Sistemas Informáticos. Curso 2024/2025

Escuela Politécnica Superior. Universidad Autónoma de Madrid

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

Miguel Jesús Paterson González y Blanca Matas Gavira

Índice

1. Rediseño de la Base de Datos

Introducción

En la práctica se partió de una base de datos que contenía información sobre películas y ventas, organizada en varias tablas relacionadas. El rediseño de la base de datos tuvo como principal objetivo mejorar su estructura y normalización, garantizando la integridad de los datos y optimizando las consultas para satisfacer los requisitos de la práctica.

Es importante, saber que en nuestro **docker-compose.yml** vamos a ejecutar la base de datos desde el puerto 5432 (puerto por defecto para postgresql) por lo que, si se tiene instalado en el sistema host el postgresql, se deberá de hacer una parada del mismo, si está en ubuntu puede utilizar el siguiente comando:

```
sudo systemctl stop postgresql
```

En caso de que quiera iniciarlo de nuevo, solo debe de ejecutar:

```
sudo systemctl restart postgresql
```

Cambios Realizados

1.1 Eliminación de Atributos Multivaluados

En la base de datos original, se encontraron varios atributos multivaluados, como el género de las películas, los idiomas y los países. Para resolver esto, se realizaron las siguientes modificaciones:

- Se añadieron columnas de género (**genres**), idioma (**languages**) y país (**countries**) directamente en la tabla **imdb_movies**. Estas columnas ahora almacenan la información necesaria como texto separado por comas, evitando así la complejidad de manejar relaciones adicionales para atributos multivaluados en este caso particular.

1.2 Incorporación de Restricciones y Relaciones

Para garantizar la integridad de los datos y mejorar la robustez del modelo, se añadieron varias restricciones y claves foráneas. Los cambios fueron:

- **Claves Primarias y Foráneas**: Se establecieron claves primarias y foráneas donde no existían, para garantizar la unicidad de las filas y su correcta relación.
- **Cambios en Cascada**: Se aplicaron acciones en cascada (**ON DELETE CASCADE**) en aquellas relaciones que requerían garantizar que, al eliminar un registro, se actualizaran o eliminaran los registros dependientes. Esto fue crucial para tablas como **orderdetail** y **likes**, que dependen de otras entidades como **orders** y **customers**.

1.3 Nuevas Funcionalidades

Se incorporaron nuevas funcionalidades para ampliar la base de datos según los requisitos de la práctica:

- **Saldo de Clientes**: Se añadió la columna **balance** en la tabla **customers** para gestionar el saldo de cada cliente. Se implementó un procedimiento almacenado **setCustomersBalance()** para asignar un saldo aleatorio a cada cliente existente.
- **Sistema de Likes**: Se creó la tabla **likes** para almacenar las valoraciones de los usuarios sobre las películas. La tabla tiene una clave primaria compuesta por **customer_id** y **movie_id**, para evitar duplicidades y garantizar que un cliente solo pueda valorar cada película una vez. Además, se consideró la posibilidad de que un usuario pueda modificar o eliminar su valoración.
- **Modificación del Tamaño de Contraseña**: Para adaptarse a nuevos requerimientos de seguridad, se modificó la columna **password** en la tabla **customers** para permitir contraseñas más largas, cambiando el tipo de **VARCHAR(128)** a **VARCHAR(512)**.

Documentación de los Scripts de actualización

Todos los cambios realizados en el rediseño se implementaron en varios scripts SQL, cada uno con un propósito específico:

actualiza.sql

Este script realiza modificaciones importantes en el diseño original de la base de datos:

- **Añadir la columna balance a la tabla customers**: Esta columna se agregó para gestionar el saldo de cada cliente y poder pagar los pedidos.
- **Crear la tabla likes**: Almacena las valoraciones de los usuarios a las películas, asegurando que un cliente pueda valorar una película solo una vez.
- **Modificación del tamaño de la columna password en customers**: Se incrementó la longitud para permitir contraseñas más largas y seguras.
- **Agregar restricciones y claves foráneas**: Para garantizar la integridad referencial, se añadieron restricciones y claves foráneas en varias tablas, como **orders**, **orderdetail**, **inventory**, y **imdb_actormovies**.
- **Creación de índices**: Para reducir el coste de ejecución del resto de scripts y algunas consultas.

actualizaPrecios.sql

Este script tiene como objetivo calcular y actualizar el precio total de los pedidos:

- **Función updateOrderTotal(id_order)**: Actualiza el precio de cada detalle del pedido en función del precio de los productos actuales, recalculando el total con impuestos.
- **Función updateAllOrdersTotalBatch(batch_size)**: Procesa todos los pedidos por lotes de 500, utilizando la función anterior para mantener los totales actualizados.

