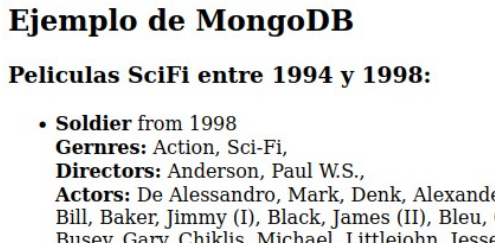
Memoria Sistemas Informáticos P3

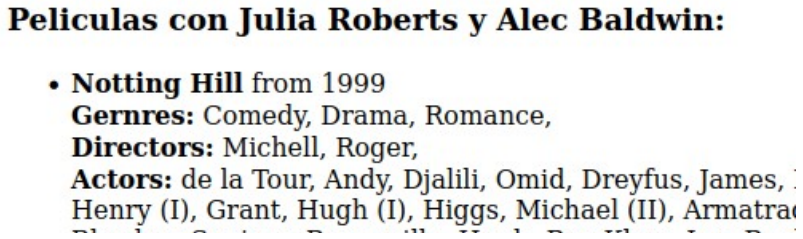
Para este proyecto, utilizaremos el esqueleto de aplicación proporcionado por la escuela, y utilizamos un makefile (explicado durante esta memoria) para agilizar las tareas.

Apartado 1: NoSQL

En este apartado creamos un ejecutable que obtenga las 400 películas británicas más actuales de las base de datos SQL proporcionada y las almacene en documentos de MongoDB. Este archivo (“createMongoDBFromPostgreSQLDB.py”, dentro de “app/”) consiste en, a través de consultas SQL a la BD original, obtener estas películas y su información, para despues guradarlas en una estructura que insertaremos en la base de datos de MongoDB “si1”, dentro de la colección “topUK”. También averiguamos cuales son las películas relacionadas, tal y como se exige en el guión de la práctica.

A través de la app proporcionada, con algunas modificaciones, podemos observar diferentes películas de esta BD que cumplen ciertos criterios:





Para este apartado hay preparado un comando en el makefile:

```
$ make apartado1
```

que consiste en eliminar la bd (si la hubiese), crearla y poblarla con “dump_v1.5-P3.sql.gz” y ejecutar el “.py” para crear la bd en MongoDB. Esto último también se puede hacer por separado con:

```
$ make to_mongo
```

Apartado 2: Optimización

E)

Para es te apartado utilizaremos el siguiente código que se exige en las cuestiones a), b), c):

```
drop index if exists indexTA
create index indexTA on customers(creditcardtype);

EXPLAIN analyze SELECT count (distinct customers.city)
FROM orders, customers
WHERE orders.customerid = customers.customerid
  AND extract(year FROM orderdate) = 2016
  AND extract(month FROM orderdate) = 04
  AND creditcardtype = 'VISA';


SELECT count (distinct customers.city)
FROM orders, customers
WHERE orders.customerid = customers.customerid
  AND extract(year from orderdate) = 2016
  AND extract(month from orderdate) = 04
  AND creditcardtype = 'VISA';
```

Empezaremos ejecutando el ultimo de los tres que es el correspondiente al apartado a) cuya resolución es:

```
count
-------
   282
(1 fila)
```

Es decir existen un total de 282 ciudades distintas

Procedemos ahora a ejecutar el segundo bloque o apartado b) del que obtenemos el siguente resultado

```
 Aggregate  (cost=5403.71..5403.72 rows=1 width=8) (actual time=34.925..34.926 rows=1 loops=1)
   ->  Gather  (cost=1000.28..5403.71 rows=1 width=118) (actual time=0.528..36.354 rows=304 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         ->  Nested Loop  (cost=0.29..4403.61 rows=1 width=118) (actual time=0.188..31.912 rows=152 loops=2)
               ->  Parallel Seq Scan on orders  (cost=0.00..4378.47 rows=3 width=4) (actual time=0.091..29.270 rows=594 loops=2)
                     Filter: ((date_part('year'::text, (orderdate)::timestamp without time zone) = '2016'::double precision) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))
                     Rows Removed by Filter: 90301
               ->  Index Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=122) (actual time=0.004..0.004 rows=0 loops=1188)
                     Index Cond: (customerid = orders.customerid)
                     Filter: ((creditcardtype)::text = 'VISA'::text)
                     Rows Removed by Filter: 1
 Planning time: 0.230 ms
 Execution time: 36.804 ms
(14 filas)
```

