Práctica 6: Índices y optimización de las BD

Autores

Airam Rafael Luque León (alu0101335148@ull.edu.es)

Andrés Pérez Castellano (alu0101313511@ull.edu.es)

Tareas

1. Restauración de la base de datos postgres_air.

Primero que nada, se debe crear la base de datos, sobre la que restauraremos la información

```
CREATE DATABASE hettie;
```

Una vez creada, se restaura la información de la base de datos

```
pg_restore -x --no-owner -U postgres -d hettie /home/usuario/postgres_air.backup;
```

2. Incluir en la base de datos las siguientes sentencias SQL. ¿Qué acciones realizan?

```
SET search_path TO postgres_air;
CREATE INDEX flight_departure_airport ON
flight(departure_airport);
CREATE INDEX flight_scheduled_departure ON postgres_air.flight
(scheduled_departure);
CREATE INDEX flight_update_ts ON postgres_air.flight (update_ts);
CREATE INDEX booking_leg_booking_id ON postgres_air.booking_leg
(booking_id);
CREATE INDEX booking_leg_update_ts ON postgres_air.booking_leg
(update_ts);
CREATE INDEX account_last_name
ON account (last_name);
```

Primeramente estamos ajustando el espacio de nombres de la base de datos, para que las siguientes sentencias que se ejecuten, encuentren los objetos (tablas, vistas, etc) que se han definido previamente. Con las siguientes sentencias, estamos creando índices, para algunos atributos de las tablas de la base de datos.

Los índices son estructuras de datos que permiten acelerar las búsquedas en las tablas, a costa de un mayor consumo en memoria. Por defecto, se utiliza el algoritmo **btree**.

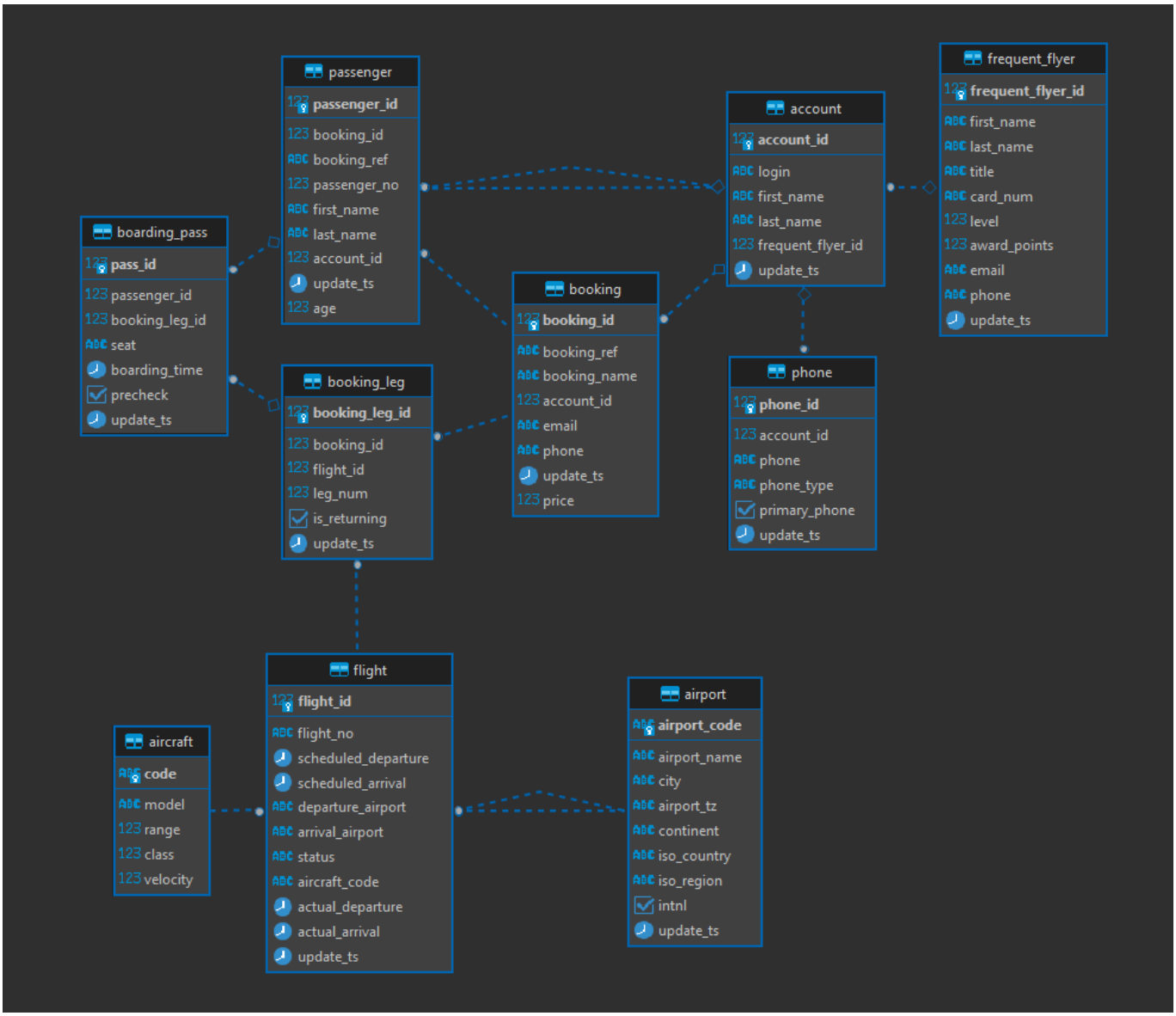
3. Identificar las tablas principales y sus principales elementos.