Al no hacerlo con un update y hacerlo con un bucle tarda más. Para solucionar eso se utilizan los siguientes índices:

```
CREATE INDEX IF NOT EXISTS idx_orders_customerid ON orders (customerid);
CREATE INDEX IF NOT EXISTS idx_orderdetail_orderid ON orderdetail (orderid);
CREATE INDEX IF NOT EXISTS idx_orderdetail_prodid ON orderdetail (prod_id);
```

actualizaTablas.sql

Este script se enfoca en corregir y mejorar la estructura de las tablas:

- **Eliminación de tablas no necesarias** como **imdb_moviegenres**, **imdb_movielanguages**, y **imdb_moviecountries**, consolidando esta información en columnas nuevas dentro de **imdb_movies**.
- **Creación de índices y modificación de columnas**: Añade índices para optimizar consultas y modifica tipos de columnas sin cambiar necesariamente el tipo a **BOOLEAN**.

actualizaCarrito.sql

Este script se encarga de crear un trigger para mantener la información de los carritos de compra:

- **Trigger actualizaCarrito**: Se ejecuta cuando se añaden, actualizan o eliminan artículos del carrito (**orders**), y se asegura de que la información del pedido se actualice correctamente.

pagado.sql

Este script se centra en el proceso de pago de un pedido:

- **Trigger pagado**: Actualiza el estado del pedido a **Paid**, descontando el saldo del cliente y actualizando el stock del inventario.

Conclusión del Rediseño

El proceso de rediseño de la base de datos permitió normalizar y mejorar la estructura original, haciéndola más eficiente y fácil de mantener. Las modificaciones realizadas aseguran la integridad de los datos y proporcionan una base sólida para la integración posterior con la API en Python y la optimización de consultas.

Finalmente, el diagrama final de nuestra base de datos se encuentra en el Anexo (***diagrama base de datos 1***) para entender el rediseño de una manera más visual.

2. Integración con Python de la Base de Datos

Introducción

En esta sección de la memoria se detalla cómo se llevó a cabo la integración de la base de datos con Python utilizando una API REST. Esta integración se implementó para permitir que los usuarios interactúen con la base de datos de manera intuitiva y sencilla, facilitando operaciones como la gestión de carritos de compra, la consulta de productos y la realización de pedidos. Además, se utilizó Docker para contenerizar tanto la API como la base de datos, lo que simplifica la gestión y el despliegue de los servicios.

Conexión a la Base de Datos

La conexión a la base de datos se realiza mediante SQLAlchemy y se configura utilizando el URL de la base de datos en el archivo **api.py**. Dado que estamos utilizando Docker, la conexión no se realiza al **localhost** directamente, sino a un servicio llamado **db**, que se define en el archivo **docker-compose.yml**. De esta manera, garantizamos que la API se pueda conectar correctamente a la base de datos ya que se usa el atributo **depends_on** para asegurarse de que el servicio de la base de datos esté activo antes de levantar la API.

El URL de la conexión se define de la siguiente manera:

```
DATABASE_URL = "postgresql://alumnodb:1234@db:5432/si1"
```

Este URL sigue el formato:

```
"postgresql://<usuariodb>:<contraseñadb>@<host>:<puerto>/<nombre_de_base_de_datos"
```

En este caso:

- **usuario**: alumnodb
- **contraseña**: 1234
- **host**: db (especificado en el **docker-compose.yml**)
- **puerto**: 5432 (puerto predeterminado de PostgreSQL)
- **nombre_de_base_de_datos**: si1

Endpoints Definidos en api.py

El archivo **api.py** contiene múltiples endpoints que permiten la interacción del usuario con la base de datos a través de la API REST para que un usuario pueda añadir, eliminar, editar o comprar productos de su carrito.