Podemos observar una versión extendida y analizada de la query anterior en la que además de ver los distintos filtros podemos observar los tipos de datos incluso el tiempo de ejecución.

Y por último ejecutamos los dos primeros bloques, es decir creamos un índice con lo que obtenemos el siguiente resultado:

```
 Aggregate  (cost=5403.71..5403.72 rows=1 width=8) (actual time=20.614..20.614 rows=1 loops=1)
   ->  Gather  (cost=1000.28..5403.71 rows=1 width=118) (actual time=0.592..22.289 rows=304 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         ->  Nested Loop  (cost=0.29..4403.61 rows=1 width=118) (actual time=0.176..18.319 rows=152 loops=2)
               ->  Parallel Seq Scan on orders  (cost=0.00..4378.47 rows=3 width=4) (actual time=0.106..16.396 rows=594 loops=2)
                     Filter: ((date_part('year'::text, (orderdate)::timestamp without time zone) = '2016'::double precision) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))
                     Rows Removed by Filter: 90301
               ->  Index Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=122) (actual time=0.003..0.003 rows=0 loops=1188)
                     Index Cond: (customerid = orders.customerid)
                     Filter: ((creditcardtype)::text = 'VISA'::text)
                     Rows Removed by Filter: 1
 Planning time: 0.489 ms
 Execution time: 22.686 ms
(14 filas)
```

Como podemos observar se reduce significativamente el tiempo de ejecución gracias al índice así como el planning time aumenta pero de forma prácticamente insignificante. Por lo que concluimos que el índice es claramente positivo para esta query.

F)

EJECUCIÓN 1

```
select customerid
from customerswhere customerid not in (select customerid
from orders
where status='Paid');
```

Ejecución:

```
 Seq Scan on customers  (cost=3978.08..4520.73 rows=7226 width=4) (actual
time=24.711..27.816 rows=4688 loops=1)
   Filter: (NOT (hashed SubPlan 1))
   Rows Removed by Filter: 9405
   SubPlan 1
     ->  Seq Scan on orders  (cost=0.00..3975.80 rows=913 width=4) (actual
time=0.014..19.547 rows=18163 loops=1)
           Filter: ((status)::text = 'Paid'::text)
           Rows Removed by Filter: 163627
 Planning time: 0.251 ms
 Execution time: 28.028 ms
```

En esta primera consulta, nos fijamos que para hacer el filtro NOT utiliza hashed SubPlan 1, que recorre recorre la tabla “customers” seleccionando y filtrando los “customersid” mediante la subconsulta entre paréntesis.

EJECUCIÓN 2

EXPLAIN analyze select customerid from (select customerid from customers union all select customerid from orders where status='Paid') as A group by customerid having count(*) =1;

Ejecución:

```
HashAggregate  (cost=4568.28..4570.28 rows=200 width=4) (actual time=34.078..36.465 rows=4688 loops=1)
  Group Key: customers.customerid
  Filter: (count(*) = 1)
  Rows Removed by Filter: 9405
  ->  Append  (cost=0.00..4491.45 rows=15365 width=4) (actual time=0.008..22.675 rows=32256 loops=1)
        ->  Seq Scan on customers  (cost=0.00..506.52 rows=14452 width=4) (actual time=0.008..1.851 rows=14093 loops=1)
        ->  Seq Scan on orders  (cost=0.00..3975.80 rows=913 width=4) (actual time=0.011..19.038 rows=18163 loops=1)
              Filter: ((status)::text = 'Paid'::text)
              Rows Removed by Filter: 163627
 Planning time: 0.219 ms
 Execution time: 36.862 ms
```

