ACTIVIDAD 4 - OPTIMIZACIÓN BASE DE DATOS

Cristóbal Suárez Abad

ADMINISTRACIÓN DE SISTEMAS GESTORES DE BASES DE DATOS - 2º ASIR

Índice

Optimización de Rendimiento en MySQL"

1. Escenario y Preparación

Una empresa tiene una base de datos con una tabla masiva llamada `pedidos_brutos`. Actualmente, las consultas de reportes tardan demasiado y el almacenamiento está creciendo de forma ineficiente debido a un mal diseño de tipos de datos.

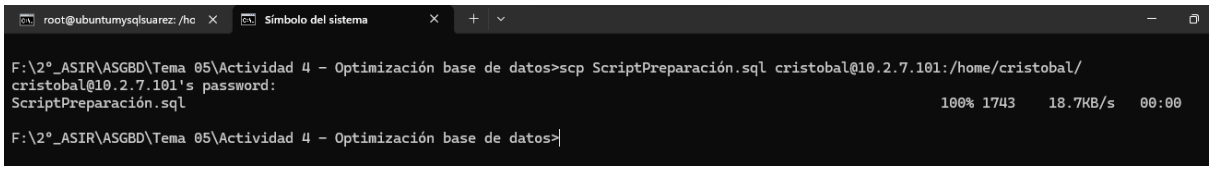
Tarea inicial: Importar el script SQL (proporcionado) que contiene con muchos registros:

En nuestro caso, en un cliente Windows, nos situamos con el CMD en el directorio donde tenemos descargado el script:

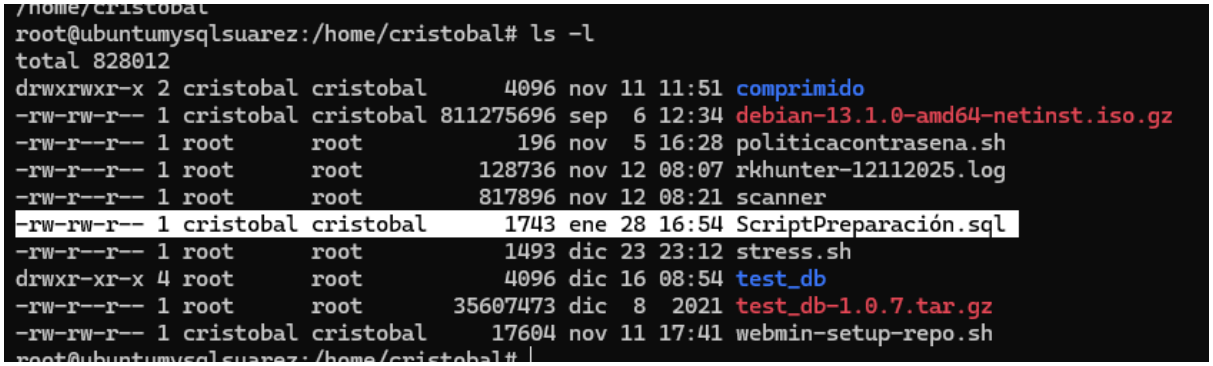
```
F:\2°_ASIR\ASGBD\Tema 05\Actividad 4 - Optimización base de datos>dir
 El volumen de la unidad F es Ventoy
 El número de serie del volumen es: DAD3-DE8B

 Directorio de F:\2°_ASIR\ASGBD\Tema 05\Actividad 4 - Optimización base de datos

28/01/2026  12:06    <DIR>          .
28/01/2026  11:13    <DIR>          ..
28/01/2026  11:02           217.294 Actividad 4 - Optimización base de datos.docx
28/01/2026  11:02           217.294 ORIGINAL - Actividad 4 - Optimización base de datos.docx
28/01/2026  12:06             1.743 ScriptPreparación.sql
               3 archivos        436.331 bytes
               2 dirs  16.916.570.112 bytes libres

F:\2°_ASIR\ASGBD\Tema 05\Actividad 4 - Optimización base de datos>
```

