SISTEMAS INFORMÁTICOS I
PRÁCTICA 2

AUTORES
CARLOS GARCÍA SANTA
EDUARDO JUNOY ORTEGA

SISTEMAS INFORMÁTICOS
GRUPO1321: PAREJA 05

INGENIERÍA INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD AUTÓNOMA DE MADRID

19/11/2023

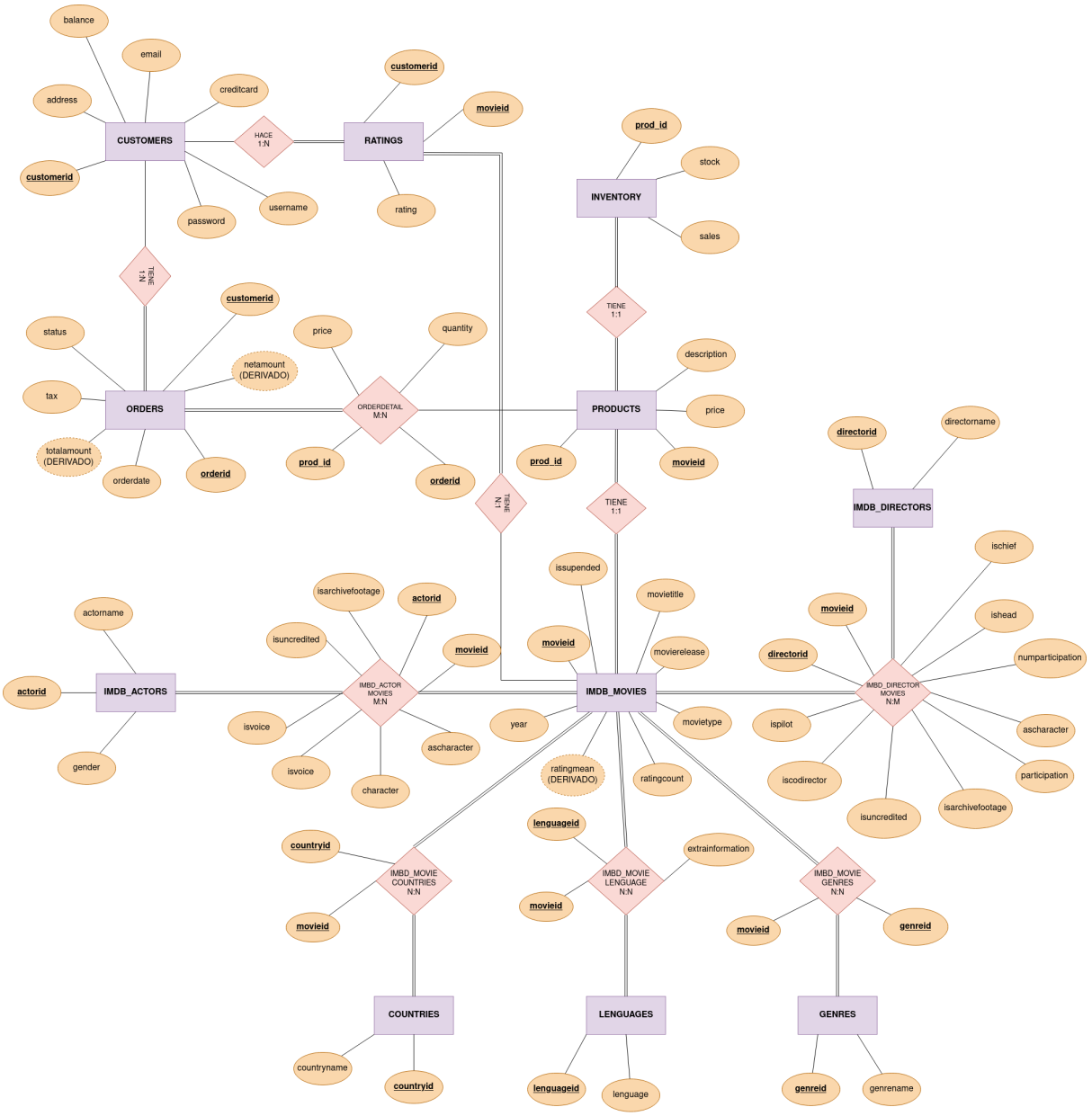
Universidad Autónoma de Madrid

Índice

Introducción

Esta memoria documenta los conocimientos desarrollados y aplicados en la segunda práctica. Esta se ha centrado en explorar las prestaciones y posibilidades avanzadas que provee un sistema gestor de base de datos a través de: el manejo y gestión de bases de datos, programación de consultas, el uso de funciones y la optimización de consultas con el uso de índices y comandos.