En esta consulta, se juntan las tablas “customers” y “orders” a través de “union all”, como resultado obtenemos una tabla de “customersid” con “orders.status” ya pagados. Estos “customersid” se agrupan para no quedarse con duplicados.
Se observa que el coste es mayor que en la anterior consulta, esto es ya que es recorren las dos tablas enteras. Así como el tiempo de ejecución. El ahorro en tiempo de planificación es tan pequeño que no es significante

EJECUCIÓN 3

```
select customerid
from customers
except
select customerid
from orders
where status='Paid';
```

Ejecución:

```
HashSetOp Except  (cost=0.00..4674.38 rows=14452 width=8) (actual
time=38.287..39.360 rows=4688 loops=1)
  ->  Append  (cost=0.00..4635.97 rows=15365 width=8) (actual time=0.013..29.229
rows=32256 loops=1)
      ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..651.04 rows=14452 width=8)
(actual time=0.012..3.110 rows=14093 loops=1)
            ->  Seq Scan on customers  (cost=0.00..506.52 rows=14452 width=4) (actual
time=0.011..1.960 rows=14093 loops=1)
      ->  Subquery Scan on "*SELECT* 2"  (cost=0.00..3984.93 rows=913 width=8)
(actual time=0.011..23.683 rows=18163 loops=1)
            ->  Seq Scan on orders  (cost=0.00..3975.80 rows=913 width=4) (actual
time=0.011..21.942 rows=18163 loops=1)
                  Filter: ((status)::text = 'Paid'::text)
                  Rows Removed by Filter: 163627
 Planning time: 0.221 ms
 Execution time: 39.953 ms
```

En esta última consulta, recorren y juntan las dos tablas, eliminando los duplicados con el “EXCEPT”. El tiempo de ejecución aumenta así como su tiempo de planificación. Esta consulta se beneficiaría de la paralelización de las tablas.

G)

Para este ejercicio iremos ejecutando y analizando distintas queries. Empezaremos ejecutándolas sobre la base de datos limpia sin índices ni estadísticas.

EXPLAIN analyze select count(*) from orders where status is null;

```
Aggregate  (cost=3521.72..3521.73 rows=1 width=8) (actual time=16.943..16.943 rows=1 loops=1)
  ->  Seq Scan on orders  (cost=0.00..3519.44 rows=913 width=0) (actual time=16.934..16.934 rows=0 loops=1)
        Filter: (status IS NULL)
        Rows Removed by Filter: 181790
 Planning time: 0.208 ms
 Execution time: 17.015 ms
```

EXPLAIN analyze select count(*) from orders where status='shipped';

```
Aggregate  (cost=3978.08..3978.09 rows=1 width=8) (actual time=53.478..53.479 rows=1 loops=1)
  ->  Seq Scan on orders  (cost=0.00..3975.80 rows=913 width=0) (actual time=0.007..40.760 rows=127323 loops=1)
        Filter: ((status)::text = 'Shipped'::text)
        Rows Removed by Filter: 54467
 Planning time: 0.064 ms
 Execution time: 53.520 ms
```

Ambas ejecuciones siguen una planificación similar la primera al no existir null en la columna tarda mucho menos.