Ahora lo mandamos al servidor donde tenemos nuestra base de datos con el siguiente comando:

scp ScriptPreparación.sql cristobal@10.2.7.101:/home/cristobal/



Desde el servidor podemos ver que lo hemos recepcionado:

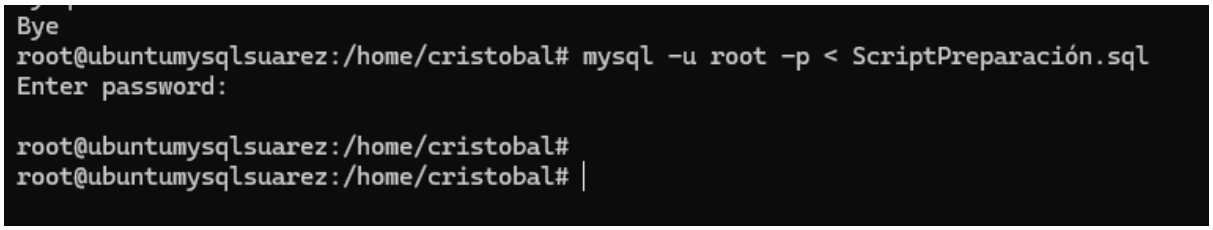


Una vez dentro del servidor de la base de datos tenemos dos opciones para introducir el script.

- **Desde la terminal de Linux**

mysql -u root -p < ScriptPreparación.sql

Tarda un poco el script.



Si entramos, podemos ver que ya está creada.

- **Desde dentro de MariaDB/MySQL**

Si ya estás logueado en el monitor de MySQL (mysql -u root -p):

SOURCE /home/cristobal/ScriptPreparación.sql;

2. Análisis del Rendimiento de Consultas

Antes de tocar la estructura, los alumnos deben identificar dónde están los cuellos de botella.

Tareas:

ATENCIÓN: Posiciónate dentro de la base de datos.

USE logistica_db;

```
mysql> USE logistica_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
```

Para saber en que base de datos estas, usa: **SELECT DATABASE();**

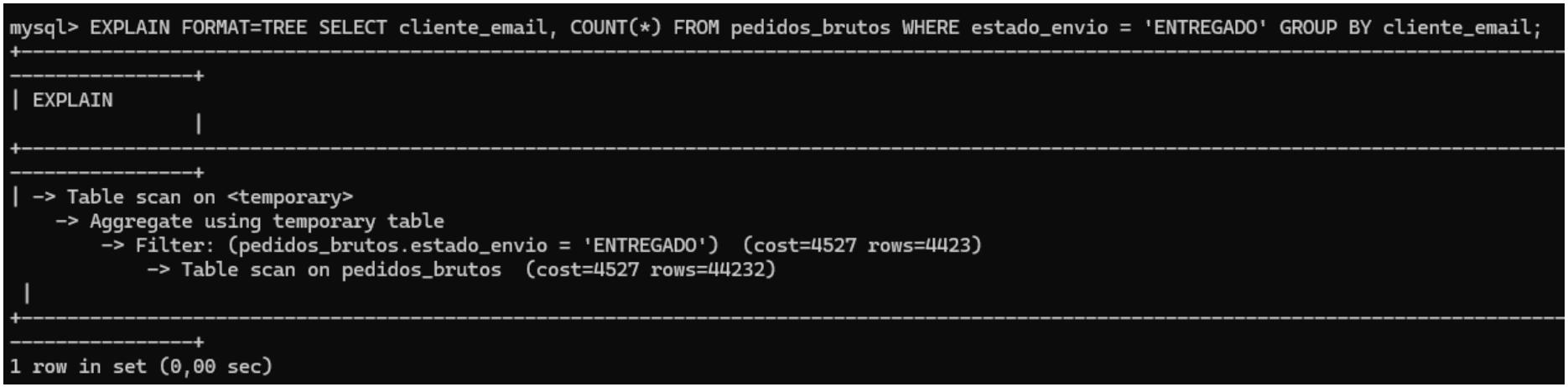
1. Utiliza consultas que creas que pueden ser utilizadas para el uso de esta tabla. Para cada consulta aplica los siguientes pasos:
    - **Uso de EXPLAIN:** Explica lo más importante de lo que ves.
    - **Uso de EXPLAIN ANALYZE:** Explica lo más importante de lo que ves.
    - **Documentación del estado inicial:** Registrar los tiempos y el plan de ejecución inicial en un breve informe.