Antes de definir las funciones, hay que tener claro que hemos entendido por **carrito** (la base del pago de los clientes). El **carrito** es un order que es único, es el único order que tiene el status en **Pending** porque no observamos el sentido de que haya más de un carrito abierto porque no deberias de abrir otro hasta pagar el anterior. Esto es porque hemos deducido que los estados por los que pasa un pedido son: **Pending** cuando aún no se ha pagado, **Paid** cuando ya se ha pagado con éxito, **Processed** cuando está preparándose el envío y **Shipped** una vez se ha enviado. A continuación se describe cada endpoint y su función:

- **/login (POST):**
  Permite a un usuario iniciar sesión proporcionando su nombre de usuario y contraseña. Si las credenciales son correctas, devuelve un **customer_id** para autenticar futuras solicitudes.

- **/add_balance (POST):**
  Añade saldo a la cuenta de un usuario autenticado. Se verifica que las credenciales sean correctas antes de realizar la operación.

- **/add_to_cart (POST):**
  Añade un producto al carrito del usuario. Comprueba primero si el usuario tiene credenciales válidas y luego si hay suficiente stock disponible antes de realizar la acción.

- **/pay_order (POST):**
  Realiza el pago de un pedido pendiente del usuario. Se verifica que el usuario tenga saldo suficiente antes de marcar el pedido como pagado.

- **/view_cart (GET):**
  Muestra el contenido del carrito del usuario con los productos agregados, incluyendo la cantidad y el precio total de cada línea del carrito.

- **/remove_from_cart (POST):**
  Elimina un producto específico del carrito del usuario.

- **/edit_cart (POST):**
  Permite editar la cantidad de un producto en el carrito del usuario.

- **/get_orders (GET):**
  Devuelve el historial de pedidos del usuario, incluyendo detalles como la fecha del pedido, el estado y el monto total.

- **/get_order_details (GET):**
  Proporciona los detalles de un pedido específico, incluyendo los productos y sus cantidades.

- **/get_order_status (GET):**
  Devuelve el estado actual de un pedido específico.

- **/get_balance (GET):**
  Proporciona el saldo actual del usuario.

- **/change_password (POST):**
  Permite al usuario cambiar su contraseña proporcionando la contraseña actual y la nueva.

- **/create_user (POST):**
  Crea un nuevo usuario en la base de datos con los detalles proporcionados como nombre de usuario, dirección, correo electrónico y tarjeta de crédito.

- **/delete_user (POST):**
  Elimina un usuario de la base de datos tras verificar sus credenciales.

- **/get_available_products (GET):**
  Devuelve una lista de productos disponibles para su compra, incluyendo su ID, título, precio, stock y descripción.

- **/get_available_products_filter (GET):**
  Devuelve los productos filtrados por idioma y género, según los parámetros especificados por el usuario.

- **/get_product_info (GET):**
  Devuelve la información de un producto específico basado en su **prod_id**. Incluye el título de la película, precio, stock, y descripción.

- **/get_movie_info (GET):**
  Devuelve la información de una película específica, incluyendo su título, director, año, tipo, país, género y el idioma.

- **/get_languages (GET):**
  Proporciona una lista de todos los idiomas distintos disponibles en las películas.

- **/get_genres (GET):**
  Proporciona una lista de todos los géneros distintos disponibles en las películas.

Funciones Probadas en client.py

El archivo **client.py** fue utilizado como un cliente estático para probar los diferentes endpoints de la API. Las funciones se implementan en secuencia para simular el flujo completo de compra de un usuario. Las principales acciones realizadas fueron:

1. **Crear un Usuario**: Se crea un usuario nuevo con información básica como nombre de usuario, dirección, correo electrónico y tarjeta de crédito.

2. **Inicio de Sesión**: El usuario inicia sesión proporcionando su nombre de usuario y contraseña para obtener un **customer_id** que se utilizará en las futuras transacciones.

