir que la implementación realizada hasta la fecha sobre la API es excelente.

Funciones EXTRA realizadas en la api:

Listado de productos en carrito

Como función extra hemos implementado `list_cart_products()` una función que nos permite ver los objetos que tenemos en nuestro carrito pendiente, para asegurarnos de que vamos a comprar únicamente lo que queremos.

```

@app.route('/cart_products', methods=['POST'])
async def list_cart_products():
    data = await request.get_json()
    customer_id = data.get('user_id') # Recibe el 'user_id'
                                     # Hemos optado por esta opción

    if not customer_id:
        return jsonify({"error": "user_id is required"}), 400

    # Crea una sesión asincrónica con la base de datos
    async with SessionLocal() as session:
        # Consulta los productos en el carrito del usuario
        order = await session.execute(select(Order).filter_by(
            order_id = order_id).scalar_one_or_none())

```

```

if not order:
    return jsonify({"message": "No pending order found"})

order_details_result = await session.execute(
    select(OrderDetail, Product)
    .join(Product, Product.prod_id == OrderDetail.product_id)
    .filter(OrderDetail.orderid == order.orderid)
)

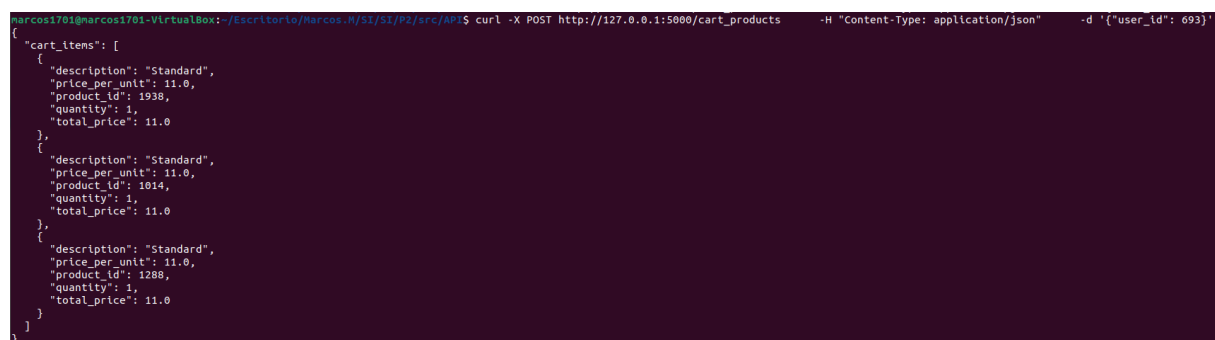
cart_items = [
    {
        "product_id": product.prod_id,
        "description": product.description,
        "quantity": order_detail.quantity,
        "price_per_unit": order_detail.price,
        "total_price": order_detail.quantity * order_detail.price
    }
    for order_detail, product in order_details_result.items()
]

# Devuelve el carrito en formato JSON
return jsonify({"cart_items": cart_items}), 200

```

Para probarla hemos ejecutado el siguiente comando suponiendo que somos el usuario 693:

```
curl -X POST http://127.0.0.1:5000/cart_products -H "Content-Type: application/json" -d '{"user_id": 693}'
```



```

marcos1701@marcos1701-VirtualBox: ~/Escritorio/marcos.N/SI/SI/P2/src/API$ curl -X POST http://127.0.0.1:5000/cart_products -H "Content-Type: application/json" -d '{"user_id": 693}'
{"cart_items": [{"description": "Standard", "price_per_unit": 11.0, "product_id": 1938, "quantity": 1, "total_price": 11.0}, {"description": "Standard", "price_per_unit": 11.0, "product_id": 1014, "quantity": 1, "total_price": 11.0}, {"description": "Standard", "price_per_unit": 11.0, "product_id": 1288, "quantity": 1, "total_price": 11.0}]}

```

Registro de nuevos usuarios

Como segunda función extra, hemos decidido implementar el registro de nuevos usuarios (`register_user()`), ya que lo consideramos algo imprescindible en un servicio web.

Hemos decido que los nuevos usuarios se creen con un balance de cero, para evitar posibles fallas de seguridad en cuanto a las peticiones al servidor, esto también nos permite centrar unicamente nuestro foco de protección en cuanto al balance en la función de `add_balance`. También hemos dedidido añadir un manejador de errores de integridad, con la finalidad de evitar la creación de multicuentas para evitar posibles fraudes. Esta medida ayuda a mantener la integridad del sistema y protege contra el abuso de cuentas múltiples. Por último, hemos calculado el último valor de `customerid` de la base de datos proporcionada por el profesorado para poder calcular el identificador del nuevo usuario y así evitar una violación de primary key.

A continuación, se muestra el código implementado para el registro de nuevos usuarios:

```
@app.route('/register', methods=['POST'])
async def register_user():
    data = await request.get_json()

    # Extracción y verificación de datos
    required_fields = ["address", "email", "creditcard", "use"]
    for field in required_fields:
        if field not in data:
            return jsonify({"error": f"{field} is required"})

    async with SessionLocal() as session:
        # Verifica si el email ya existe
        result = await session.execute(select(User).filter_by(
            existing_email_user = result.scalar_one_or_none()
        ))
        if existing_email_user:
            return jsonify({"error": "Email already exists"})

        # Verifica si el username ya existe
```

```

result = await session.execute(select(User).filter_by(
existing_username_user = result.scalar_one_or_none()
if existing_username_user:
    return jsonify({"error": "Username already exists

# Obtiene el siguiente valor disponible para customer
result = await session.execute(select(func.max(User.c
max_customerid = result.scalar() or 0 # Si no hay re
new_customerid = max_customerid + 1

# Crea nuevo usuario si el email y username no existe
new_user = User(
    customerid=new_customerid,
    address=data['address'],
    email=data['email'],
    creditcard=data['creditcard'],
    username=data['username'],
    password=data['password'],
    balance=0 #Hemos decidido que el saldo inicial

)

# Guardaa usuario en la base de datos
session.add(new_user)
try:
    await session.commit()
    return jsonify({"message": "User registered succe
except IntegrityError: # Captura el error de int
    await session.rollback()
    return jsonify({"error": "Email or username alrea
except Exception as e:
    await session.rollback()
    return jsonify({"error": str(e)}), 500