a) Pedidos entregados por cliente:

```
SELECT cliente_email, COUNT(*) AS total_pedidos
FROM pedidos_brutos
WHERE estado_envio = 'ENTREGADO'
GROUP BY cliente_email;
```

```
mysql> SELECT cliente_email, COUNT(*) AS total_pedidos
    -> FROM pedidos_brutos
    -> WHERE estado_envio = 'ENTREGADO'
    -> GROUP BY cliente_email;
+---------------------+---------------+
| cliente_email       | total_pedidos |
+---------------------+---------------+
| email610@empresa.com |            15 |
| email435@empresa.com |            20 |
| email830@empresa.com |            14 |
| email933@empresa.com |            15 |
| email824@empresa.com |            13 |
| email504@empresa.com |            15 |
| email813@empresa.com |            19 |
| email406@empresa.com |            17 |
```

```
EXPLAIN FORMAT=TREE
SELECT cliente_email, COUNT(*)
FROM pedidos_brutos
WHERE estado_envio = 'ENTREGADO'
GROUP BY cliente_email;
```



Solo con “EXPLAIN” es un análisis teórico.

- “**Table scan**”: lee todas las filas de la tabla, malo para el rendimiento. No existe un índice útil sobre “**estado_envio**”, así que MySQL no puede ir directo a los registros “ENTREGADO” y tiene que leer todas las filas.
- “**Aggregate using temporary table**”: Para hacer el GROUP BY cliente_email, MySQL crea una tabla temporal.
- “**Filter: (estado_envio = 'ENTREGADO')**”: Este el filtro del “where” que hemos usado antes. Nos indica su coste y cuantas filas (“rows”) ha leído.
- “**Table scan on pedidos_brutos**”: nos da el resultado de escanear la tabla. En este caso es el mismo que le del filtrado.

EXPLAIN ANALYZE FORMAT=TREE
SELECT cliente_email, COUNT(*)
FROM pedidos_brutos
WHERE estado_envio = 'ENTREGADO'
GROUP BY cliente_email;

```
mysql> EXPLAIN ANALYZE FORMAT=TREE SELECT cliente_email, COUNT(*) FROM pedidos_brutos WHERE estado_envio = 'ENTREGADO' GROUP BY cliente_email;
+------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN

+------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
| -> Table scan on <temporary>  (actual time=133..133 rows=1000 loops=1)
    -> Aggregate using temporary table  (actual time=133..133 rows=1000 loops=1)
        -> Filter: (pedidos_brutos.estado_envio = 'ENTREGADO')  (cost=4527 rows=4423) (actual time=0.0869..102 rows=16583 loops=1)
            -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.0632..82.9 rows=50000 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
1 row in set (0,14 sec)
```

Con “EXPLAIN ANALYZE” es un análisis real:

- “**Table scan on pedidos_brutos**”: El coste teórico dice que tendría que leer 44.232 filas, mientras que el real ha leído 50.000. Tiempo usado: 82.9 milisegundos.
- “**Filter**”: Pasa algo parecido, esperaba leer 4.423 filas, pero al final ha tenido que leer 16.583 filas. Tiempo usado: 102 milisegundos.
- “**Aggregate**”: Es el tiempo real de crear la tabla en el directorio temporal. Tiempo usado: 133 milisegundos.
- “**Table scan on <temporary>**”: Es el coste total del ejercicio. “1000 rows” es porque crea mil filas (Hay mil clientes).
- “**loops=1**”: significa que este paso se ejecutó una sola vez.