Parte 1: Diseño de la BD



1. Rediseño de la base de datos

Adición de Claves Foráneas:

Tablas imdb_actormovies, inventory, orderdetail, y orders:

Se añadieron restricciones de clave foránea (FOREIGN KEY) a estas tablas para establecer relaciones con las tablas referenciadas (**imdb_actors, imdb_movies, products, orders, customers**). Esto asegura la integridad

referencial, es decir, que no se puedan insertar datos en estas tablas que no tengan correspondencia en las tablas a las que hacen referencia.

Normalización de Datos y Creación de Nuevas Tablas:

Creación de las tablas countries, languages, y genres:

Estas tablas se crean para almacenar información única sobre países, idiomas y géneros, respectivamente. Cada una tiene un id como clave primaria y un campo para el nombre, asegurando que los nombres sean únicos y no nulos.

- Se extraen datos de las columnas **country**, **language**, y **genre** de las tablas **imdb_moviecountries**, **imdb_movielanguages**, y **imdb_moviegenres**, respectivamente, y se insertan en las nuevas tablas creadas.
- Se añaden nuevas columnas (**countryid**, **languageid**, **genreid**) a las tablas **imdb_moviecountries**, **imdb_movielanguages**, y **imdb_moviegenres**.
- Se establecen relaciones de clave foránea con las nuevas tablas (**countries**, **languages**, **genres**).
- Se actualizan las tablas originales para que el id correspondiente reemplace el nombre del país, idioma o género.
- Las columnas originales **country**, **language**, y **genre** se eliminan de las tablas **imdb_moviecountries**, **imdb_movielanguages**, y **imdb_moviegenres**.

Modificación de la Tabla customers

Se añade un nuevo campo **balance** de tipo NUMERIC a la tabla **customers**, este campo está destinado a almacenar el saldo de cada cliente.

Creación de la Tabla ratings

Se crea una nueva tabla **ratings** para almacenar las valoraciones que los clientes dan a las películas, la tabla tiene los campos **movieid**, **customerid**, y **rating**. Se establece una restricción CHECK en rating para asegurar que los valores estén entre 1 y 10, y se añaden claves foráneas (fk_movie, fk_customerid) para vincular **movieid** y **customerid** con las tablas **imdb_movies** y customers, respectivamente, además se establece una restricción UNIQUE en la combinación (movieid, customerid) para evitar que un mismo cliente valore dos veces la misma película.

Modificaciones en la Tabla imdb_movies

Se añaden dos nuevos campos a la tabla **imdb_movies**: **ratingmean** (valoración media) y **ratingcount** (número de valoraciones), estos campos están

destinados a almacenar la valoración media y el total de valoraciones recibidas para cada película.

Cambio en la Tabla customers

Se altera el tipo de dato del campo **password** a CHARACTER VARYING(96).

Creación de Procedimiento Almacenado

Se crea un procedimiento almacenado llamado **setCustomersBalance** que inicializa el campo balance de la tabla customers con un valor aleatorio entre 0 y un valor máximo especificado (initialBalance).

2. Funciones

setPrice.sql: Esta consulta actualiza la tabla **orderdetail** ajustando el precio de cada orden basándose en cuantos años han pasado desde la fecha del pedido hasta el año 2022 (último año que se ha encontrado en orderdate, que se considera como el actual), para pedidos realizados antes de 2022. Se utiliza el precio actual del producto de la tabla **products** y disminuye este precio en un 2% por cada año transcurrido, esto se logra mediante la función **POWER(0.98, (2022 - EXTRACT(YEAR FROM o.orderdate)))**, el resultado se multiplica por el precio actual del producto y se redondea a dos decimales.