```

Prueba de ejecucución:

Hemos introducido un nuevo usuario en la base de datos mediante la nueva función implementada:

```
curl -X POST http://127.0.0.1:5000/register -H "Content-Type: application/json" -d '{"address": "Calle de la magia", "email": "sistemas.informaticos@estudiante.uam.es", "creditcard": "12344803485308034574", "username": "Marcos_Y_Nacho", "password": "semana_de_Javier"}'
```

```
marcos1701@marcos1701-VirtualBox: /Escritorio/Marcos_Y_Nacho/SI/P2/src/API$ curl -X POST http://127.0.0.1:5000/register -H "Content-Type: application/json" -d '{"address": "Calle de la magia", "email": "sistemas.informaticos@estudiante.uam.es", "creditcard": "12344803485308034574", "username": "Marcos_Y_Nacho", "password": "semana_de_Javier"}'
{"message": "User registered successfully"}
```

```
sql=# SELECT * FROM customers WHERE email = 'sistemas.informaticos@estudiante.uam.es';
 customerid | address | email | creditcard | username | password | balance
-----
 14094 | Calle de la magia | sistemas.informaticos@estudiante.uam.es | 12344803485308034574 | Marcos_Y_Nacho | semana_de_Javier | 0.00
(1 row)
```

Como podemos observar la implementación es más que correcta.

Eliminación de usuarios

Por último, para implementar la eliminación de un usuario en nuestra API, hemos desarrollado una función `eliminar_usuario()`. Consideramos esta funcionalidad esencial para permitir que los usuarios puedan eliminar sus cuentas, asegurando al mismo tiempo la integridad de sus datos asociados. Este proceso incluye la eliminación de todos los pedidos y detalles de pedidos relacionados con el usuario, garantizando una limpieza completa y evitando datos huérfanos en la base de datos.

Dado que la eliminación de registros está relacionada directamente con el usuario y sus pedidos, decidimos realizarlo en varias etapas. En primer lugar, se recupera al usuario mediante su `customerid`, y luego se seleccionan todos sus pedidos y los detalles asociados a cada uno de ellos. Los detalles de pedidos se eliminan primero, seguidos por los pedidos, y finalmente el usuario en sí. Este orden en cascada asegura la consistencia de la base de datos y previene errores de integridad.

Para manejar posibles errores y mantener la fiabilidad del sistema, también se ha implementado un bloque de manejo de excepciones. En caso de que ocurra

algún error durante el proceso de eliminación, se realiza un rollback para revertir cualquier cambio no deseado en la base de datos, protegiendo así la integridad de los datos. Finalmente, tras una eliminación exitosa o una excepción, la sesión se cierra para liberar recursos.

A continuación se presenta el código implementado para la función de eliminación de usuario:

```
@app.route('/delete_user', methods=['DELETE'])
async def eliminar_usuario():
    data = await request.get_json()
    user_id = data.get('customerid')
    async with SessionLocal() as session:
        # Buscar al usuario en la base de datos
        result = await session.execute(select(User).filter_by(
            user_id=user_id))
        user = result.scalars().first() # Obtener el primer usuario

        if not user:
            return jsonify({"message": "Usuario no encontrado"})

        try:
            # Obtiene todos los pedidos del usuario
            orders = await session.execute(select(Order).filter_by(
                user_id=user_id))
            orders_list = orders.scalars().all()

            # Obtiene los detalles de los pedidos
            orders_details_list = []
            for order in orders_list:
                order_details = await session.execute(select(
                    OrderDetail).filter_by(order_id=order.id))
                orders_details_list.extend(order_details.scalars().all())

            # Elimina los detalles de los pedidos
            for order_detail in orders_details_list:
                await session.delete(order_detail)

            # Elimina los pedidos
            for order in orders_list:
                await session.delete(order)
```

```

# Finalmente, elimina el usuario
await session.delete(user)

# Commit para guardar los cambios
await session.commit()

return jsonify({"message": "Usuario, sus pedidos"
except Exception as e:
    await session.rollback() # Hace rollback en caso
    return jsonify({"message": "Hubo un error al elim
finally:
    await session.close()

```

Prueba de ejecución:

Hemos eliminado un usuario de la base de datos mediante la nueva función implementada.

Datos del usuario Javier con id 14094 antes de su eliminación:

```

s1=# SELECT * FROM customers WHERE customerid =14094;
 customerid | address | email | creditcard | username | password | balance
-----+-----+-----+-----+-----+-----+-----
14094 | 1234 Main St, Springfield, IL | LaSemanaDeJavier@uam.com | 1234-5678-9012-3456 | Javier | securepassword123 | 9795.51
(1 row)

s1=# SELECT * FROM orders WHERE customerid = 14094;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
181791 | 2024-11-11 | 14094 | 169.0 | 21 | 204.4900000000000000 | Paid
(1 row)

s1=# SELECT * FROM orderdetail WHERE orderid = 181791;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
181791 | 1 | 13 | 2
181791 | 2 | 15 | 2
181791 | 3 | 18 | 2
181791 | 4 | 22.5 | 2
181791 | 5 | 16 | 2
(5 rows)

```

Comando ejecutado para la eliminación del usuario:

```

curl -X DELETE http://127.0.0.1:5000/delete_user -H "Content-Type: application/json"
-d '{"customerid": 14094}'

```

```

marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos.N/SI/SI/P2/src/API$ curl -X DELETE http://127.0.0.1:5000/delete_user -H "Content-Type: application/json" -d '{"user_id": 14094}'
{"message": "Usuario, sus pedidos y detalles han sido eliminados"}

```

```

s1=# SELECT * FROM customers WHERE customerid =14094;
 customerid | address | email | creditcard | username | password | balance
-----+-----+-----+-----+-----+-----+-----
(0 rows)

s1=# SELECT * FROM orders WHERE customerid = 14094;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
(0 rows)

s1=# SELECT * FROM orderdetail WHERE orderid = 181791;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
(0 rows)

```

Como podemos comprobar, la eliminación del usuario y de todos sus datos correspondientes, han sido borrados de manera correcta.