3. **Añadir Saldo**: Se añade saldo a la cuenta del usuario para asegurarse de que tiene fondos suficientes para realizar compras. Y después se obtiene el saldo que tiene en la cuenta.

4. **Buscar Productos Disponibles Filtrados por Idioma y Género**: Se realiza una búsqueda de productos disponibles, filtrando por las preferencias del usuario.

5. **Añadir Productos al Carrito**: Se añaden múltiples productos al carrito con las cantidades especificadas.

6. **Ver el Contenido del Carrito**: Se visualiza el contenido del carrito actual, mostrando los productos agregados y el costo total de cada uno.

7. **Editar Cantidades en el Carrito**: Se modifica la cantidad de un producto ya presente en el carrito para probar la funcionalidad de actualización.

8. **Eliminar un Producto del Carrito**: Se elimina un producto del carrito y se visualiza nuevamente para confirmar el cambio.

9. **Verificar el Saldo Disponible**: Antes de proceder al pago, se comprueba que el usuario tenga saldo suficiente.

10. **Pagar el pedido**: Se realiza el pago del pedido, lo cual cambia el estado del pedido a "Pagado" y descuenta el saldo del usuario.

11. **Verificar el Historial de Pedidos**: Se verifica el historial de pedidos para confirmar que el pedido ha sido realizado exitosamente.

12. **Cambiar la contraseña:** Se cambia la contraseña del usuario y se actualiza la variable de contraseña que se usará en el resto de pruebas, en caso de que las siguientes operaciones no den correctas, se verifica que falla al cambiar la contraseña.

13. **Verificar el saldo después del pago:** Verifica que el saldo ha cambiado tras pagar el carito.

14. **Obtener información de una película:** Imprime por pantalla todos los datos de una película por pantalla.

15. **Listar idiomas**: Lista todos los idiomas y los imprime por pantalla para que luego puedas filtrar por idioma.

16. **Listar géneros:** Lista todos los géneros y los imprime por pantalla para que luego puedas filtrar por género.
17. **Listar todos los productos disponibles:** Lista todos los productos disponibles, por defecto está comentado porque ocupa toda la pantalla y no deja ver el resto de resultados.
18. **Eliminar usuario:** Elimina el usuario creado para no alterar la base de datos y se verifica que esté eliminado haciendo otro log in (debe dar error).

En resumen, **client.py** además de tener todas las funcionalidades que nos permiten conectarnos al servicio que se conecta a la base de datos, hemos creado un main para que

se pueda usar como una prueba completa para el flujo de compra, desde la creación del usuario hasta el pago de los pedidos, verificando la interacción entre la API y la base de datos.

3. Optimización

Estudio del impacto de un índice:

a. Crear una consulta.

Para este estudio, se nos pedía una consulta que muestre el número de estados (columna state) distintos con clientes que tienen pedidos en un año dado usando el formato YYYY (por ejemplo 2017) y que además pertenecen a un país (country) determinado, por ejemplo, Perú. El diseño de consulta que hicimos fue este, utilizamos como referencia los datos propuestos (2017, Perú) y vamos a probar el impacto de los índices en esta query.

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

b. Estudiar el plan de ejecución de la consulta.

Al ejecutar la sentencia **EXPLAIN** sobre la consulta inicial, obtuvimos el siguiente plan de ejecución:

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

El plan de ejecución inicial muestra una exploración secuencial paralela en la tabla **orders** para filtrar los pedidos del año 2017, lo cual representa un coste significativo (3291.03), y una exploración secuencial en la tabla **customers**.

c. Identificar un índice que mejore el rendimiento de la consulta.