Las tablas principales son las siguientes:

- **account**: contiene información sobre los clientes de la aerolínea.
- **airport**: contiene información sobre los aeropuertos.
- **flight**: contiene información sobre los vuelos.
- **boarding_pass**: contiene información sobre los billetes de los vuelos.
- **passenger**: contiene información sobre los pasajeros.
- **booking_leg**: contiene información sobre las reservas de los vuelos.

4. Obtener un diagrama Entidad-Relación.

Para esta tarea utilizaremos DBeaver, que nos permite generar el diagrama de forma automática.



5. Realizar la siguiente consulta:

```sql
SELECT flight_id, scheduled_departure
FROM flight f
JOIN airport a
ON departure_airport=airport_code
AND iso_country='US'
```

a) Utilice el comando EXPLAIN para obtener el plan de consulta.

Al usar este comando, se obtiene la siguiente información:

```
QUERY PLAN
---------------------------------------------------------------------------------------------------------------
Hash Join  (cost=20.09..17229.60 rows=144636 width=12) (actual time=0.485..378.781 rows=338858 loops=1)
  Hash Cond: (f.departure_airport = a.airport_code)
  ->  Seq Scan on flight f  (cost=0.00..15404.76 rows=683176 width=16) (actual time=0.045..197.186 rows=683176 loops=1)
  ->  Hash  (cost=18.33..18.33 rows=141 width=4) (actual time=0.401..0.402 rows=141 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        ->  Seq Scan on airport a  (cost=0.00..18.33 rows=141 width=4) (actual time=0.040..0.307 rows=141 loops=1)
              Filter: (iso_country = 'US'::text)
              Rows Removed by Filter: 525
Planning Time: 0.663 ms
Execution Time: 395.120 ms
```

b) Lea el resultado del plan y determine el costo total de la consulta, el costo de configuración, la cantidad de filas que se devolverán y cantidad de filas estimadas.

Costo total de la consulta: 17229.60

Costo de configuración: 20.09

Cantidad de filas que se devolverán: 338858

Cantidad de filas estimadas: 144636

c) Repita la consulta del paso (b) de esta actividad, esta vez limitando el número de registros devueltos a 15.

```
QUERY PLAN
---------------------------------------------------------------------------------------------------------------
Limit  (cost=0.70..6.43 rows=15 width=12) (actual time=4.226..4.263 rows=15 loops=1)
  ->  Merge Join  (cost=0.70..55254.44 rows=144636 width=12) (actual time=4.224..4.259 rows=15 loops=1)
        Merge Cond: (f.departure_airport = a.airport_code)
        ->  Index Scan using flight_departure_airport on flight f  (cost=0.42..52031.86 rows=683176 width=16) (actual time=0.028..3.607 rows=2433 loops=1)
        ->  Index Scan using airport_pkey on airport a  (cost=0.28..67.93 rows=141 width=4) (actual time=0.055..0.055 rows=1 loops=1)
              Filter: (iso_country = 'US'::text)
              Rows Removed by Filter: 4
Planning Time: 1.060 ms
Execution Time: 4.364 ms
```

d) Revise el plan de consulta actualizado y compare su resultado con el resultado del paso anterior, prestando especial atención a cuántos pasos están involucrados en el plan de consulta y cuál es el costo del paso limitante.

Costo total de la consulta: 6.43

Costo de configuración: 0.70

Cantidad de filas que se devolverán: 15

Cantidad de filas estimadas: 15

Dado que se ha limitado explícitamente el número de filas devueltas, el sistema conoce con certeza el coste real de la operación y por tanto su estimación es perfecta, en este caso.

También vemos como la estrategia de búsqueda cambia entre ambos casos:

- Sin límite: se usa `Seq scan`.
- Con límite: se usa `Index Scan`.

La [documentación](#) de `postgres` tiene un ejemplo similar con `limit`.

6. Realice la siguiente consulta similar a la del paso anterior.

```sql
SELECT flight_id, scheduled_departure
FROM flight f
JOIN airport a
ON departure_airport=airport_code
AND iso_country='CZ'
```

a) Utilice el comando EXPLAIN para obtener el plan de consulta.

```
QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=20.40..2965.52 rows=1026 width=12) (actual time=25.398..51.237 rows=1066 loops=1)
  ->  Seq Scan on airport a  (cost=0.00..18.33 rows=1 width=4) (actual time=0.241..0.361 rows=1 loops=1)
        Filter: (iso_country = 'CZ'::text)
        Rows Removed by Filter: 665
  ->  Bitmap Heap Scan on flight f  (cost=20.40..2936.91 rows=1029 width=16) (actual time=25.140..50.208 rows=1066 loops=1)
        Recheck Cond: (departure_airport = a.airport_code)
        Heap Blocks: exact=802
        ->  Bitmap Index Scan on flight_departure_airport  (cost=0.00..20.14 rows=1029 width=0) (actual time=25.005..25.005 rows=1066 loops=1)
              Index Cond: (departure_airport = a.airport_code)
Planning Time: 1.242 ms
Execution Time: 51.592 ms
```

b) Compare los resultados con los obtenidos en la anterior interrogante.