Cliente

cliente.py

```

import requests
import colorama
from colorama import Fore, Style

BASE_URL = "http://127.0.0.1:5000"

# 1. Registra y autentica al usuario
def register_user(address, email, creditcard, username, password):
    data = {
        "username": username,
        "email": email,
        "password": password,
        "address": address,
        "creditcard": creditcard
    }
    response = requests.post(f"{BASE_URL}/register", json=data)
    print("Register Response:", response.json())
    return response

# 2. Inicio de sesión del usuario
def login_user(username, password):

```

```

data = {
    "username": username,
    "password": password
}
response = requests.post(f"{BASE_URL}/login", json=data)
print("Login Response:", response.json())
return response

# 3. Añadir saldo a la cuenta del usuario
def add_balance(customer_id, amount):
    data = {
        "customerid": customer_id,
        "amount": amount
    }
    response = requests.post(f"{BASE_URL}/add_balance", json=
    print("Add Balance Response:", response.json())
    return response

# 4. Añadir productos al carrito
def add_to_cart(customer_id, product_id, quantity):
    data = {
        "customerid" : customer_id,
        "prod_id": product_id,
        "quantity": quantity
    }
    response = requests.post(f"{BASE_URL}/add_to_cart", json=
    print("Add TO Cart Response:", response.json())
    return response

# 5. Ver carrito actual
def view_cart(customer_id):
    data = {
        "user_id": customer_id
    }
    response = requests.post(f"{BASE_URL}/cart_products", jso
    print("View Cart Response:", response.json())
    return response

```

```

# 6. Realizar el pago del carrito
def checkout(customer_id):
    data = {
        "customerid": customer_id
    }
    response = requests.post(f"{BASE_URL}/pay_cart", json=data)
    print("Checkout Response:", response.json())
    return response

# 7. Eliminar el usuario
def customer_delete(customer_id):
    data = {
        "customerid": customer_id
    }
    response = requests.delete(f"{BASE_URL}/delete_user", json=data)
    print("Delete User Response:", response.json())
    return response

# Función principal de pruebas exhaustivas
def main():

    # Inicializamos colorama
    colorama.init(autoreset=True)

    # Definición del tick verde
    tick = "✓"
    frase_exito = "Operación exitosa"

    # Definición de la "X" roja
    x_roja = "X"
    frase_fracaso = "Operación fallida"

    # Usuario de prueba
    username = "Javier"
    email = "LaSemanaDeJavier@uam.com"
    password = "securepassword123"

```

```

address = "1234 Main St, Springfield, IL"
creditcard = "1234-5678-9012-3456"
initial_balance = 10000 # Para garantizar que se pueden
product_quantity = 2 # Cantidad por producto en las pruebas
customer_id = None
products = [1,2,3,4,5] # Lista de ids de productos a añadir

# Prueba 1: Registrar usuario
print("\n=== Registration Test ===")
register_response = register_user(address, email, creditcard)

if register_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al crear el usuario")

# Prueba 2: Iniciar sesión
print("\n=== Login Test ===")
login_response = login_user(username, password)
if login_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al iniciar sesión")
customer_id = login_response.json().get("customerid")

# Prueba 3: Añadir saldo al usuario
print("\n=== Add Balance Test ===")
add_balance_response = add_balance(customer_id, initial_balance)
if add_balance_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al añadir saldo al usuario")

# Prueba 4: Añadir productos al carrito
print("\n=== Add Products to Cart Test ===")

```

```

for prod_id in products: # Seleccionamos hasta 3 products
    add_to_cart_response = add_to_cart(customer_id, prod_id)
    if add_to_cart_response.status_code == 200:
        print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
    else:
        print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
        raise Exception("Error al añadir productos al carrito")

# Prueba 5: Verificar el carrito actual
print("\n=== View Cart Test ===")
view_cart_response = view_cart(customer_id)
if view_cart_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al ver el carrito")

# Prueba 6: Realizar el pago del carrito
print("\n=== Checkout Test ===")
checkout_response = checkout(customer_id)
if checkout_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al realizar el pago del carrito")

# Prueba 7: Eliminar el usuario
print("\n=== Delete User Test ===")
delete_response = customer_delete(customer_id)
if delete_response.status_code == 200:
    print(f"{Fore.GREEN}{tick} {frase_exito}{Style.RESET_ALL}")
else:
    print(f"{Fore.RED}{x_roja} {frase_fracaso}{Style.RESET_ALL}")
    raise Exception("Error al eliminar el usuario")

if __name__ == "__main__":
    main()

```


Este script que hemos realizado automatiza una serie de pruebas para verificar el correcto funcionamiento de la API REST. El código está organizado en funciones que simulan las principales interacciones de un usuario con el sistema, desde registrarse hasta eliminar su cuenta, y cuenta con retroalimentación visual para cada prueba (indicada con un tick verde para éxito y una "X" roja para fallo), lo cual facilita la detección de errores en cada paso.

