

UD5 - EXAMEN DE OPTIMIZACIÓN DE BASES DE DATOS (MySQL)

Cristóbal Suárez Abad

1. Contexto Real

Un marketplace de tecnología en plena expansión, está sufriendo graves problemas de rendimiento en su plataforma. La base de datos actual fue creada de forma rápida por un equipo que no aplicó criterios de optimización. Actualmente, con cerca de 1 millón de registros, las búsquedas de clientes tardan segundos, el servidor de disco está saturado y el proceso de generación de reportes mensuales bloquea el sistema.

Tu objetivo como experto en bases de datos es auditar, proponer cambios estructurales y demostrar con métricas la mejora de rendimiento tras la optimización.

2. Estructura Ineficiente (Script Inicial)

Ejecuta el script para crear el escenario base.

```
D:\2*_ASIR\ASGBO\Tema 05\Examen>scp "UD5 - Examen (Script).sql" cristobal@10.2.7.101:/home/cristobal/
cristobal@10.2.7.101's password:
UD5 - Examen (Script).sql
D:\2*_ASIR\ASGBO\Tema 05\Examen>
```

`mysql -u root -p < UD5\ -\ Examen\ \ (Script\).sql`

```
root@ubuntu:mysqlsuarez:/home/cristobal# mysql -u root -p < UD5\ -\ Examen\ \ (Script\).sql
Enter password:
```

Se lleva un rato

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| 2ASIR    |
| employees|
| global_express_db |
| information_schema |
| logistica_db |
| mysql    |
| performance_schema |
| sys      |
| tienda   |
+-----+
9 rows in set (0,02 sec)

mysql> |
```

3. Consultas más frecuentes (Workload)

Estas son las consultas que el departamento de ventas y atención al cliente ejecuta constantemente y que actualmente van muy lentas:

1. **Búsqueda de pedidos por cliente:**

```
SELECT * FROM pedidos_brutos WHERE cliente_email =  
'juan.perez@email.com';
```

2. **Filtrado por categoría y rango de precios:**

```
SELECT producto_nombre, precio_unitario FROM pedidos_brutos  
WHERE categoria_nombre = 'Laptops' AND precio_unitario > 1000  
ORDER BY precio_unitario DESC;
```

3. **Búsqueda de palabras clave en la descripción:**

```
SELECT producto_nombre FROM pedidos_brutos WHERE  
comentario_producto LIKE '%potente%';
```

4. **Reporte de ventas totales por país:**

```
SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida)  
FROM pedidos_brutos GROUP BY codigo_pais;
```

4. Tareas del Alumno

Fase 1: Análisis Inicial (2 puntos)

- Ejecuta las consultas anteriores usando **EXPLAIN** y **EXPLAIN ANALYZE**.
- Documenta los puntos críticos.

1. Búsqueda de pedidos por cliente:

EXPLAIN FORMAT=TREE

SELECT * FROM pedidos_brutos WHERE cliente_email = 'user2774@express.com';

```
mysql> EXPLAIN FORMAT=TREE
  -> SELECT * FROM pedidos_brutos WHERE cliente_email = 'user2774@express.com';
+-----+
| EXPLAIN
+-----+
| -> Filter: (pedidos_brutos.cliente_email = 'user2774@express.com') (cost=4980 rows=4844)
  -> Table scan on pedidos_brutos (cost=4980 rows=48435)
  |
+-----+
1 row in set (0.00 sec)
```

Table Scan: Realiza una lectura de todas las filas de la tabla, en este caso ESTIMA unas 48435, para luego filtrar las que tienen el email del usuario (unas 4844)

EXPLAIN ANALYZE FORMAT=TREE

SELECT * FROM pedidos_brutos WHERE cliente_email = 'user2774@express.com';

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
      -> SELECT * FROM pedidos_brutos WHERE cliente_email = 'user2774@express.com';
+-----+
| EXPLAIN                                     |
+-----+
| -> Filter: (pedidos_brutos.cliente_email = 'user2774@express.com') (cost=4980 rows=4844) (actual time=17.2..130 rows=11 loops=1)
      -> Table scan on pedidos_brutos (cost=4980 rows=48435) (actual time=0.0835..111 rows=50000 loops=1)
      |
+-----+
1 row in set (0.14 sec)
```

Table Scane: La estimación y la realidad varían poco, al final tiene que leer las 50.000 filas de la tabla (estimaba 48.435). Y tarda 130 milisegundos. Loop= 1 → Solo se repite una vez el proceso.

Muestra 11 filas.