Ahora ejecutaremos el índice:

create index indexSTTS on orders(status);

Los resultados han sido los siguentes:

EXPLAIN analyze select count(*) from orders where status is null;

```
Aggregate  (cost=1499.42..1499.43 rows=1 width=8) (actual time=0.049..0.049 rows=1 loops=1)
  ->  Bitmap Heap Scan on orders  (cost=19.46..1497.15 rows=909 width=0) (actual time=0.046..0.046 rows=0 loops=1)
        Recheck Cond: (status IS NULL)
        ->  Bitmap Index Scan on indexstts  (cost=0.00..19.24 rows=909 width=0) (actual time=0.044..0.044 rows=0 loops=1)
              Index Cond: (status IS NULL)
 Planning time: 0.332 ms
 Execution time: 0.087 ms
```

EXPLAIN analyze select count(*) from orders where status='shipped';

```
Aggregate  (cost=1501.70..1501.71 rows=1 width=8) (actual
time=34.473..34.474 rows=1 loops=1)
   ->  Bitmap Heap Scan on orders  (cost=19.46..1499.42 rows=909
width=0) (actual time=10.778..25.965 rows=127323 loops=1)
         Recheck Cond: ((status)::text = 'Shipped'::text)
         Heap Blocks: exact=1686
         ->  Bitmap Index Scan on indexstts  (cost=0.00..19.24
rows=909 width=0) (actual time=10.562..10.562 rows=127323 loops=1)
               Index Cond: ((status)::text = 'Shipped'::text)
 Planning time: 0.082 ms
 Execution time: 34.518 ms
```

Como podemos comprobar gracias al índice los tiempos de ejecución han bajado significativamente sobre todo en la primera como era de esperar tras la creación de un índice. Sin embargo en la segunda vemos que la mejora es menor en parte por el hecho de que tiene que recorrer dos veces el índice.

Ahora ejecutamos ANALYZE para obtener estadísticas y hacemos lo mismo

EXPLAIN analyze select count(*) from orders where status is null;

```
Aggregate  (cost=7.26..7.27 rows=1 width=8) (actual time=0.011..0.011 rows=1 loops=1)
  ->  Index Only Scan using indexstts on orders  (cost=0.42..7.26 rows=1 width=0) (actual
time=0.008..0.008 rows=0 loops=1)
       Index Cond: (status IS NULL)
       Heap Fetches: 0
 Planning time: 0.335 ms
 Execution time: 0.039 ms
```

EXPLAIN analyze select count(*) from orders where status='Shipped';

```
Finalize Aggregate  (cost=4218.59..4218.60 rows=1 width=8) (actual time=23.356..23.357
rows=1 loops=1)
  ->  Gather  (cost=4218.48..4218.59 rows=1 width=8) (actual time=23.235..25.540
rows=2 loops=1)
       Workers Planned: 1
       Workers Launched: 1
       ->  Partial Aggregate  (cost=3218.48..3218.49 rows=1 width=8) (actual
time=21.117..21.117 rows=1 loops=2)
             ->  Parallel Seq Scan on orders  (cost=0.00..3030.69 rows=75115 width=0)
(actual time=0.011..16.293 rows=63662 loops=2)
                   Filter: ((status)::text = 'Shipped'::text)
                   Rows Removed by Filter: 27234
 Planning time: 0.076 ms
 Execution time: 25.575 ms
```

La sentencia ANALYZE genera estadísticas sobre la tabla orders que permite mejorar la planificación de estas y así reducir sustancialmente sus tiempos de ejecución. Por ejemplo en la primera se utiliza únicamente el índice creado. Mientras en la segunda no utiliza el índice sino las estadísticas obtenidas lo que tiene sentido ya que hemos repetido esta consulta hasta en 3 ocasiones de ahí la mejora en tiempo. Aún así tiene sentido que siga siendo más costosa que la primera

EJECUTAMOS LAS DOS QUERIES RESTANTES:

EXPLAIN analyze select count(*) from orders where status='Paid';

```
Aggregate  (cost=2319.31..2319.32 rows=1 width=8) (actual time=7.676..7.676 rows=1 loops=1)
  ->  Bitmap Heap Scan on orders  (cost=355.71..2274.37 rows=17973 width=0) (actual time=1.520..6.545 rows=18163 loops=1)
        Recheck Cond: ((status)::text = 'Paid'::text)
        Heap Blocks: exact=1686
        ->  Bitmap Index Scan on indexstts  (cost=0.00..351.22 rows=17973 width=0) (actual time=1.345..1.345 rows=18163 loops=1)
              Index Cond: ((status)::text = 'Paid'::text)
 Planning time: 0.104 ms
 Execution time: 7.725 ms
```