Descripción de las Funciones Principales

1. Registro y Autenticación:

- `register_user` : Crea un nuevo usuario enviando datos como nombre de usuario, email, dirección y tarjeta de crédito al endpoint `/register` . El usuario se crea con un balance inicial de cero para evitar problemas de seguridad.
- `login_user` : Autentica al usuario en el sistema utilizando su nombre de usuario y contraseña. Al iniciar sesión exitosamente, se recibe el `customer_id` , que se utiliza para identificar al usuario en las pruebas posteriores.

2. Gestión de Balance:

- `add_balance` : Añade saldo a la cuenta del usuario, lo cual es esencial para realizar compras en el sistema. Este endpoint recibe el `customer_id` y el monto a agregar.

3. Gestión del Carrito:

- `add_to_cart` : Agrega productos al carrito de compras del usuario. Se utiliza una lista de identificadores de productos y cantidades predefinidas para realizar múltiples pruebas de adición al carrito.
- `view_cart` : Recupera el contenido actual del carrito, permitiendo verificar que los productos añadidos están correctamente almacenados.

4. Realización de Pago:

- `checkout` : Finaliza la compra y realiza el pago del carrito, lo cual debe reducir el saldo del usuario de acuerdo con el total de la compra y vaciar el carrito.

5. Eliminación del Usuario:

- `customer_delete` : Permite eliminar el usuario de la base de datos. Elimina el usuario junto con los datos asociados, como pedidos y detalles del carrito, asegurando la integridad de la base de datos.

Ejecución de Pruebas

Cada prueba se inicia en la función `main`, que implementa las pruebas de forma secuencial, imprimiendo mensajes de éxito o fallo. Se define un usuario de prueba y valores de inicialización para los tests. Además, se muestra un resumen de cada paso con retroalimentación en pantalla:

- **Éxito:** Se muestra un "✓" en verde para cada prueba completada exitosamente.
- **Error:** En caso de fallo, se muestra una "X" en rojo y el programa arroja una excepción para detener el flujo, ayudando a identificar la prueba específica que presenta problemas.

Al final, este script proporciona una visión clara y completa de la interacción del usuario con el sistema, validando desde la creación hasta la eliminación de la cuenta y comprobando la funcionalidad de cada endpoint.

Optimización

Lo primero que tenemos que hacer para poder operar con la otra base de datos proporcionada es modificar el archivo `docker-compose.yml` para que inicie esta nueva base de datos que evita las actualizaciones automáticas de estadísticas:

```
version: '3.4'
services:
  db:
    image: postgres:14.8
    restart: on-failure
    environment:
      - POSTGRES_PASSWORD=1234
      - POSTGRES_DB=si1
      - POSTGRES_USER=alumnodb
    ports:
      - "127.0.0.1:5432:5432"
```

```

volumes:
  - ./dump_p2_v1.sql.gz:/docker-entrypoint-initdb.d/1_data.sql.gz
  - ./scripts/actualiza.sql:/actualiza.sql
  - ./scripts/actualizaPrecios.sql:/actualizaPrecios.sql
  - ./scripts/actualizaTablas.sql:/actualizaTablas.sql
  - ./scripts/actualizaCarrito.sql:/actualizaCarrito.sql
  - ./scripts/pagado.sql:/pagado.sql
  - ./scripts/actualizaPreciosPedidos.sql:/actualizaPreciosPedidos.sql

db_v2:
  image: postgres:14.8
  restart: on-failure
  environment:
    - POSTGRES_PASSWORD=1234
    - POSTGRES_DB=si2
    - POSTGRES_USER=alumnodb
  ports:
    - "127.0.0.1:5433:5432"
  volumes:
    - ./dump_p2_v2.sql.gz:/docker-entrypoint-initdb.d/2_data.sql.gz

```

Como se puede observar, usamos otro puerto para evitar conflictos con la base de datos original.

- **db** : La versión original en el puerto **5432**.
- **db_v2** : La nueva versión, en el puerto **5433**, con autovacuum y auto-analyze deshabilitados.

Nota : Si quiere comprobar los resultados de nuestras consultas en su maquina personal para la corrección de la práctica, ejecute los siguientes comandos en la terminal para acceder a esta segunda base de datos :)

```

sudo docker exec -it db_db_v2_1 bash
psql -U alumnodb -d si2

```

Estudio del impacto de un índice:

Apartado A

estadosDistintos.sql

```
SELECT COUNT(DISTINCT state) AS estados_distintos
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = :anio
      AND c.country = :pais;
```

Donde `:anio` y `:pais` representan *placeholders*, es decir, son nombres que sirven para señalar dónde deben ir los valores que se pasarán en la consulta.

Apartado B

Para realizar este apartado, hemos creado un archivo python donde ejecutar las consultas. Este es el código implementado:

```
import psycopg2

def ejecutar_consulta(anio, pais):
    # Configura los parámetros de conexión a la base de datos
    conexion = psycopg2.connect(
        host="127.0.0.1", # Dirección de la base de datos (localhost)
        port="5433",      # Puerto, de la segunda base de datos
        database="si2",    # Nombre de la base de datos
        user="alumnodb",   # Usuario de la base de datos
        password="1234"    # Contraseña de la base de datos
    )

    # Define el cursor
    cursor = conexion.cursor()

    # Define la consulta SQL con placeholders
    consulta_sql = """
        EXPLAIN
        SELECT COUNT(DISTINCT state)
        FROM customers AS c
        JOIN orders AS o ON c.customerid = o.customerid
    """
```

```

        WHERE EXTRACT(YEAR FROM o.orderdate) = %s AND c.count
    """

    try:
        # Ejecuta la consulta con los valores dados para anio
        cursor.execute(consulta_sql, (anio, pais))

        # Recupera el resultado
        resultado = cursor.fetchone()[0]
        print(f"El número de estados distintos: {resultado}")

    except Exception as e:
        print(f"Error al ejecutar la consulta: {e}")

    finally:
        # Cierra el cursor y la conexión
        cursor.close()
        conexion.close()

# Llama a la función con los parámetros deseados
ejecutar_consulta(2017, 'Perú')