<table>
<tr>
<td>Antes de ejecutar setPrice:</td>
<td>

	orderid	prod_id	price	quantity
1	1,043	4,003	[NULL]	1
2	1,050	466	[NULL]	1
3	1,050	2,480	[NULL]	1

</td>
</tr>
<tr>
<td>Después de la ejecución de setPrice y query de prueba:

```
SELECT
od.orderid,
o.orderdate,
p.price AS actual_price,
od.price AS order_price
FROM
   orders o
   JOIN orderdetail od ON
o.orderid = od.orderid
   JOIN products p ON od.prod_id =
p.prod_id
```

Esta query compara el precio actual que aparece en products con el precio que debe aparecer en orderdetail según la fecha en la que se ordenó el pedido.</td>
<td>

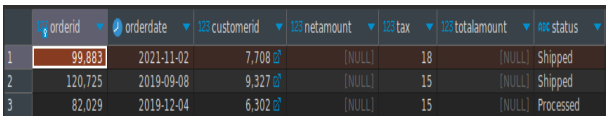
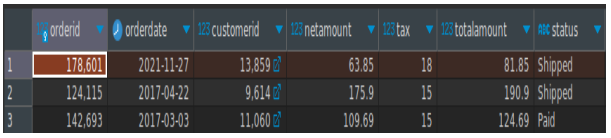
	orderid	orderdate	actual price	order price
1	1,187	2019-01-01	16.8	15.81

	orderid	orderdate	actual price	order price
1	174,528	2022-10-01	21.6	21.6

	orderid	prod_id	price	quantity
5737	1,043	971	10.85	1
5738	1,043	4,003	12.65	1

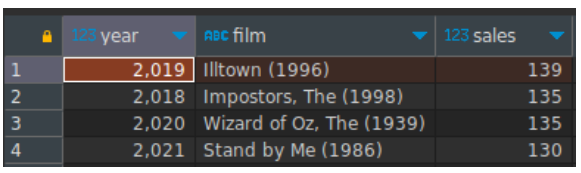
Se puede ver en el segundo resultado que no se modifica el precio en orderdetails de los pedidos realizados en 2022.
</td>
</tr>
</table>

setOrderAmount.sql: El procedimiento setOrderAmount actualiza las columnas **netamount** y **totalamount** en la tabla **orders**, inicialmente, calcula la cantidad neta de cada pedido sumando el producto de precio y cantidad con **SUM(price * quantity)** para cada pedido en **orderdetail**. Luego, actualiza **orders**, estableciendo **netamount** con esta cantidad neta calculada y **totalamount** como la suma de la cantidad neta más impuestos, pero solo si estas columnas están actualmente vacías (NULL). La actualización afecta únicamente a aquellos pedidos que tienen al menos una de las dos columnas (**netamount** o **totalamount**) vacías.

Antes de ejecutar setOrderAmount:	
Después de ejecutar setOrderAmount:	

getTopSales.sql: En la función getTopSales, se calculan inicialmente las ventas totales de cada película por año utilizando **SUM(od.quantity)**, lo que suma la cantidad total de unidades vendidas de cada película en lugar de contar simplemente las apariciones del **movieid**, y **date_part('year', o.orderdate)** para extraer el año de la fecha del pedido para agrupar las ventas por año. Luego para determinar las películas más vendidas por año, se aplica la función **RANK()** a estas sumas, asignando un rango a cada película dentro de cada año, donde el rango 1 indica la película más vendida. Después de asignar los rangos, la consulta filtra para seleccionar solo aquellas películas con **sales_rank = 1**, es decir, las más vendidas en cada año. La función devuelve estas películas más vendidas en orden descendente de ventas.

```
SELECT * FROM getTopSales(2018, 2021)
```



```
SELECT
date_part('year',o.orderdate)AS
year_sales,
im.movietitle,
sum(o2.quantity) AS total_quantity
FROM  orders o
      JOIN    orderdetail    o2    ON
o.orderid = o2.orderid
      JOIN products p ON o2.prod_id
= p.prod_id
      JOIN    imdb_movies    im    ON
im.movieid = p.movieid
WHERE
date_part('year',o.orderdate)=2019
GROUP   BY   p.movieid,   year_sales,
im.movietitle
ORDER BY total_quantity
DESC
LIMIT 1
```

Query para comprobar si se calcula correctamente el número de ventas en cada año.