Identificamos la necesidad de crear índices que ayudaran a mejorar el rendimiento, especialmente enfocados en los filtros por **country** y la relación entre orders y **customers**. Los índices creados fueron los siguientes:

1. **idx_customers_country**

```
CREATE INDEX IF NOT EXISTS idx_customers_country ON customers (country);
```

Tras crear este índice, el plan de ejecución mostró una mejora significativa al reemplazar la exploración secuencial por un Bitmap Heap Scan y Bitmap Index Scan sobre la columna country. El coste de la consulta se redujo a:

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

2. **idx_orders_customerid_orderdate**

```
CREATE INDEX IF NOT EXISTS idx_orders_customerid_orderdate ON
orders (customerid, orderdate);
```

Tras la creación de este índice, el plan de ejecución cambió a un Index Only Scan, logrando una mejora significativa en el rendimiento al permitir que se pudiera hacer un acceso más eficiente tanto a la relación con **customerid** como al filtrado por **orderdate**. El coste de la consulta se redujo a:

```
Aggregate  (cost=2688.75..2688.76 rows=1 width=8)
  ->  Nested Loop  (cost=0.42..2688.74 rows=5 width=118)
        ->  Seq Scan on customers c  (cost=0.00..528.16 rows=70
width=122)
```

```
              Filter: ((country)::text = 'Peru'::text)
        ->  Index Only Scan using idx_orders_customerid_orderdate
on orders o  (cost=0.42..30.82 rows=5 width=4)
              Index Cond: (customerid = c.customerid)
              Filter: (EXTRACT(year FROM orderdate) =
'2017'::numeric)
```

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

- Inicialmente, la consulta realizaba exploraciones secuenciales costosas tanto en **orders** como en **customers**, con un coste total de **4821.98**.
- La creación del índice **idx_customers_country** ayudó a reducir el coste relacionado con el filtrado por **country**, disminuyendo el coste total a **4473.65**.
- El índice compuesto **idx_orders_customerid_orderdate** resultó ser la optimización más efectiva, permitiendo un **Index Only Scan** que redujo el coste total de la consulta a **2688.75**.

Con la implementación de ambos índices, el coste de la consulta se reduce a:

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

Con esto, vemos que lo más óptimo para esta query es la creación de los 2 índices y es lo que añadiremos a la query.

e. Probar distintos índices y discutir los resultados.

- Inicialmente, la consulta realizaba exploraciones secuenciales costosas tanto en **orders** como en **customers**, con un coste total de **4821.98**.
- Pensamos que el problema podía estar en crear un índice en orders sobre customerid, pero vimos que no había mejoría y que sería con el mismo coste total.
- La creación del índice **idx_customers_country** ayudó a reducir el coste relacionado con el filtrado por **country**, disminuyendo el coste total a **4473.65**.

- El índice compuesto **idx_orders_customerid_orderdate** resultó ser la optimización más efectiva, permitiendo un **Index Only Scan** que redujo el coste total de la consulta a **2688.75**.
- Finalmente, la implementación de ambos índices reduce el coste total de la consulta a **2340.43**.

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

En esta sección, se estudia el impacto de diferentes alternativas para una consulta que busca identificar a los clientes que no tienen pedidos con el estado 'Paid'. Se presentan tres formas de realizar esta consulta y se analiza el plan de ejecución de cada una de ellas, comparando los resultados. Se han implementado las consultas ejecutadas en el script anexo1.sql

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

1. Usando **NOT IN**:

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

Resultado:

- Esta consulta devuelve los **customerid** de los clientes que no tienen pedidos con el estado 'Paid'.
- **Plan de Ejecución**:

```
Index Only Scan using customers_pkey on customers
(cost=3961.93..4372.56 rows=7046 width=4)
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    ->  Seq Scan on orders  (cost=0.00..3959.38
rows=909 width=4)
          Filter: ((status)::text = 'Paid'::text)
```

- **Análisis**: La consulta realiza un **Index Only Scan** en la tabla **customers** y una exploración secuencial (**Seq Scan**) en la tabla **orders**. La operación **NOT IN** resulta costosa debido a la necesidad de verificar cada cliente en la subconsulta.