EXPLAIN analyze select count(*) from orders where status='Processed;

```
Aggregate  (cost=2948.20..2948.21 rows=1 width=8) (actual time=15.500..15.500 rows=1 loops=1)
  ->  Bitmap Heap Scan on orders  (cost=712.37..2857.89 rows=36122 width=0) (actual time=5.604..12.753 rows=36304 loops=1)
        Recheck Cond: ((status)::text = 'Processed'::text)
        Heap Blocks: exact=1685
        ->  Bitmap Index Scan on indexstts  (cost=0.00..703.33 rows=36122 width=0) (actual time=5.402..5.402 rows=36304 loops=1)
              Index Cond: ((status)::text = 'Processed'::text)
 Planning time: 0.095 ms
 Execution time: 15.544 ms
```

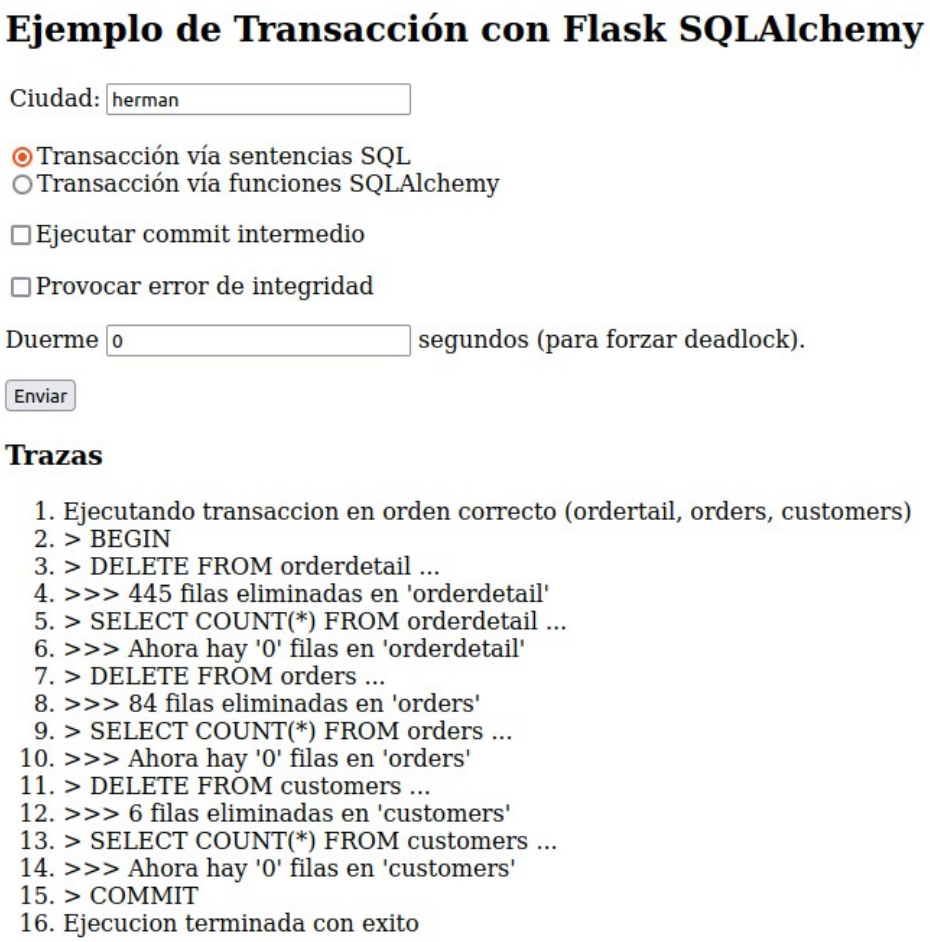
Ninguna de las dos utiliza las estadísticas ya que es la primera vez que se ejecutan por lo que utilizan el índice ambas tienen un coste mucho menor que ‘Shipped’ por lo que tardan más en ejecutarse de hecho ‘Processed’ tarde más que ‘Paid’ pues esta afecta a menos filas.