	year_sales	movietitle	total_quantity
1	2,019	Illtown (1996)	139

	year_sales	movietitle	total_quantity
1	2,018	Impostors, The (19	135

	year_sales	movietitle	total_quantity
1	2,020	Wizard of Oz, The (	135

	year_sales	movietitle	total_quantity
1	2,021	Stand by Me (1986	130

getTopActors.sql: La función getTopActors primero crea una lista de todas las películas por actor en el género especificado, incluyendo el nombre del actor, el ID de la película, el año y el título, posteriormente, cuenta el número de películas en las que cada actor ha participado y determina la primera película en la que participó dentro del género. Luego, obtiene detalles adicionales sobre estas películas de debut, incluyendo el director, y finalmente la función retorna una lista ordenada de actores basada en la frecuencia de sus actuaciones en el género, incluyendo información sobre su película de debut y el director de esa película.

```
select * from
getTopActors('Drama')
```

	actorname	nummovies	debutyear	debutfilm	directorname
1	Duvall, Robert (I)	26	1,962	To Kill a Mockingbird (1962)	Mulligan, Robert
2	Jackson, Samuel L.	26	1,988	School Daze (1988)	Lee, Spike
2437	Lathan, Sanaa	5	1,999	Life (1999/I)	Demme, Ted
2438	Lathan, Sanaa	5	1,999	Wood, The (1999)	Famuyiwa, Rick

En el último resultado vemos que un actor puede tener dos películas en su año debut y que no se seleccionan las actores con menos de 5 apariciones en un género.

```
SELECT
	ia.actorname, ia2.actorid, C
COUNT(ia2.actorid)
FROM
	imdb_actors ia
	JOIN imdb_actormovies ia2 ON
ia.actorid = ia2.actorid
	JOIN imdb_movies im ON
ia2.movieid = im.movieid
	JOIN imdb_moviegenres im2 ON
im.movieid = im2.movieid
	JOIN genres g ON im2.genreid =
g.genreid
WHERE
	g.genrename = 'Drama' AND
ia.actorname = 'Jackson, Samuel L.'
GROUP BY
	ia2.actorid, ia.actorname
ORDER BY
	count(ia2.actorid)
DESC
```

Query para probar si se cuentan el número de películas de un actor en un genero correctamente

	actorname	actorid	count
1	Jackson, Samuel L.	305,494	26

	actorname	actorid	count
1	Lathan, Sanaa	937,496	5

3. Triggers

updOrders: Este trigger se dispara después de **insertar**, **actualizar** o **eliminar** un registro en la tabla **orderdetail**. Su función es actualizar los campos **netamount** y **totalamount** en la tabla **orders**. En el caso de que haya una **inserción** o **actualización** en **orderdetail**, calcula el **netamount** y **totalamount** (incluyendo impuestos) para el pedido correspondiente basándose en el **orderid**. Si hay una eliminación, se usa el **orderid** del registro eliminado.

updRatings: Este trigger se activa después de cualquier **inserción**, **actualización** o **eliminación** en la tabla **ratings**. Su objetivo es actualizar el promedio de calificaciones (**ratingmean**) y el conteo de calificaciones (**ratingcount**) para cada película en la tabla **imdb_movies**. Al insertar o actualizar una calificación, recalcula estos valores para la película correspondiente. Si una calificación es eliminada, actualiza estos valores considerando la eliminación.

updInventoryAndCustomer: Este trigger se ejecuta después de una **actualización** en la tabla **orders**. Su función es manejar los cambios en el estado de los pedidos, en concreto cuando un pedido cambia a estado **'Paid'**, en ese caso, actualiza la tabla **inventory** restando las cantidades vendidas del stock y aumentando las ventas, y ajusta el balance del cliente en la tabla customers restando el total del pedido.

4. Integración con Python

Para la integración con Python usamos SQLAlchemy que es una biblioteca para Python que nos facilita la interacción con bases de datos, en el código importamos funciones de SQLAlchemy para crear una conexión con la base de datos y escribir consultas SQL con **from sqlalchemy import create_engine from sqlalchemy.sql import text**. Definimos las variables **DATABASE_TYPE, USERNAME, PASSWORD, HOST, DATABASE_NAME, y PORT** que almacenan la información necesaria para conectarse a la base de datos PostgreSQL. A continuación, utilizamos **create_engine** para crear un motor de base de datos que proporciona acceso a la base de datos, y **engine.connect()** para establecer una conexión. Las variables **year1** y **year2** se utilizan en la función SQL **getTopSales** que se ejecuta a través de **conn.execute**, utilizando la función **text** para escribir la consulta y pasar parámetros.