2. Filtrado por categoría y rango de precios:

EXPLAIN FORMAT=TREE

SELECT producto_nombre, precio_unitario FROM pedidos_brutos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;

```
mysql> EXPLAIN FORMAT=TREE
-> SELECT producto_nombre, precio_unitario FROM pedidos_brutos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;
+-----+
| EXPLAIN                                     |
+-----+
| -> Sort: pedidos_brutos.precio_unitario DESC (cost=4980 rows=48435)
|   -> Filter: ((pedidos_brutos.categoria_nombre = 'Hogar') and (pedidos_brutos.precio_unitario > 500)) (cost=4980 rows=48435)
|     -> Table scan on pedidos_brutos (cost=4980 rows=48435)
|
```

Table Scan: Lee otra vez la tabla entera (estima 48.435 filas) y después filtrar por categoría_nombre y precio_unitario.

EXPLAIN ANALYZE FORMAT=TREE

SELECT producto_nombre, precio_unitario FROM pedidos_brutos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
-> SELECT producto_nombre, precio_unitario FROM pedidos_brutos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;
+-----+
| EXPLAIN
+-----+
| -> Sort: pedidos_brutos.precio_unitario DESC (cost=4980 rows=48435) (actual time=106..106 rows=269 loops=1)
   -> Filter: ((pedidos_brutos.categoria_nombre = 'Hogar') and (pedidos_brutos.precio_unitario > 500)) (cost=4980 rows=48435) (actual time=0.326..106 rows=269 loops=1)
     -> Table scan on pedidos_brutos (cost=4980 rows=48435) (actual time=0.0932..88.2 rows=50000 loops=1)
|
```

Table Scan, lee la tabla entera 50.000 (de 0.0932 milisegundos a 88.2 milisegundos) y poca diferencia con la estimación. Después filtra por categoria_nombre y precio_unitario (desde 0.326 ms a 106 ms). Tarda en total 106 ms. Solo un loop.

3. Búsqueda de palabras clave en la descripción:

EXPLAIN FORMAT=TREE

SELECT producto_nombre FROM pedidos_brutos WHERE comentario_producto LIKE '%calidad%';

```
mysql> EXPLAIN FORMAT=TREE
      -> SELECT producto_nombre FROM pedidos_brutos WHERE comentario_producto LIKE '%calidad%';
+-----+
| EXPLAIN
+-----+
| -> Filter: (pedidos_brutos.comentario_producto like '%calidad%') (cost=4980 rows=5381)
      -> Table scan on pedidos_brutos (cost=4980 rows=48435)
      |
+-----+
1 row in set (0.00 sec)
```

Table scan: otra vez tiene que leer toda la tabla.

EXPLAIN ANALYZE FORMAT=TREE

SELECT producto_nombre FROM pedidos_brutos WHERE comentario_producto LIKE '%calidad%';

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
      -> SELECT producto_nombre FROM pedidos_brutos WHERE comentario_producto LIKE '%calidad%';
+-----+
| EXPLAIN                                     |
+-----+
| -> Filter: (pedidos_brutos.comentario_producto like '%calidad%') (cost=4980 rows=5381) (actual time=0.127..157 rows=25026 loops=1)
|   -> Table scan on pedidos_brutos (cost=4980 rows=48435) (actual time=0.0773..83.7 rows=50000 loops=1)
|
+-----+
```

Poca diferencia en estimado y realidad a la hora de hacer el table scan (0.0773 ms a 83.7 ms), un loop.

Después filtra por comentario_producto. Para “calidad” filtra 25.026 filas. Solo un loop (una vez). Tarda en total 157 ms.

4. Reporte de ventas totales por país:

EXPLAIN FORMAT=TREE

SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos_brutos GROUP BY codigo_pais;

```
mysql> EXPLAIN FORMAT=TREE
      -> SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos_brutos GROUP BY codigo_pais;
+-----+
| EXPLAIN
+-----+
| -> Table scan on <temporary>
    -> Aggregate using temporary table
        -> Table scan on pedidos_brutos (cost=4980 rows=48435)
    |
+-----+
1 row in set (0,00 sec)
```

Vuelve a hacer un table scan (leer toda la tabla).

EXPLAIN ANALYZE FORMAT=TREE

SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos_brutos GROUP BY codigo_pais;

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
-> SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos_brutos GROUP BY codigo_pais;
+-----+
| EXPLAIN                                     |
+-----+
| -> Table scan on <temporary> (actual time=173..173 rows=3 loops=1)
  -> Aggregate using temporary table (actual time=173..173 rows=3 loops=1)
    -> Table scan on pedidos_brutos (cost=4980 rows=48435) (actual time=0.0654..86.9 rows=50000 loops=1)
    |
+-----+
```

Poca diferencia en el estimado del table scan con la realidad. (0.0654 ms a 86.9 ms) Lee las 50.000 filas, loop=1 una sola vez.

Después usa una tabla temporal para el Group By (apenas tarda nada, de 173ms a 173 ms).

La operación tarda en total 173 ms.

Fase 2: Optimización Estructural (8 puntos)

- Tras el diagnóstico inicial, transforma la tabla masiva `pedidos_brutos` en una estructura relacional eficiente.

En este caso voy a crear una nueva tabla “pedidos” y migrar los datos a ella.