```

Al ejecutar el archivo Ejecucion_consultas.py, obtenemos el siguiente resultado:

```

marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos.M/SI/SI/P2/src/DB/scripts_optimizacion$ python3 Ejecucion_consultas.py
El número de estados distintos: Aggregate (cost=4821.98..4821.99 rows=1 width=8)

```

```

si2=# EXPLAIN
SELECT COUNT(DISTINCT state)
FROM customers AS c
JOIN orders AS o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = 2017 AND c.country = 'Perú';
               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: (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 = 'Perú'::text)

(10 rows)

```

Componentes del Plan de Ejecución

1. `Aggregate` :

- Esta es la operación que se está llevando a cabo. En este caso, se trata de una operación de agregación (`COUNT(DISTINCT state)`).
- PostgreSQL está usando una **agregación** para contar los distintos estados.

2. `cost=4821.98..4821.99` :

- El `cost` se refiere al costo estimado de la operación en términos de recursos computacionales (como CPU, I/O, etc.).
- El primer número (`4821.98`) es el costo estimado de **inicio** de la operación, mientras que el segundo (`4821.99`) es el costo estimado **final** después de completar la operación.
- Este valor es solo una estimación para ayudar al optimizador a elegir la mejor manera de ejecutar la consulta.

3. `rows=1` :

- Esta es la estimación de cuántas filas se devolverán como resultado de la operación de agregación. En este caso, se espera una sola fila de resultados, que es el número de estados distintos.

4. `width=8` :

- Este valor indica el tamaño promedio de cada fila en bytes. En este caso, el valor es 8, lo que significa que cada fila del resultado de la operación `Aggregate` ocupa 8 bytes.

¿Qué significa esta salida?

El optimizador de PostgreSQL está sugiriendo que la consulta será una agregación (en este caso, un `COUNT(DISTINCT state)`), y está estimando que el costo de realizar esta agregación es bajo (`4821.98` hasta `4821.99`). El número de filas procesadas es bajo (1 fila de resultado) y el costo en términos de recursos es bastante reducido

Apartados C , D y E

Para **obtener** una mejor optimización en la ejecución de la consulta previamente mencionada:

- Creamos un Índice en las Columnas `customerid` y `country` de la Tabla `customers`

Dado que la consulta filtra por `country`, empezaremos creando un índice en `customers` sobre esta columna:

```
CREATE INDEX idx_customers_country ON customers(country);
```

- Creamos un Índice en `orders` sobre `customerid` y `orderdate`

Dado que la consulta también filtra por año en `orderdate`, un índice en `orderdate` mejorará el rendimiento. Aquí usamos una función sobre `orderdate` para obtener solo el año, lo que permitirá que el índice sea más efectivo:

```
CREATE INDEX idx_orders_orderdate_year ON orders(EXTRACT(YEAR FROM orderdate));
```

Pruebas de ejecución:

Ejecución previa a la inyección de los índices:

```

s12=# EXPLAIN ANALYZE
SELECT COUNT(DISTINCT state) AS estados_distintos
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = 2017
AND c.country = 'Perú';

-----
QUERY PLAN
-----
Aggregate  (cost=4821.98..4821.99 rows=1 width=8) (actual time=23.682..26.646 rows=1 loops=1)
-> Gather  (cost=1529.04..4821.97 rows=5 width=118) (actual time=23.669..26.632 rows=0 loops=1)
    Workers Planned: 1
    Workers Launched: 1
    -> Hash Join (cost=529.04..3821.47 rows=3 width=118) (actual time=1.518..1.520 rows=0 loops=2)
        Hash Cond: (o.customerid = c.customerid)
        -> Parallel Seq Scan on orders o (cost=0.00..3291.03 rows=535 width=4) (actual time=0.025..0.026 rows=1 loops=2)
            Filter: (EXTRACT(year FROM orderdate) = '2017'::numeric)
            Rows Removed by Filter: 142
        -> Hash (cost=528.16..528.16 rows=70 width=122) (actual time=1.428..1.429 rows=0 loops=2)
            Buckets: 1024 Batches: 1 Memory Usage: 8kB
            -> Seq Scan on customers c (cost=0.00..528.16 rows=70 width=122) (actual time=1.427..1.428 rows=0 loops=2)
                Filter: ((country)::text = 'Perú'::text)
                Rows Removed by Filter: 14093
Planning Time: 0.277 ms
Execution Time: 26.679 ms
(16 rows)

```

Ejecución posterior a la inyección de los índices:

```

s12=# EXPLAIN ANALYZE
SELECT COUNT(DISTINCT state) AS estados_distintos
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = 2017
AND c.country = 'Perú';

-----
QUERY PLAN
-----
Aggregate  (cost=1681.90..1681.91 rows=1 width=8) (actual time=2.402..2.403 rows=1 loops=1)
-> Hash Join (cost=200.17..1681.89 rows=5 width=118) (actual time=2.378..2.379 rows=0 loops=1)
    Hash Cond: (o.customerid = c.customerid)
    -> Bitmap Heap Scan on orders o (cost=19.46..1498.79 rows=909 width=4) (actual time=2.353..2.354 rows=1 loops=1)
        Recheck Cond: (EXTRACT(year FROM orderdate) = '2017'::numeric)
        Heap Blocks: exact=1
        -> Bitmap Index Scan on idx_orders_orderdate_year (cost=0.00..19.24 rows=909 width=0) (actual time=2.191..2.191 rows=36445 loops=1)
            Index Cond: (EXTRACT(year FROM orderdate) = '2017'::numeric)
    -> Hash (cost=179.83..179.83 rows=70 width=122) (actual time=0.019..0.019 rows=0 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 8kB
        -> Bitmap Heap Scan on customers c (cost=4.83..179.83 rows=70 width=122) (actual time=0.018..0.018 rows=0 loops=1)
            Recheck Cond: ((country)::text = 'Perú'::text)
            -> Bitmap Index Scan on idx_customers_country (cost=0.00..4.81 rows=70 width=0) (actual time=0.017..0.017 rows=0 loops=1)
                Index Cond: ((country)::text = 'Perú'::text)
Planning Time: 0.960 ms
Execution Time: 2.450 ms
(16 rows)