Resultado de ejecutar mostrarTabla.py:

```
(base) carlos@carlos-IdeaPad-3-15ITL6:~/Downloads/P2_SI1_P01$ python3 mostrarTabla.py
(2021, 'Stand by Me (1986)', 130)
(2022, 'Jerk, The (1979)', 52)
```

Si lo ejecutamos en dbeaver con select * from getTopSales(2021, 2022) obtenemos el mismo resultado:

	123 year	ABC film	123 sales
1	2,021	Stand by Me (1986)	130
2	2,022	Jerk, The (1979)	52

Parte 2: Optimización

k) Estudio del impacto de un índice

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

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

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

```
EXPLAIN
SELECT COUNT(DISTINCT c.state)
FROM public.customers c
```

```
JOIN public.orders o ON c.customerid = o.customerid
WHERE c.country = 'Peru' AND EXTRACT(YEAR FROM o.orderdate) = 2017;
```

```
                                  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 = 'Peru'::text)
(10 filas)

 count
-------
   185
(1 fila)
```

Utilizando la instrucción EXPLAIN, se ha calculado un costo total aproximado de 4821.99 para la ejecución de la consulta. Este costo incluye tanto la localización como el procesamiento de una única fila. Se identificó que la estrategia de unión de las tablas es mediante un 'Hash Join'. Inicialmente, se realiza un escaneo secuencial en paralelo (Parallel Seq Scan) en la tabla "orders", lo que representa un costo considerable de 3291.03. Este proceso conlleva un escaneo completo de la tabla "orders", aplicando un filtro basado en las condiciones de la cláusula WHERE. A continuación, se efectúa un escaneo secuencial (Seq Scan) en la tabla "customers" que tiene un costo proyectado de 528.16, donde se filtran los registros que coinciden con el país 'Peru'. La combinación de estos conjuntos de datos se realiza mediante un "Hash Join" que tiene un costo estimado de 3821.47, juntando la información según la coincidencia en los identificadores de cliente (customerid).

c,d) Identificar un índice que mejore el rendimiento de la consulta y crearlo (si ya existía, borrarlo). Estudiar el nuevo plan de ejecución y compararlo con el anterior.

```
CREATE INDEX INDEX_ORDERDATE ON public.orders(EXTRACT(YEAR FROM
ORDERDATE));
```

```
                                   QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=2030.23..2030.24 rows=1 width=8)
   ->  Hash Join  (cost=548.50..2030.22 rows=5 width=118)
         Hash Cond: (o.customerid = c.customerid)
         ->  Bitmap Heap Scan on orders o  (cost=19.46..1498.79 rows=909 width=4)
               Recheck Cond: (EXTRACT(year FROM orderdate) = '2017'::numeric)
               ->  Bitmap Index Scan on index_orderdate  (cost=0.00..19.24 rows=909 width=0)
                     Index Cond: (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 filas)

 count
-------
   185
(1 fila)
```

Después de implementar el nuevo índice, lo hemos ejecutado en conjunto con la instrucción EXPLAIN para evaluar su impacto en la eficiencia y otros aspectos relevantes del proceso. Los resultados de las pruebas han confirmado una notable disminución en los costos de ejecución, pasando de 4821.99 a 2030.23. Esto representa una mejora considerable en términos de rendimiento y optimización del uso de recursos. Esta mejora se atribuye al hecho de que el índice fue creado específicamente para optimizar las columnas referenciadas en la cláusula WHERE de nuestra consulta.