SELECT COUNT(DISTINCT id_pedido) AS Total_Pedidos FROM pedidos_brutos;

```
mysql> SELECT COUNT(DISTINCT id_pedido) AS Total_Pedidos
    -> FROM pedidos_brutos;
+---------------+
| Total_Pedidos |
+---------------+
|         50000 |
+---------------+
1 row in set (0,09 sec)
```

Se ha leído los 50.000 pedidos. Después ha filtrado cuál estaba en “entregado” (16.583).

SELECT COUNT(DISTINCT id_pedido) AS envios_entregados FROM pedidos_brutos WHERE estado_envio = 'entregado';

```
mysql> SELECT COUNT(DISTINCT id_pedido) AS envios_entregados FROM pedidos_brutos WHERE estado_envio = 'entregado';
+--------------------+
| envios_entregados |
+--------------------+
|              16583 |
+--------------------+
1 row in set (0,11 sec)
```

Y después lo relaciona con los clientes.

SELECT COUNT(DISTINCT cliente_email) AS clientes_con_envio FROM pedidos_brutos WHERE estado_envio = 'entregado';

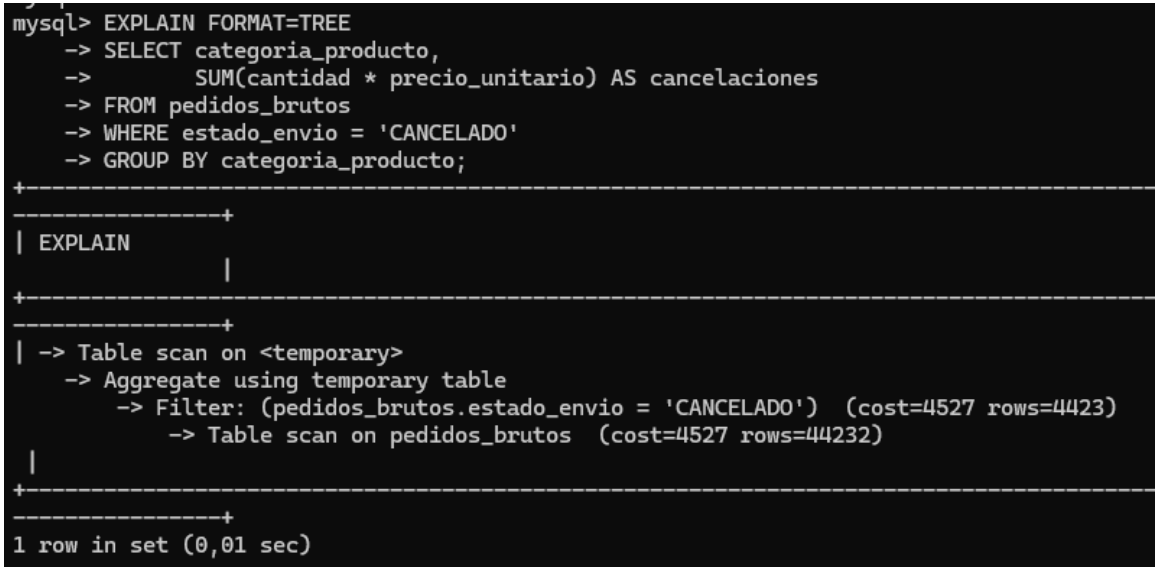
```
mysql> SELECT COUNT(DISTINCT cliente_email) AS clientes_con_envio FROM pedidos_brutos WHERE estado_envio = 'entregado';
+--------------------+
| clientes_con_envio |
+--------------------+
|               1000 |
+--------------------+
1 row in set (0,16 sec)
```