Las consultas que estamos comparando son bastante parecidas, pero los planes difieren un poco, en el plan. En parte, esto puede estar relacionado con el hecho de que en Estados Unidos hay más aeropuertos que en la República Checa.

```
hettie=# SELECT COUNT(*) FROM airport WHERE iso_country = 'US';
 count
-------
   141
(1 row)

hettie=# SELECT COUNT(*) FROM airport WHERE iso_country = 'CZ';
 count
-------
     1
(1 row)
```

c) Explique la diferencia de rendimiento.

Como se ha comentado en el apartado anterior debido a la diferencia de aeropuertos entre los dos países, el planificador cambia la búsqueda cambia al tipo `BitMap`, frente a la estrategia secuencial ya que la primera es más eficiente. Pero sobre todo la diferencia se encuentra en la diferencia de filas que retorna cada consulta.

7. Compare estas dos consultas e indique cual es la de mayor rendimiento y la causa. Construya una vista con la que obtengan un mejor rendimiento.

Consulta A:

```
SELECT flight_id
,departure_airport
,arrival_airport
FROM flight
WHERE scheduled_arrival BETWEEN
'2020-10-14' AND '2020-10-15';
```

Consulta B:

```
SELECT flight_id
,departure_airport
,arrival_airport
FROM flight
WHERE scheduled_arrival:: date='2020-10-14';
```

a) Construya índices para con los algoritmos que sean válidos para A y B.

Se construyen varios índices para el atributo `scheduled_arrival`, puesto que está implicado en el filtro de la cláusula `WHERE`.

Se utilizan algoritmos que permitan trabajar con operadores de comparación booleanos:

- `btree`
- `hash`
- `brin`

```
CREATE INDEX scheduled_arrival_idx ON flight USING btree (scheduled_arrival);
-- Benchmarks
DROP INDEX scheduled_arrival_idx;

CREATE INDEX scheduled_arrival_idx ON flight USING hash (scheduled_arrival);
-- Benchmarks
DROP INDEX scheduled_arrival_idx;

CREATE INDEX scheduled_arrival_idx ON flight USING brin (scheduled_arrival);
-- Benchmarks
DROP INDEX scheduled_arrival_idx;
```

b) Evalué el rendimiento (tiempo) que se obtuvieron con lo nuevos índices al realizar las consultas. Construya una tabla con los resultados.

Tabla de resultados:

Consulta	Algoritmo	Coste total de la consulta	Nº Filas totales	Nº Filas estimada	T. planificación (ms)	T. ejecución (ms)
A	ninguno	14234.35	4516	3975	0.659	68.129
B	ninguno	14178.45	4502	3416	0.141	90.957
A	btree	7111.17	4516	3975	0.732	27.490
B	btree	14178.45	4502	3416	0.135	97.208
A	hash	14234.35	4516	3975	1.360	69.886
B	hash	14178.45	4502	3416	0.175	79.025
A	brin	8867.32	4516	3975	0.240	116.627
B	brin	14178.45	4502	3416	0.184	76.040

> Nota: cabe destacar que la consulta B en su versión sin `::date`, obtiene el mejor rendimiento con 0.232 ms (hash) y 0.534 ms (btree).

Cuando no utilizamos índices, el planificador opta por una estrategia `Parallel Seq Scan` (2 procesos). Dado que se trata de una búsqueda secuencial, el sistema debe realizar todas las comprobaciones posibles. Según la intuición, un filtro más restrictivo como el de B, será propenso a generar más filas que A, aunque por las pruebas realizadas, parece que la operación *type casting* al tipo *date* está penalizando a la consulta B.

El algoritmo `btree` acelera considerablemente la ejecución de A. Es una estructura de datos muy eficiente para la búsqueda de valores dentro de un rango, como es el caso del operador `BETWEEN`. Curiosamente, el

planificador decide ignorar el índice para la consulta B.

Por las propias características del algoritmo `hash`, solamente pueden sacarle partido aquellos filtros de comparación de igualdad, para un único valor del atributo. Este no es caso de la consulta A y por tanto el planificador elige otro tipo de plan.

El índice `brin` no parece estar optimizado para el tipo de dato que utiliza el atributo `scheduled_arrival`.

En general, la consulta A ofrece mejor desempeño (btree), creamos una vista para ella:

```
CREATE VIEW vuelos_dias_14_15 AS
SELECT flight_id, departure_airport, arrival_airport
FROM flight
WHERE scheduled_arrival BETWEEN '2020-10-14' AND '2020-10-15';
```