e) **Probad distintos índices y discutid los resultados**

```
CREATE INDEX INDEX_COUNTRY ON public.customers(COUNTRY);
```

```
                                   QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=1681.90..1681.91 rows=1 width=8)
   ->  Hash Join  (cost=200.17..1681.89 rows=5 width=118)
         Hash Cond: (o.customerid = c.customerid)
         ->  Bitmap Heap Scan on orders o  (cost=19.46..1498.79 rows=909 width=4)
               Recheck Cond: (EXTRACT(year FROM orderdate) = '2017'::numeric)
               ->  Bitmap Index Scan on index_orderdate  (cost=0.00..19.24 rows=909 width=0)
                     Index Cond: (EXTRACT(year FROM orderdate) = '2017'::numeric)
         ->  Hash  (cost=179.83..179.83 rows=70 width=122)
               ->  Bitmap Heap Scan on customers c  (cost=4.83..179.83 rows=70 width=122)
                     Recheck Cond: ((country)::text = 'Peru'::text)
                     ->  Bitmap Index Scan on index_country  (cost=0.00..4.81 rows=70 width=0)
                           Index Cond: ((country)::text = 'Peru'::text)
(12 filas)

 count
-------
   185
(1 fila)
```

Hemos introducido un nuevo índice denominado “INDEX_COUNTRY” y lo hemos puesto a prueba en comparación con el índice previamente creado. Iniciamos el proceso eliminando el índice antiguo mediante un comando DROP. Posteriormente, al ejecutar nuevamente la consulta con el nuevo índice, el resultado mostró un costo de 1681.90 en la operación de agregación.

Al comparar este resultado con los obtenidos anteriormente, se deduce que el índice "INDEX_COUNTRY” logra una reducción más significativa en los costos de ejecución y otros factores relacionados. Por lo tanto, concluimos que el uso del índice "INDEX_COUNTRY” resulta en una optimización más efectiva del rendimiento y los recursos en comparación con el índice "INDEX_ORDERDATE”.

f) Modificar el script estadosDistintos.sql, añadiendo las sentencias de creación de índices y de planificación.

```
CREATE INDEX INDEX_ORDERDATE ON public.orders(EXTRACT(YEAR FROM
ORDERDATE));
CREATE INDEX INDEX_COUNTRY ON public.customers(COUNTRY);

--DROP INDEX INDEX_ORDERDATE;
--DROP INDEX INDEX_COUNTRY;

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

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

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

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

La primera consulta emplea 'NOT IN' para identificar aquellos customerid en la tabla customers que no coinciden con ningún customerid en los pedidos marcados como 'Paid'. Esta operación se lleva a cabo mediante una subconsulta en la tabla orders.

En la segunda consulta, se combina el uso de 'UNION ALL' para fusionar los customerid de customers y aquellos de orders con un estado 'Paid'. Posteriormente, aplica 'GROUP BY' y 'HAVING' para filtrar y mostrar solo aquellos customerid que aparecen una sola vez.

La tercera consulta, por su parte, recurre a 'EXCEPT' para obtener los customerid de customers excluyendo aquellos que se encuentran en los pedidos con estado 'Paid', identificados a través de una subconsulta en orders.

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

La consulta que inmediatamente empieza a entregar resultados es aquella que emplea la cláusula EXCEPT, ya que esta operación selecciona directamente aquellos registros de la tabla customers que no tienen correspondencia en la tabla orders. Este método es eficiente porque identifica y excluye los elementos de customers que no se encuentran en orders sin la necesidad de procesar completamente la tabla orders. En otras palabras, la consulta con EXCEPT es eficaz en generar resultados de forma más rápida al enfocarse específicamente en los registros no coincidentes entre las dos tablas mencionadas.

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

En lo que respecta a la ejecución en paralelo, la consulta que más probablemente se beneficie de esta técnica es aquella que incluye operaciones de agrupación GROUP BY y combina varias subconsultas mediante UNION ALL. Estas operaciones son particularmente aptas para ser ejecutadas de manera paralela, lo que puede acelerar significativamente el proceso de tratamiento de los datos. Esta ventaja depende, no obstante, de que tanto el sistema de gestión de bases de datos como su configuración actual permitan y sean compatibles con la ejecución paralela. En resumen, la capacidad de realizar tareas en paralelo puede optimizar notablemente el rendimiento de consultas complejas que involucran agrupaciones y uniones de múltiples conjuntos de datos.

m) Estudio del impacto de la generación de estadísticas:

b,c) Partir de la base de datos suministrada para este apartado (recién creada y cargada de datos). Estudiar con la sentencia EXPLAIN el coste de ejecución de las dos consultas indicadas en el Anexo 2.

El plan de ejecución revela que esta consulta realiza una operación de agregación para calcular el recuento de filas en la tabla "orders" donde el campo "STATUS" es NULL. Se utiliza una exploración secuencial de la tabla (Seq Scan), y se aplica un filtro para seleccionar las filas deseadas. El costo estimado de esta consulta es de 3507.17, pero el tiempo de ejecución real es sorprendentemente bajo, con un

promedio de 12.125 ms. Esto se debe a que el filtro selecciona una minoría de las filas en la tabla, lo que reduce significativamente la cantidad de datos a procesar.

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3507.17..3507.18 rows=1 width=8) (actual time=12.099..12.100 rows=1 loops=1)
   ->  Seq Scan on orders  (cost=0.00..3504.90 rows=909 width=0) (actual time=12.096..12.096 rows=0 loops=1)
         Filter: (status IS NULL)
         Rows Removed by Filter: 181790
 Planning Time: 0.119 ms
 Execution Time: 12.125 ms
(6 filas)
```

