

## GENERAL

La practica entregada contiene los tres ficheros SQL, la aplicación con los ficheros database.py y routes.py, los templates html de las paginas dadas, y el start.wsgi para acceder a todas las paginas desde el localhost.

Abajo estan las respuestas a las preguntas del enunciado con capturas de pantalla y descripciones de los metodos que hemos utilizado.

## OPTIMIZACIÓN

- A) Si hacemos EXPLAIN de la consulta, nos indica el coste de esta en diferentes partes, como lo que cuesta realizar el filtro, o juntar todo.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate	(cost=5627.93..5627.94 rows=1 width=8)	
2	-> Gather	(cost=1000.00..5627.92 rows=2 width=4)	
3	Workers Planned: 1		
4	-> Parallel Seq Scan on orders	(cost=0.00..4627.72 rows=1 width=4)	
5	Filter: ((totalamount > '100'::numeric) AND (date part('year'::text		

Si creamos dos índices, uno sobre la columna del año y el otro sobre la columna del mes (para los que se usara al hacer el filtro), podemos ver como los costes totales se reducen:

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate	(cost=58.05..58.06 rows=1 width=8)	
2	-> Bitmap Heap Scan on orders	(cost=38.73..58.05 rows=2 width=4)	
3	Recheck Cond: ((date part('month'::text, (orderdate)::timestamp without time		
4	Filter: (totalamount > '100'::numeric)		
5	-> BitmapAnd	(cost=38.73..38.73 rows=5 width=0)	
6	-> Bitmap Index Scan on mes	(cost=0.00..19.24 rows=909 width=0)	
7	Index Cond: (date part('month'::text, (orderdate)::timestamp with		
8	-> Bitmap Index Scan on anno	(cost=0.00..19.24 rows=909 width=0)	
9	Index Cond: (date part('year'::text, (orderdate)::timestamp witho		

Pero creamos un único índice sobre ambas columnas del filtro, y volvemos a hacer un EXPLAIN sobre la consulta, podemos ver como los costes de esta se reducen aún más:

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate	(cost=23.80..23.81 rows=1 width=8)	
2	-> Bitmap Heap Scan on orders	(cost=4.47..23.79 rows=2 width=4)	
3	Recheck Cond: ((date part('year'::text, (orderdate)::timestamp v		
4	Filter: (totalamount > '100'::numeric)		
5	-> Bitmap Index Scan on anno	(cost=0.00..4.47 rows=5 width=0)	
6	Index Cond: ((date part('year'::text, (orderdate)::timestamp		

Por tanto, para esta consulta podemos concluir que lo mejor para reducir costes seria hacer un único índice sobre ambas columnas de la búsqueda.

### Lista de clientes por mes

Mes y año: Abril 2015

#### Parámetros del listado:

Umbral mínimo:	300
Intervalo:	5
Número máximo de entradas:	1000

- ☐ Usar prepare  
☒ Parar si no hay clientes

B) Haciendo la consulta en apache, se ve que el rendimiento se cambia bastante. El tiempo de ejecución mejora con el uso de un índice, pero si el índice se tiene que crear de nuevo al ejecutar el programa, entonces el tiempo de rendimiento es peor.

### Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 69 ms

[Nueva consulta](#)

La imagen de arriba es el tiempo de ejecución sin un índice.

### Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 307 ms

[Nueva consulta](#)

La imagen de arriba es el tiempo de ejecución con la creación de un índice en el PREPARE.

## Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 17 ms

[Nueva consulta](#)

La imagen de arriba es el tiempo de ejecución con el índice ya creado.

Se ve que el mejor rendimiento es el que tiene un índice ya creado. El índice se hace sobre el año y mes. Otros índices se podrían hacer sobre el mes solo o el año solo, pero el mejor índice se hace juntando las dos cosas.