```

Al añadir estos índices la ejecución muestra una mejora significativa al eliminar la exploración secuencial. Cabe recalcar que el tiempo de planificación aumenta, pero es insignificante si tenemos en cuenta el tiempo de ejecución total.

Apartado F

Actualizamos

`estadosDistintos.sql` para que incluya las sentencias de creación de índices, como se indica en el enunciado:

```
CREATE INDEX IF NOT EXISTS idx_customers_country ON customers
CREATE INDEX IF NOT EXISTS idx_orders_orderdate_year ON orders

SELECT COUNT(DISTINCT state) AS estados_distintos
FROM customers c
JOIN orders o ON c.customerid = o.customerid
WHERE EXTRACT(YEAR FROM o.orderdate) = :anio
AND c.country = :pais;
```

Esta estructura de

`estadosDistintos.sql` asegura que los índices necesarios están presentes antes de ejecutar la consulta, optimizando el rendimiento de la misma.

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

Como podemos observar, el anexo 1 contiene las siguientes consultas:

- Consulta 1: `NOT IN`

```
EXPLAIN
SELECT customerid
FROM customers
```



```
WHERE customerid NOT IN (
    SELECT customerid
    FROM orders
    WHERE status = 'Paid'
);
```

- Esta consulta usa el operador **NOT IN**, que generalmente requiere escanear la tabla **customers** y luego verificar que el **customerid** no esté en los resultados de la subconsulta.
- La subconsulta se ejecuta primero, y sus resultados se almacenan para luego comparar con los **customerid** en **customers**

Ejemplo de ejecución:

```
sl2=# EXPLAIN ANALYZE SELECT customerid
FROM customers
WHERE customerid NOT IN (
    SELECT customerid
    FROM orders
    WHERE status = 'Paid'
);
                                QUERY PLAN
-----
Index Only Scan using customers_pkey on customers  (cost=3961.93..4372.56 rows=7046 width=4) (actual time=85.197..93.682 rows=4688 loops=1)
  Filter: (NOT (hashed SubPlan 1))
  Rows Removed by Filter: 9405
  Heap Fetches: 0
  SubPlan 1
    -> Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4) (actual time=0.161..70.865 rows=18163 loops=1)
        Filter: ((status)::text = 'Paid'::text)
        Rows Removed by Filter: 163627
  Planning Time: 6.063 ms
  Execution Time: 94.774 ms
(10 rows)
```

- Consulta 2: **GROUP BY** con **UNION ALL**

```
EXPLAIN
SELECT customerid
FROM (
    SELECT customerid
    FROM customers
    UNION ALL
    SELECT customerid
    FROM orders
    WHERE status = 'Paid'
) AS A
GROUP BY customerid
HAVING COUNT(*) = 1;
```

- Esta consulta combina dos conjuntos de datos con **UNION ALL**, agrupándolos en una tabla temporal llamada **A**.

- Luego, usa `GROUP BY` para contar cuántas veces aparece cada `customerid` en `A`.
- La condición `HAVING COUNT(*) = 1` filtra los `customerid` que sólo aparecen en `customers`.

Ejemplo de ejecución:

```

s12=# EXPLAIN ANALYZE SELECT customerid
FROM (
  SELECT customerid
  FROM customers
  UNION ALL
  SELECT customerid
  FROM orders
  WHERE status = 'Paid'
) AS A
GROUP BY customerid
HAVING COUNT(*) = 1;

                                QUERY PLAN
-----
Finalize GroupAggregate (cost=4425.26..4476.43 rows=1 width=4) (actual time=77.878..104.928 rows=4688 loops=1)
  Group Key: customers.customerid
  Filter: (count(*) = 1)
  Rows Removed by Filter: 9485
  -> Gather Merge (cost=4425.26..4471.93 rows=400 width=12) (actual time=77.869..100.601 rows=24167 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Sort (cost=3425.24..3425.74 rows=200 width=12) (actual time=34.173..35.415 rows=8056 loops=3)
      Sort Key: customers.customerid
      Sort Method: quicksort Memory: 797kB
      Worker 0: Sort Method: quicksort Memory: 108kB
      Worker 1: Sort Method: quicksort Memory: 1045kB
      -> Partial HashAggregate (cost=3415.59..3417.59 rows=200 width=12) (actual time=24.270..28.420 rows=8056 loops=3)
        Group Key: customers.customerid
        Batches: 1 Memory Usage: 1185kB
        Worker 0: Batches: 1 Memory Usage: 209kB
        Worker 1: Batches: 1 Memory Usage: 2081kB
        -> Parallel Append (cost=0.00..3383.01 rows=6516 width=4) (actual time=0.104..19.640 rows=10752 loops=3)
          -> Parallel Index Only Scan using customers_pkey on customers (cost=0.29..317.65 rows=8290 width=4) (actual time=0.080..1.617 rows=14093 loops=1)
            Heap Fetches: 0
          -> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=535 width=4) (actual time=0.114..15.150 rows=6054 loops=3)
            Filter: ((status)::text = 'Paid'::text)
            Rows Removed by Filter: 54542
Planning Time: 0.586 ms
Execution Time: 105.386 ms
(25 rows)

```

- Consulta 3: `EXCEPT`

```

EXPLAIN
SELECT customerid
FROM customers
EXCEPT
SELECT customerid
FROM orders
WHERE status = 'Paid';

```

- La consulta usa `EXCEPT`, que elimina los registros de la primera consulta (`customers`) que también están en la segunda (`orders WHERE status='Paid'`).
- Esta consulta podría beneficiarse del uso de índices, especialmente si hay índices en `customerid`.