En esta segunda consulta, nuevamente se realiza una operación de agregación para calcular el recuento de filas en la tabla "orders", pero esta vez se filtra en base a la condición de que el campo "STATUS" sea igual a 'Shipped'. Al igual que en la primera consulta, se utiliza una exploración secuencial de la tabla (Seq Scan). El costo estimado de esta consulta es de 3961.65, y el tiempo de ejecución real es de 30.886 ms. La diferencia en el tiempo de ejecución en comparación con la primera consulta se debe al filtro más restrictivo que selecciona un subconjunto menor de filas, lo que resulta en un mayor costo estimado.

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3961.65..3961.66 rows=1 width=8) (actual time=30.858..30.859 rows=1 loops=1)
   ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=0) (actual time=0.011..24.706 rows=127323 loops=
         Filter: ((status)::text = 'Shipped'::text)
         Rows Removed by Filter: 54467
 Planning Time: 0.109 ms
 Execution Time: 30.886 ms
(6 filas)
```

Como conclusión de estas dos consultas podemos determinar que la primera consulta es más eficiente en términos de tiempo de ejecución debido a una condición de filtro menos restrictiva (NULL en lugar de una cadena específica), lo que resulta en un menor costo estimado y un tiempo de ejecución más rápido.

d,e) Crear un índice en la tabla orders por la columna status. Estudiar de nuevo la planificación de las mismas consultas.

```
CREATE INDEX INDEX_STATUS ON public.orders(STATUS)
```

Como hemos observado en el apartado anterior, la diferencia en el tiempo de ejecución en comparación con la primera consulta se debe al filtro más restrictivo en la segunda consulta, que selecciona un subconjunto menor de filas, lo que resulta en un costo estimado más alto. De cualquier forma, se observa cómo se han reducido significativamente los tiempos de ejecución después de crear el índice.

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3507.17..3507.18 rows=1 width=8) (actual time=12.198..12.198 rows=1 loops=1)
   ->  Seq Scan on orders  (cost=0.00..3504.90 rows=909 width=0) (actual time=12.193..12.194 rows=0 loops=1)
         Filter: (status IS NULL)
         Rows Removed by Filter: 181790
 Planning Time: 0.075 ms
 Execution Time: 12.227 ms
(6 filas)

                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3961.65..3961.66 rows=1 width=8) (actual time=20.105..20.106 rows=1 loops=1)
   ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=0) (actual time=0.004..16.404 rows=127323 loops=1)
         Filter: ((status)::text = 'Shipped'::text)
         Rows Removed by Filter: 54467
 Planning Time: 0.039 ms
 Execution Time: 20.121 ms
(6 filas)
```

El código utilizado, en conjunto, es el siguiente:

```sql
EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

CREATE INDEX INDEX_STATUS ON public.orders(STATUS);
--DROP INDEX INDEX_STATUS;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';
```

f,g,h) Ejecutar la sentencia ANALYZE para generar las estadísticas sobre la tabla orders. Estudiar de nuevo el coste de las consultas y comparar con el coste anterior a la generación de estadísticas. Comparar el plan de ejecución y