C)

- i) Nada más ejecutarse, la primera consulta devuelve los resultados. Esto es porque se hace un sequential scan directamente.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4)		
2	Filter: (NOT (hashed SubPlan 1))		
3	SubPlan 1		
4	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)		
5	Filter: ((status)::text = 'Paid'::text)		

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	HashAggregate (cost=4537.41..4539.41 rows=200 width=4)		
2	Group Key: customers.customerid		
3	Filter: (count(*) = 1)		
4	-> Append (cost=0.00..4462.40 rows=15002 width=4)		
5	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)		
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)		
7	Filter: ((status)::text = 'Paid'::text)		

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8)		
2	-> Append (cost=0.00..4603.32 rows=15002 width=8)		
3	-> Subquery Scan on "SELECT 1" (cost=0.00..634.86 rows=14093 width=8)		
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)		
5	-> Subquery Scan on "SELECT 2" (cost=0.00..3968.47 rows=909 width=8)		
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)		
7	Filter: ((status)::text = 'Paid'::text)		

ii) La segunda consulta, que usa consultas anidadas con uniones es la que más se beneficia de la ejecución en paralelo. La tercera también se beneficiaría, por el uso de subqueries, pero no tanto como la segunda.

D) El generador de estadísticas es un aspecto interno de la base de datos que calcula el mejor plan de ejecución a base de pruebas internas. La planificación de las dos consultas es la misma al principio, usando un sequential scan en ambos casos. Después de la creación del índice, la planificación de ambas consultas cambia a usar una búsqueda de índice. El coste de rendimiento usando esta planificación es mejor en ambas consultas. Al ejecutar el comando ANALYZE, se cambia de nuevo la planificación, mejorando el rendimiento en el caso de la primera consulta (respecto a las primeras ejecuciones), y empeorando el rendimiento de la segunda consulta (respecto a las primeras ejecuciones).

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate (cost=3507.17..3507.18 rows=1 width=8)		
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0)		
3	Filter: (status IS NULL)		

Primera consulta, sin indice.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate (cost=3961.65..3961.66 rows=1 width=8)		
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)		
3	Filter: ((status)::text = 'Shipped'::text)		

Segunda consulta, sin indice.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate (cost=1496.52..1496.53 rows=1 width=8)		
2	-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)		
3	Recheck Cond: (status IS NULL)		
4	-> Bitmap Index Scan on estado (cost=0.00..19.24 rows=909 width=0)		
5	Index Cond: (status IS NULL)		

Primera consulta con indice.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Aggregate (cost=1498.79..1498.80 rows=1 width=8)		
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)		
3	Recheck Cond: ((status)::text = 'Shipped'::text)		
4	-> Bitmap Index Scan on estado (cost=0.00..19.24 rows=909 width=0)		
5	Index Cond: ((status)::text = 'Shipped'::text)		

Segunda consulta con indice.

Output pane	Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text			
1	Aggregate (cost=7.28..7.29 rows=1 width=8)			
2	-> Index Only Scan using estado on orders (cost=0.42..7.28 rows=			
3	Index Cond: (status IS NULL)			

Primera consulta despues de ejecutar ANALYZE.

Data Output	Explain	Messages	History
	<b>QUERY PLAN</b> text		
1	Finalize Aggregate (cost=4210.57..4210.58 rows=1 width=8)		
2	-> Gather (cost=4210.45..4210.56 rows=1 width=8)		
3	Workers Planned: 1		
4	-> Partial Aggregate (cost=3210.45..3210.46 rows=1 width=8)		
5	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74705 width=8)		
6	Filter: ((status)::text = 'Shipped'::text)		

Segunda consulta despues de ejecutar ANALYZE.

## TRANSACCIONES Y DEADLOCKS

E) Ejemplo de la pagina principal de borraCliente.



### Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL  
☐ Transacción vía funciones SQLAlchemy

- ☐ Ejecutar commit intermedio  
☐ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

#### Trazas

El programa permite el borrado de un cliente por su ID. Da la opción de hacerlos usando transacciones de SQL o usando SQLAlchemy. También se puede elegir si se hace commits intermedios o si se quiere provocar un fallo para comprobar el rollback. Como se ve abajo, se hace primero el borrado de los registros asociados, y luego se borra el cliente.

### Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL  
☐ Transacción vía funciones SQLAlchemy

- ☐ Ejecutar commit intermedio  
☐ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

#### Trazas

1. Order con id 120 borrado de la tabla orderdetail
2. Order con id 117 borrado de la tabla orderdetail
3. Order con id 119 borrado de la tabla orderdetail
4. Order con id 116 borrado de la tabla orderdetail
5. Order con id 118 borrado de la tabla orderdetail
6. Orders del customer 3 borrado de la tabla orders
7. Customer con id 3 borrado de la tabla customers
8. La transacción se ha hecho con éxito.