Ejemplo de ejecución:

```

sl2=# EXPLAIN ANALYZE SELECT customerid
FROM customers
EXCEPT
SELECT customerid
FROM orders
WHERE status = 'Paid';

QUERY PLAN
-----
HashSetOp Except (cost=0.29..4597.59 rows=14093 width=8) (actual time=54.175..54.848 rows=4688 loops=1)
-> Append (cost=0.29..4560.09 rows=15002 width=8) (actual time=0.138..40.029 rows=32256 loops=1)
-> Subquery Scan on "**SELECT* 1" (cost=0.29..516.61 rows=14093 width=8) (actual time=0.138..4.322 rows=14093 loops=1)
-> Index Only Scan using customers_pkey on customers (cost=0.29..375.68 rows=14093 width=4) (actual time=0.135..2.698 rows=14093 loops=1)
    Heap Fetches: 0
-> Subquery Scan on "**SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.016..25.987 rows=18163 loops=1)
-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.015..24.125 rows=18163 loops=1)
    Filter: ((status)::text = 'Paid'::text)
    Rows Removed by Filter: 163627
Planning Time: 0.207 ms
Execution Time: 55.154 ms
(11 rows)

```

Apartado A

La

segunda consulta, que utiliza `UNION ALL` y `GROUP BY`, tiene una mayor probabilidad de devolver resultados parciales antes de finalizar completamente su ejecución. Esto se debe a que `UNION ALL` no elimina duplicados y, junto con el filtrado por `HAVING COUNT(*) = 1`, puede identificar y filtrar registros a medida que avanza, sin necesidad de procesar todo el conjunto de datos para comenzar a producir resultados parciales.

Apartado B

La

tercera consulta (que utiliza `EXCEPT`) es la que más se beneficia de la ejecución en paralelo. En PostgreSQL, `EXCEPT` es susceptible de paralelización, ya que se puede dividir la carga de ambas subconsultas y ejecutarlas en paralelo, especialmente si existen índices adecuados en las columnas comparadas (`customerid` en este caso). Esto contribuye a su menor tiempo de ejecución en comparación con las otras consultas.

Estudio del impacto de la generación de estadísticas:

Apartados A y B

Inicialización de la base de datos y carga de datos.

Apartado C

Ejecución de las consultas del Anexo 2 con Explain para estudiar su coste:

Consulta 1:

```

si2=# EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
               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)

```

cost=3507.17..3507.18 :

- El **cost** se refiere al costo estimado de la operación en términos de recursos computacionales (como CPU, I/O, etc.).
 - **3507.17** : costo estimado de **inicio** de la operación
 - **3507.18** : costo estimado **final** después de completar la operación

Consulta 2:

```

si2=# EXPLAIN SELECT count(*) FROM orders WHERE status = 'Shipped';
               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 = 'Shipped'::text)
(3 rows)

```

cost=3961.65..3961.66 :

- El **cost** se refiere al costo estimado de la operación en términos de recursos computacionales (como CPU, I/O, etc.).
 - **3961.65** : costo estimado de **inicio** de la operación
 - **3961.66** : costo estimado **final** después de completar la operación

Apartado D

Creemos un nuevo índice en la columna status de la tabla orders con el fin de optimizar las consultas que filtran por status:

```
CREATE INDEX idx_orders_status ON orders(status);
```

```

si2=# CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX
si2=# EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
               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)

si2=# EXPLAIN SELECT count(*) FROM orders WHERE status = 'Shipped';
               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 = 'Shipped'::text)
(3 rows)

```

Como podemos observar, al introducir el nuevo índice en la base de datos, si volvemos a ejecutar las mismas consultas, los costes de ejecución disminuyen de forma exponencial, pasando de `3507.17..3507.18` a `22.48..22.49` en el caso de la primera consulta y de `3961.65..3961.66` a `22.48..22.49` en la segunda. Esto se debe a que se cambia el recorrido secuencial por uno indexado.

Apartado F

Vamos a ejecutar la sentencia `ANALYZE` para generar estadísticas actualizadas sobre la tabla `orders` y así permitir que el planificador de consultas use datos más precisos:

```
ANALYZE orders;
```

Apartado G

Una vez que hemos ejecutado

`ANALYZE` para generar estadísticas de la tabla `orders`, el siguiente paso es ver cómo esto afecta a los planes de ejecución de las consultas del Anexo 2.

Volvemos a ejecutar las consultas tras el

```
ANALYZE :
```

```

si2=# ANALYZE orders;
ANALYZE
si2=# EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
               QUERY PLAN
-----
Aggregate  (cost=4.32..4.33 rows=1 width=8)
->  Index Only Scan using idx_orders_status on orders  (cost=0.29..4.31 rows=1 width=0)
    Index Cond: (status IS NULL)
(3 rows)

si2=# EXPLAIN SELECT count(*) FROM orders WHERE status = 'Shipped';
               QUERY PLAN
-----
Aggregate  (cost=2996.14..2996.15 rows=1 width=8)
->  Index Only Scan using idx_orders_status on orders  (cost=0.29..2676.66 rows=127792 width=0)
    Index Cond: (status = 'Shipped'::text)
(3 rows)

```

La ejecución proporciona información importante sobre cómo `ANALYZE` y el índice en `status` afectan el plan de ejecución para ambas consultas (`status IS NULL` y `status = 'Shipped'`). Vamos a realizar un estudio más detallado del resultado:

1. Ejecución de `ANALYZE`:

- Al ejecutar `ANALYZE`, generamos estadísticas para la tabla `orders`. Esto permite al optimizador de PostgreSQL tener información precisa sobre la distribución de datos en la columna `status`.