discutir el resultado. Comparar con la planificación de las otras dos consultas proporcionadas y comentar los resultados.

```
ANALYZE VERBOSE public.orders;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Paid';

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Processed';
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=4.32..4.33 rows=1 width=8) (actual time=0.006..0.007 rows=1 loops=1)
   ->  Index Only Scan using index_status on orders  (cost=0.29..4.31 rows=1 width=0) (actual time=0.004..0.004 rows=0 loops=1)
         Index Cond: (status IS NULL)
         Heap Fetches: 0
 Planning Time: 0.059 ms
 Execution Time: 0.019 ms
(6 filas)

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3011.06..3011.07 rows=1 width=8) (actual time=8.838..8.838 rows=1 loops=1)
   ->  Index Only Scan using index_status on orders  (cost=0.29..2690.21 rows=128338 width=0) (actual time=0.009..5.390 rows=127323 loops=1)
         Index Cond: (status = 'Shipped'::text)
         Heap Fetches: 0
 Planning Time: 0.029 ms
 Execution Time: 8.847 ms
(6 filas)

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=420.72..420.73 rows=1 width=8) (actual time=1.504..1.504 rows=1 loops=1)
   ->  Index Only Scan using index_status on orders  (cost=0.29..376.16 rows=17821 width=0) (actual time=0.017..0.944 rows=18163 loops=1)
         Index Cond: (status = 'Paid'::text)
         Heap Fetches: 0
 Planning Time: 0.026 ms
 Execution Time: 1.512 ms
(6 filas)

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=836.91..836.92 rows=1 width=8) (actual time=2.915..2.916 rows=1 loops=1)
   ->  Index Only Scan using index_status on orders  (cost=0.29..747.84 rows=35631 width=0) (actual time=0.010..1.829 rows=36304 loops=1)
         Index Cond: (status = 'Processed'::text)
         Heap Fetches: 0
 Planning Time: 0.025 ms
 Execution Time: 2.927 ms
(6 filas)
```

Luego de ejecutar el comando "ANALYZE" en PostgreSQL, se han observado reducciones significativas en los costos de ejecución tanto en la primera como en la segunda consulta. Esta mejora se debe a que:

El comando "ANALYZE" se utiliza para recopilar estadísticas precisas sobre las tablas y los índices en la base de datos. Estas estadísticas proporcionan al optimizador de consultas información valiosa sobre cómo están distribuidos los datos, el tamaño de las tablas y otros detalles relevantes.

Al disponer de información actualizada sobre la distribución de los datos y la cantidad de filas en las tablas, el optimizador puede tomar decisiones más acertadas sobre cómo acceder y unir los datos requeridos para una consulta específica. Esto puede llevar a la selección de planes de ejecución más eficientes, lo que se traduce en una disminución potencial del tiempo de ejecución y de los costos asociados con la consulta.

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

El contenido del archivo es el siguiente:

```sql
EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
```

```
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

CREATE INDEX INDEX_STATUS ON public.orders(STATUS);
--DROP INDEX INDEX_STATUS;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

ANALYZE VERBOSE public.orders;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Paid';

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Processed';
```

Cuestiones

¿Qué hace el generador de estadísticas?

El generador de estadísticas en PostgreSQL, activado por la sentencia ANALYZE, recopila información sobre las tablas de la base de datos y sus contenidos. Estas estadísticas incluyen datos como el número de filas, la distribución de los valores en las columnas, la frecuencia de los valores, etc. Esta información es crucial para el optimizador de consultas de PostgreSQL, ya que le permite tomar decisiones informadas sobre el mejor plan de ejecución para una consulta. Por ejemplo, puede decidir si usar un índice, qué tipo de join realizar, y en qué orden unir las tablas. Cuanto más precisas sean las estadísticas, más eficiente puede ser la planificación de las consultas.

¿Por qué la planificación de las dos consultas es la misma hasta que se generan las estadísticas?

Antes de generar estadísticas con ANALYZE, el planificador de consultas de PostgreSQL tiene que hacer suposiciones basadas en información limitada o desactualizada sobre los datos. Esto puede llevar a que el planificador elija el mismo plan de ejecución para diferentes consultas que podrían beneficiarse de planes diferentes. Una vez que se actualizan las estadísticas con ANALYZE, el planificador tiene información más precisa sobre los datos, lo que le permite tomar decisiones más informadas y potencialmente diferentes para cada consulta. Así, la planificación de las consultas puede diferir notablemente después de generar estadísticas actualizadas, lo que a menudo resulta en una mayor eficiencia y rendimiento.