Borrado con solo SQL.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☐ Transacción vía sentencias SQL  
☒ Transacción vía funciones SQLAlchemy  
☐ Ejecutar commit intermedio  
☐ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Order con id 120 borrado de la tabla orderdetail
2. Order con id 117 borrado de la tabla orderdetail
3. Order con id 119 borrado de la tabla orderdetail
4. Order con id 116 borrado de la tabla orderdetail
5. Order con id 118 borrado de la tabla orderdetail
6. Orders del customer 3 borrado de la tabla orders
7. Customer con id 3 borrado de la tabla customers
8. La transaccion se ha hecho con exito.

Borrado con solo SQLAlchemy.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL  
☐ Transacción vía funciones SQLAlchemy  
☒ Ejecutar commit intermedio  
☐ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Order con id 114 borrado de la tabla orderdetail
2. Order con id 115 borrado de la tabla orderdetail
3. Order con id 111 borrado de la tabla orderdetail
4. Order con id 113 borrado de la tabla orderdetail
5. Order con id 109 borrado de la tabla orderdetail
6. Order con id 112 borrado de la tabla orderdetail
7. Order con id 110 borrado de la tabla orderdetail
8. Usuario ha hecho commit.
9. Orders del customer 2 borrado de la tabla orders
10. Usuario ha hecho commit.
11. Customer con id 2 borrado de la tabla customers
12. La transaccion se ha hecho con exito.

Borrado con SQL y commits intermedios.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 

- ☐ Transacción vía sentencias SQL  
☒ Transacción vía funciones SQLAlchemy

☒ Ejecutar commit intermedio

☐ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Order con id 114 borrado de la tabla orderdetail
2. Order con id 115 borrado de la tabla orderdetail
3. Order con id 111 borrado de la tabla orderdetail
4. Order con id 113 borrado de la tabla orderdetail
5. Order con id 109 borrado de la tabla orderdetail
6. Order con id 112 borrado de la tabla orderdetail
7. Order con id 110 borrado de la tabla orderdetail
8. Usuario ha hecho commit.
9. Orders del customer 2 borrado de la tabla orders
10. Usuario ha hecho commit.
11. Customer con id 2 borrado de la tabla customers
12. La transaccion se ha hecho con exito.

Borrado con SQLAlchemy y commits intermedios.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 

- ☒ Transacción vía sentencias SQL  
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Ha saltado un error de transaccion. Rollback hecho.

Intento de borrado, provocando fallos.

F) Los datos alterados por la página o por el trigger no son visibles porque se hace el rollback antes de acabar.

El deadlock se produce porque cuando se hace un sleep, la base de datos se queda esperando una respuesta de los extremos del servidor, pero como pasa suficiente tiempo, se hace un timeout sin avisar y la pagina se queda esperando para siempre.

Para poder evitar esto, se puede crear una condicions de excepcion que avisa si una transaccion se ha hecho timeout, y entonces se manda un aviso y se deshace la transaccion.

## SEGURIDAD

G) Aqui se trabaja con SQL injections. Usando el nombre gatsby';--- se consigue hacer un select de solo el nombre de usuario, porque el '---' convierte el resto del commando en comentario.

**Ejemplo de SQL injection: Login**

Nombre:

Contraseña:

**Resultado**

Login correcto

1. First Name: italy  
Last Name: doze

En la siguiente imagen, con el uso de la contraseña 'or'1'='1, se consigue acceder a la primera cuenta en la base de datos.

**Ejemplo de SQL injection: Login**

Nombre:

Contraseña:

**Resultado**

Login correcto

1. First Name: pup  
Last Name: nosh

Si se prohíbe el uso de apostrofes, punticomas, y otros caracteres que se pueden asociar a SQL, se puede evitar el SQL injection, pero al coste de un poco de seguridad por tener contraseñas no tan seguros.

H) La consulta que consigue mostrar todas las tablas es la siguiente:

```
'; SELECT table_name  
FROM information_schema.tables  
WHERE table_type='BASE TABLE'  
AND table_schema='public';---
```

Porque se accede con conocimiento universal del funcionamiento de la base de datos postgres.

Para poder evitar eso, se podría usar un combobox, pero deja libre a ataques el protocolo GET. Entonces, cambiando el protocolo a POST y haciendo uso de un combo box se podría llegar a mejorar la protección contra ataques de SQL injection.