2. Usando **UNION ALL** y **HAVING**

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

Resultado:

- Esta consulta también devuelve los **customerid** de los clientes que no tienen pedidos con el estado 'Paid'.
- **Plan de Ejecución**:

```
Finalize GroupAggregate  (cost=4425.26..4476.43 rows=1
width=4)
  Group Key: customers.customerid
  Filter: (count(*) = 1)
  ->  Gather Merge  (cost=4425.26..4471.93 rows=400
width=12)
        Workers Planned: 2
        ->  Sort  (cost=3425.24..3425.74 rows=200
width=12)
              Sort Key: customers.customerid
              ->  Partial HashAggregate
(cost=3415.59..3417.59 rows=200 width=12)
                    Group Key: customers.customerid
                    ->  Parallel Append
(cost=0.00..3383.01 rows=6516 width=4)
                          ->  Parallel Index Only Scan
using customers_pkey on customers  (cost=0.29..317.65
rows=8290 width=4)
                          ->  Parallel Seq Scan on
orders  (cost=0.00..3023.69 rows=535 width=4)
                                Filter: ((status)::text
= 'Paid'::text)
```

- **Análisis**: Esta consulta se beneficia de la ejecución en paralelo, ya que utiliza **Parallel Append**, lo que permite un mejor rendimiento al aprovechar varios procesos para escanear las tablas **customers** y **orders**. Sin embargo, el coste sigue siendo elevado debido a la agregación y ordenación de los datos.

3. Usando **EXCEPT**

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

Resultado:

- Esta consulta devuelve los **customerid** de los clientes que no tienen pedidos con el estado 'Paid'.
- **Plan de Ejecución**:

```
HashSetOp Except  (cost=0.29..4597.59 rows=14093
width=8)
  ->  Append  (cost=0.29..4560.09 rows=15002 width=8)
        ->  Subquery Scan on "*SELECT* 1"
(cost=0.29..516.61 rows=14093 width=8)
              ->  Index Only Scan using customers_pkey
on customers  (cost=0.29..375.68 rows=14093 width=4)
        ->  Subquery Scan on "*SELECT* 2"
(cost=0.00..3968.47 rows=909 width=8)
              ->  Seq Scan on orders
(cost=0.00..3959.38 rows=909 width=4)
                    Filter: ((status)::text =
'Paid'::text)
```

- **Análisis**: La consulta con **EXCEPT** utiliza un operador de conjunto (**HashSetOp Except**), lo que permite eliminar los valores coincidentes entre ambas subconsultas. Este enfoque tiene un rendimiento relativamente alto en comparación con **NOT IN**, pero no se beneficia tanto de la paralelización como la versión con **UNION ALL** y **HAVING**.

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

La consulta con **NOT IN** es la que empieza a devolver resultados de forma más rápida, ya que filtra directamente sobre la tabla **customers** sin necesidad de unificar resultados como en las otras consultas.

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

La consulta que utiliza **UNION ALL** y **HAVING** es la que mejor se beneficia de la ejecución en paralelo, lo cual se refleja en el uso de **Parallel Append** y **Gather Merge**, distribuyendo la carga de trabajo entre varios procesos.

Estudio del impacto de la generación de estadísticas:

a. Estudio de los Planes de Ejecución Iniciales.

Se realizaron las siguientes consultas para obtener el número de pedidos con estado **NULL** y con estado **'Shipped'**:

1. Consulta para status is **NULL**:

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

Plan de Ejecución:

```
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)
```

El plan de ejecución inicial muestra un escaneo secuencial (**Seq Scan**) sobre la tabla **orders**, lo cual resulta en un coste elevado debido a la falta de índices para optimizar la búsqueda.

2. Consulta para status = **'Shipped'**:

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

Plan de Ejecución:

```
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)
```

De manera similar a la consulta anterior, se realiza un escaneo secuencial, lo cual implica un coste elevado.

b. Creación de un índice en la columna status.

Para mejorar el rendimiento de estas consultas, se creó un índice sobre la columna **status** en la tabla **orders**:

```
CREATE INDEX IF NOT EXISTS idx_orders_status ON orders (status);
```

c. Estudio de los nuevos planes de ejecución.

Tras la creación del índice, se volvieron a ejecutar ambas consultas para estudiar cómo cambiaba el plan de ejecución.

1. Plan de ejecución para status is **NULL**:

```
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)
```

Al utilizar un **Index Only Scan**, se reduce significativamente el coste de ejecución. Esto muestra cómo el índice mejora la eficiencia al evitar un escaneo completo de la tabla reduciendo el coste a **22.49**.