2. Plan de ejecución para `status IS NULL`:

- La consulta utiliza `Index Only Scan` gracias al índice `idx_orders_status`, que es muy eficiente. `Index Only Scan` es un tipo de escaneo que, si los datos requeridos están completamente en el índice (como en este caso), evita leer de la tabla y usa solo el índice.
- El `cost` es bastante bajo (4.32..4.33), lo cual refleja que el optimizador encontró una cantidad mínima de filas que cumplen `status IS NULL`, gracias a las estadísticas actualizadas.
- Conclusión:** Las estadísticas permiten al optimizador reconocer que pocas filas tienen `status IS NULL`, eligiendo `Index Only Scan` y manteniendo el costo bajo.

3. Plan de ejecución para `status = 'Shipped'`:

- Al igual que en la consulta anterior, el plan de ejecución elige `Index Only Scan` debido al índice en `status`.
- Sin embargo, el costo es mucho más alto (2996.14..2996.15), ya que el optimizador calcula que muchas filas cumplen la condición `status =`

'Shipped'. Esto indica que el valor 'Shipped' es mucho más frecuente en la columna `status`.

- **Conclusión:** Aunque la consulta sigue utilizando el índice, el alto costo refleja una mayor cantidad de filas que cumplen la condición. Esto es un buen ejemplo de cómo las estadísticas permiten al optimizador ajustar el plan de ejecución según la frecuencia de cada valor en la columna `status`.

Apartado H

Este apartado consiste en ejecutar

`EXPLAIN` para las dos consultas adicionales del Anexo 2 (`status = 'Paid'` y `status = 'Processed'`), ahora que el índice y las estadísticas ya están creados.

Consultas ejecutadas:

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

Resultado de la ejecución:

```
si2=# EXPLAIN SELECT count(*) FROM orders WHERE status = 'Paid';
               QUERY PLAN
-----
Aggregate  (cost=426.05..426.06 rows=1 width=8)
->  Index Only Scan using idx_orders_status on orders  (cost=0.29..380.83 rows=18088 width=0)
      Index Cond: (status = 'Paid'::text)
(3 rows)

si2=# EXPLAIN SELECT count(*) FROM orders WHERE status = 'Processed';
               QUERY PLAN
-----
Aggregate  (cost=842.49..842.50 rows=1 width=8)
->  Index Only Scan using idx_orders_status on orders  (cost=0.29..752.72 rows=35910 width=0)
      Index Cond: (status = 'Processed'::text)
(3 rows)
```

1. Consulta con `status = 'Paid'`:

- **Plan de Ejecución:** La consulta utiliza `Index Only Scan` gracias al índice `idx_orders_status`.
- **Coste:** El coste total es de `426.05`, mucho menor en comparación con la consulta que usa `status = 'Shipped'` en el paso anterior (que tenía un coste de 2996.14). Esto sugiere que la cantidad de filas que cumplen `status = 'Paid'` es significativamente menor que las que cumplen `status = 'Shipped'`.

- **Filas estimadas:** El optimizador estima que aproximadamente `18,088` filas cumplen la condición `status = 'Paid'`.
- **Análisis:** Este resultado muestra que el índice sigue siendo efectivo, y las estadísticas ayudan a PostgreSQL a entender que `'Paid'` es menos común que `'Shipped'`, ajustando así el costo.

2. Consulta con `status = 'Processed'`:

- **Plan de Ejecución:** Igual que las consultas anteriores, utiliza `Index Only Scan`, aprovechando el índice `idx_orders_status`.
- **Coste:** El coste total es de `842.49`, intermedio entre el costo de `status = 'Paid'` y el de `status = 'Shipped'`. Esto indica que `'Processed'` es más frecuente que `'Paid'` pero menos frecuente que `'Shipped'`.
- **Filas estimadas:** La estimación es de `35,910` filas con `status = 'Processed'`, lo cual representa una carga de datos más alta que la de `'Paid'`.
- **Análisis:** De nuevo, las estadísticas permiten que el optimizador determine con precisión la frecuencia relativa de cada valor en `status`. Esto explica por qué los costos de consulta son menores cuando el valor es menos frecuente (`Paid`) y mayores cuando es más común (`Processed` y, aún más, `Shipped`).

Apartado I

Este apartado consiste en crear un script SQL que contenga todas las consultas y sentencias que has ejecutado en los pasos anteriores, en el orden correcto.

El contenido del fichero `countStatus.sql` es el siguiente:

```
-- Consulta iniciales sobre base de datos sin índices ni esta
EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
EXPLAIN SELECT count(*) FROM orders WHERE status = 'Shipped';

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

-- Consulta después de crear el índice, pero sin estadísticas
EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
```



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

-- Genera estadísticas de la tabla orders
ANALYZE orders;

-- Consultas después de generar las estadísticas
EXPLAIN SELECT count(*) FROM orders WHERE status IS NULL;
EXPLAIN SELECT count(*) FROM orders WHERE status = 'Shipped';

-- Consultas adicionales para comparar después de tener índice
EXPLAIN SELECT count(*) FROM orders WHERE status = 'Paid';
EXPLAIN SELECT count(*) FROM orders WHERE status = 'Processed';
```

Fin de la memoria, espero que la haya disfrutado tanto como a nosotros realizarla 😊