```
CREATE TABLE pedidos (  
  id_operacion INT PRIMARY KEY,  
  fecha_operacion DATETIME,  
  cliente_nombre VARCHAR(255),  
  cliente_email VARCHAR(255),  
  producto_nombre VARCHAR(255),  
  categoria_nombre ENUM('Tecnología','Hogar','Deportes','Libros'),  
  precio_unitario DECIMAL(10,2),  
  cantidad_pedida INT,  
  comentario_producto TEXT,  
  codigo_pais CHAR(2)  
);
```

```
mysql> show tables;  
+-----+  
| Tables_in_global_express_db |  
+-----+  
| pedidos                      |  
| pedidos_brutos              |  
+-----+  
2 rows in set (0,01 sec)  
  
mysql> |
```

- Explica detalladamente el motivo de cada cambio realizado (por qué esa tabla, por qué ese tipo de dato o esa relación)

Establecemos una Primary Key en la tabla.

La fecha_operación debe tener un formato de fecha y hora.

En los campos cliente_nombre, cliente_email y producto_nombre se abusa de términos que se repiten:

- Cliente: "Cliente 1944",
- user: user2981@express.com
- Producto: "Producto 256".

Y en categoria_nombre, se puede usar ENUM en vez de VARCHAR, lo cual agiliza la búsqueda: "usan valores predefinidos, de esta manera ahorra espacio (MySQL lo almacena como un número y necesita procesar menos en las consultas)".+

Usamos CHAR(2), porque CHAR se usa cuando sabemos el número exacto de caracteres. En este caso siempre se usa dos caracteres para identificar al país.

precio_unitario usa DECIMAL (ni FLOAT o DOUBLE para evitar errores de redondeo).

Y en las búsqueda en comentario_producto, crear un index FULLTEXT, optimizado para largas cadenas de textos y luego usar búsquedas con MATCH y AGAINST.

Insertamos los datos:

```
INSERT INTO pedidos
SELECT
id_operacion,
STR_TO_DATE(fecha_operacion,'%d/%m/%Y %H:%i:%s'),
cliente_nombre,
cliente_email,
producto_nombre,
categoria_nombre,
precio_unitario,
cantidad_pedida,
comentario_producto,
codigo_pais
FROM pedidos_brutos;
```

ATENCIÓN: STR_TO_DATE(fecha_operacion,'%d/%m/%Y %H:%i:%s'),

Es para convertir la cadena de caracteres a un formato de Tiempo. Se usa el mismo que aparece en el script.

```
mysql> INSERT INTO pedidos
te_email,
producto_nombre,
categoria_nombre,
precio_unitario,
cantidad_pedida,
comentario_produc    -> SELECT
    -> id_operacion,
    -> STR_TO_DATE(fecha_operacion,'%d/%m/%Y %H:%i:%s'),
    -> cliente_nombre,
    -> cliente_email,
    -> producto_nombre,
    -> categoria_nombre,
    -> precio_unitario,
    -> cantidad_pedida,
    -> comentario_producto,
    -> codigo_pais
    -> FROM pedidos_brutos;
Query OK, 50000 rows affected (2,12 sec)
Records: 50000  Duplicates: 0  Warnings: 0
```

Ahora vamos a crear algunos índices para las búsquedas:

1. Búsqueda de pedidos por cliente:

```
SELECT * FROM pedidos_brutos WHERE cliente_email = 'user2774@express.com';
```

Le creamos el siguiente índice:

```
CREATE INDEX idx_cliente_email on pedidos (cliente_email);
```

```
mysql> CREATE INDEX idx_cliente_email on pedidos (cliente_email);  
Query OK, 0 rows affected (0,78 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

Y las búsquedas deben hacer con LIKE para evitar tener que leer “user” y “@express.com”.

```
SELECT * FROM pedidos_brutos WHERE cliente_email LIKE '%2774%';
```

EXPLAIN FORMAT=TREE

SELECT * FROM pedidos WHERE cliente_email LIKE '%2774%';

```
mysql> EXPLAIN FORMAT=TREE
      -> SELECT * FROM pedidos WHERE cliente_email LIKE '%2774%';
+-----+
| EXPLAIN
+-----+
| -> Filter: (pedidos.cliente_email like '%2774%') (cost=5087 rows=5518)
  -> Table scan on pedidos (cost=5087 rows=49665)
  |
+-----+
1 row in set (0.00 sec)
```

Las estimaciones son parecidas.