2. Plan de ejecución para status = **'Shipped'**:

```
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)
```

Al igual que en la anterior, la utilización del **Index Only Scan** reduce notablemente el coste de la consulta dejándolo en un coste de **22.49** también.

d. Generación de estadísticas con ANALYZE.

Se ejecutó la siguiente sentencia para generar estadísticas sobre la tabla orders:

```
ANALYZE orders;
```

e. Estudio de los costes tras generar estadísticas.

Tras generarlas estadísticas, se volvieron a ejecutar ambas consultas para estudiar cómo cambiaba el plan de ejecución.

1. Plan de ejecución para status is **NULL**:

```
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)
```

El coste se reduce significativamente a **4.32**, lo cual indica que el optimizador tiene ahora un mejor conocimiento de la distribución de valores en la columna **status**, y sabe que hay pocas filas con el valor **NULL**.

2. Plan de ejecución para status = **'Shipped'**:

```
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)
```

Aunque se sigue utilizando el índice, el coste es alto (**2983.54**) debido a la gran cantidad de filas con el valor **'Shipped'**. Tras la generación de estadísticas con **ANALYZE**, el optimizador obtiene información detallada sobre la frecuencia de los valores en la columna **status**, y detecta que

'Shipped' es un valor muy frecuente. Esto significa que hay muchas filas con ese valor, lo cual incrementa el coste de la consulta, ya que se deben procesar más filas para obtener el resultado.

f. Comparación con otras consultas.

Se evaluaron otras dos consultas tras la generación de las estadísticas para comparar sus planes de ejecución con los anteriores. Esto permite entender mejor cómo las estadísticas afectan a las decisiones del optimizador de PostgreSQL en función de la distribución de los valores en la columna status.

1. Consulta para status = **'Paid'**:

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

Plan de Ejecución:

```
Aggregate  (cost=434.05..434.06 rows=1 width=8)
  ->  Index Only Scan using idx_orders_status on orders
(cost=0.29..387.83 rows=18488 width=0)
        Index Cond: (status = 'Paid'::text)
```

Utiliza el índice con un coste moderado (**434.05**), lo cual refleja una menor cantidad de filas con el valor **'Paid'**. Las estadísticas han permitido al optimizador ajustar las estimaciones de filas, lo que se traduce en un plan de ejecución más eficiente en comparación con los valores que tienen alta repetición.

2. Consulta para status = **'Processed'**:

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

Plan de Ejecución:

```
Aggregate  (cost=862.23..862.24 rows=1 width=8)
  ->  Index Only Scan using idx_orders_status on orders
(cost=0.29..770.49 rows=36697 width=0)
        Index Cond: (status = 'Processed'::text)
```

El coste es de **862.23**, lo cual indica que hay una cantidad considerable de filas con el valor **'Processed'**. Sin embargo, el coste sigue siendo significativamente menor que el de **'Shipped'**, lo cual sugiere que la frecuencia de **'Processed'** es menor. El índice y las estadísticas ayudan a reducir el esfuerzo de escaneo, pero la alta cantidad de filas afecta la eficiencia.

g. Conclusión.

El impacto de generar estadísticas mediante **ANALYZE** se observa claramente en los planes de ejecución de las consultas. Antes de generar las estadísticas, el optimizador no tenía información precisa sobre la distribución de valores en la columna **status**. Esto se traduce en estimaciones incorrectas de costes, y en algunos casos se subestima el esfuerzo necesario para consultar valores frecuentes, como **'Shipped'**.

Tras ejecutar **ANALYZE**, el optimizador tiene un conocimiento mucho más preciso de la distribución de los datos, lo cual le permite tomar decisiones más informadas sobre cómo ejecutar las consultas de la manera más eficiente posible. Sin embargo, para valores con alta repetición (como **'Shipped'**), el coste aumenta debido a la cantidad de filas involucradas, aunque el índice sigue siendo útil para evitar un escaneo secuencial completo de la tabla.

Por otro lado, para valores menos frecuentes, como **NULL** o **'Paid'**, el coste se reduce drásticamente gracias al uso del índice y la correcta estimación de las filas involucradas. Esto muestra cómo la combinación de índices y estadísticas optimiza el rendimiento de las consultas.

Anexo:

Diagrama final de la Base de Datos (1)