Apartado 3: Transacciones

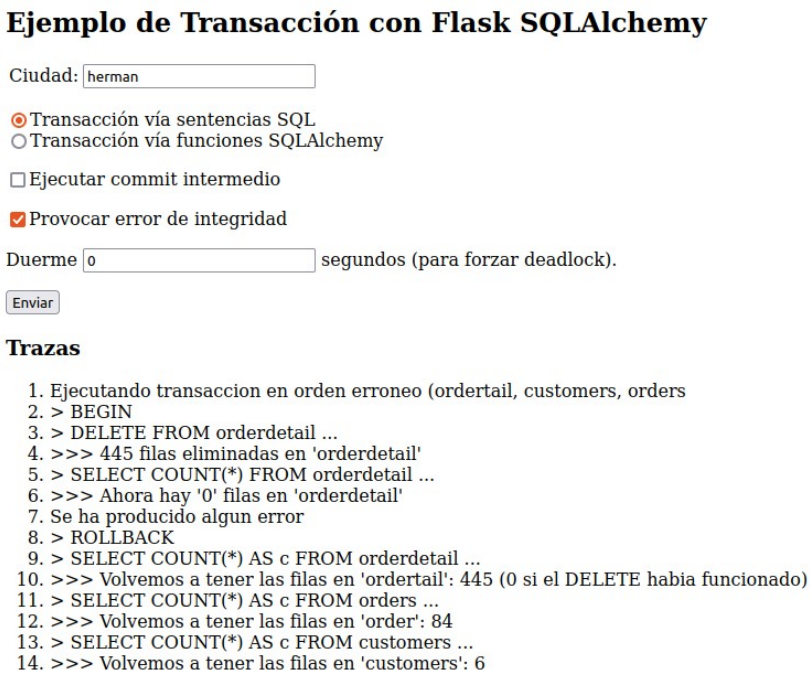
H) Estudio de transacciones:

Para este apartado, implementaremos la funcionalidad de borrar todos los clientes, junto con sus compras, que pertenezcan a una ciudad dada. Para esto, es imprescindible realizar los “DELETE” de las tablas en orden correcto: “orderdetail”, después en “order”, y para terminar en “customers”.

Si en la página correspondiente de la app (“/borrarCiudad”) marcamos la opción correspondiente a una ejecución correcta, este será el orden usado, y esta la salida:



Por el contrario, si se desea forzar una ejecución erronea, deberemos marcar la opción adecuada:



Para poder realizar esta prueba de ejecución manual erronea, es necesario desactivar el eliminado en cascada que realiza la bd, para esto creamos el archivo “no_delete_on_cascade.sql”.

El comando para preparar este apartado es:

$ make apartado3_H

que vuelve a crear y poblar la bd, y después ejecuta “no_delete_on_cascade.sql”

H) Estudio de transacciones:

En este apartado necesitamos crear un archivo “updPromo.sql” que realiza ciertas modificaciones en la base de datos.

La primera de estas es añadir una columna nueva “promo” a la tabla “customers” para guardar un descuento. Después crearemos un trigger que, ante una modificación es esta columna, aplique el descuento los pedidos del usuario. Se añadiran varios “sleep” en determinados momentos para poder observar el funcionamiento interno.

El comando para preparar este apartado es:

```
$ make apartado3_I
```

que vuelve a crear y poblar la bd y después ejecuta el “updPromo.sql”.