EXPLAIN ANALYZE FORMAT=TREE

SELECT * FROM pedidos WHERE cliente_email LIKE '%2774%';

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
      -> SELECT * FROM pedidos WHERE cliente_email LIKE '%2774%';
+-----+
---+
| EXPLAIN
  |
+-----+
---+
| -> Filter: (pedidos.cliente_email like '%2774%') (cost=5563 rows=5505) (actual time=20.8..166 rows=11 loops=1)
  -> Table scan on pedidos (cost=5563 rows=49550) (actual time=0.189..130 rows=50000 loops=1)
  |
+-----+
---+
1 row in set (0,16 sec)

mysql> |
```

Apenas hay diferencia.

2. Filtrado por categoría y rango de precios:

```
SELECT producto_nombre, precio_unitario FROM pedidos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;
```

Le creamos un índice compuesto:

```
CREATE INDEX idx_proname_preunit_categ ON pedidos (producto_nombre, precio_unitario, categoria_nombre);
```

```
mysql> CREATE INDEX idx_proname_preunit_categ ON pedidos (producto_nombre, precio_unitario, categoria_nombre);  
Query OK, 0 rows affected (0,95 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

EXPLAIN ANALYZE FORMAT=TREE

SELECT producto_nombre, precio_unitario FROM pedidos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
-> SELECT producto_nombre, precio_unitario FROM pedidos WHERE categoria_nombre = 'Hogar' AND precio_unitario > 500 ORDER BY precio_unitario DESC;
+-----+
| EXPLAIN |
+-----+
| -> Index range scan on pedidos using idx_prodname_preunit over (categoria_nombre = 'Hogar' AND 500.00 < precio_unitario) (reverse), with index condition: ((pedidos.categoria_nombre = 'Hogar') and (pedidos.precio_unitario > 500.00)) (cost=323 rows=269) (actual time=0.479..3 rows=269 loops=1) |
+-----+
1 row in set (0.00 sec)
```

Hay una considerable mejora del rendimiento usando el índice creado: Solo tiene que leer 269 filas y tarda 3 milisegundos.

3. Búsqueda de palabras clave en la descripción:

```
SELECT producto_nombre FROM pedidos WHERE comentario_producto LIKE '%calidad%';
```

Creamos el índice FULLTEXT:

```
CREATE FULLTEXT INDEX idx_comentario_producto_ft ON pedidos (comentario_producto);
```

```
EXPLAIN ANALYZE FORMAT=TREE
```

```
SELECT * FROM pedidos WHERE MATCH comentario_producto AGAINST ('calidad');
```

```
mysql> EXPLAIN ANALYZE FORMAT=TREE SELECT * FROM pedidos WHERE MATCH comentario_producto AGAINST ('calidad');
+-----+
| EXPLAIN                                     |
+-----+
| -> Filter: (match pedidos.comentario_producto against ('calidad')) (cost=1.1 rows=1) (actual time=16.7..176 rows=25026 loops=1)
|   -> Full-text index search on pedidos using idx_comentario_producto_ft (comentario_producto='calidad') (cost=1.1 rows=1) (actual time=16.7..169 rows=25026 loops=1)
|
+-----+
1 row in set (0,24 sec)
```

No hay una gran mejora en el tiempo

4. Reporte de ventas totales por país:

```
SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos GROUP BY codigo_pais;
```

Creamos un índice compuesto:

```
CREATE INDEX idx_estado_cantidadPrecio ON pedidos (codigo_pais, precio_unitario, cantidad_pedida);
```

```
EXPLAIN ANALYZE FORMAT=TREE
```

```
SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos GROUP BY codigo_pais;
```

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
-> SELECT codigo_pais, SUM(precio_unitario * cantidad_pedida) FROM pedidos GROUP BY codigo_pais;
+-----+
| EXPLAIN                                     |
+-----+
| -> Group aggregate: sum((pedidos.precio_unitario * pedidos.cantidad_pedida)) (cost=10518 rows=3) (actual time=23.9..72.1 rows=3 loops=1)
|   -> Covering index scan on pedidos using idx_estado_cantidadPrecio (cost=5563 rows=49550) (actual time=0.118..42.7 rows=50000 loops=1)
|
+-----+
```

Previamente se tarda 173 milisegundos ahora se tarda 72.1 milisegundos.

- Aplica el nuevo script a la base de datos.