b) Coste de ventas canceladas por categoría:

```
SELECT categoria_producto,
      SUM(cantidad * precio_unitario) AS cancelaciones
FROM pedidos_brutos
WHERE estado_envio = 'CANCELADO'
GROUP BY categoria_producto;
```

```
mysql> SELECT categoria_producto,
    ->        SUM(cantidad * precio_unitario) AS cancelaciones
    -> FROM pedidos_brutos
    -> WHERE estado_envio = 'CANCELADO'
    -> GROUP BY categoria_producto;
+--------------------+--------------------+
| categoria_producto | cancelaciones      |
+--------------------+--------------------+
| Hogar              |         1263841.45 |
| Juguetes           | 1298397.7700000014 |
| Electrónica        |  1229974.580000002 |
| Ropa               |  1245286.559999999 |
+--------------------+--------------------+
4 rows in set (0,12 sec)
```

```
EXPLAIN FORMAT=TREE
SELECT categoria_producto,
      SUM(cantidad * precio_unitario) AS cancelaciones
FROM pedidos_brutos
WHERE estado_envio = 'CANCELADO'
GROUP BY categoria_producto;
```



De igual manera que con el anterior, teoriza que tendrá que leer unas 44.232 filas.

EXPLAIN ANALYZE FORMAT=TREE
SELECT categoria_producto,
SUM(cantidad * precio_unitario) AS cancelaciones
FROM pedidos_brutos
WHERE estado_envio = 'CANCELADO'
GROUP BY categoria_producto;

```
mysql> EXPLAIN ANALYZE FORMAT=TREE
ROM pedidos_brutos
WHERE estado_envio = 'CANCELADO'
GROUP BY categoria_producto;
    -> SELECT categoria_producto,
    ->        SUM(cantidad * precio_unitario) AS cancelaciones
    -> FROM pedidos_brutos
    -> WHERE estado_envio = 'CANCELADO'
    -> GROUP BY categoria_producto;
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN

+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
| -> Table scan on <temporary>  (actual time=136..136 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=136..136 rows=4 loops=1)
        -> Filter: (pedidos_brutos.estado_envio = 'CANCELADO')  (cost=4527 rows=4423) (actual time=0.403..106 rows=16697 loops=1)
            -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.321..88.7 rows=50000 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
1 row in set (0,15 sec)
```

Igual que en el anterior, lee las 50.000 filas una vez (loop=1, 135 milisegundos), filtra 16.697 (las que están en “CANCELADO”).

c) Ingresos Totales:

```
SELECT
   SUM(cantidad * precio_unitario) AS ingresos_reales
FROM pedidos_brutos
WHERE estado_envio != 'CANCELADO';
```

```
mysql> SELECT
    ->     SUM(cantidad * precio_unitario) AS ingresos_reales
    -> FROM pedidos_brutos
    -> WHERE estado_envio != 'CANCELADO';
+------------------+
| ingresos_reales  |
+------------------+
| 10116507.5899999 |
+------------------+
1 row in set (0,10 sec)
```

```
EXPLAIN FORMAT=TREE
SELECT
   SUM(cantidad * precio_unitario) AS ingresos_reales
FROM pedidos_brutos
WHERE estado_envio != 'CANCELADO';
```

```
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Aggregate: sum((pedidos_brutos.cantidad * pedidos_brutos.precio_unitario))  (cost=9566 rows=1)
    -> Filter: (pedidos_brutos.estado_envio <> 'CANCELADO')  (cost=5084 rows=44821)
        -> Table scan on pedidos_brutos  (cost=5084 rows=49801)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

```
EXPLAIN ANALYZE FORMAT=TREE
SELECT
   SUM(cantidad * precio_unitario) AS ingresos_reales
FROM pedidos_brutos
WHERE estado_envio != 'CANCELADO';
```

```
-> Aggregate: sum((pedidos_brutos.cantidad * pedidos_brutos.precio_unitario))  (cost=9566 rows=1) (actual time=125..125 rows=1 loops=1)
  -> Filter: (pedidos_brutos.estado_envio <> 'CANCELADO')  (cost=5084 rows=44821) (actual time=0.0763..110 rows=33438 loops=1)
      -> Table scan on pedidos_brutos  (cost=5084 rows=49801) (actual time=0.0672..89.5 rows=50000 loops=1)
```

Igual que en los anteriores, hay discrepancia entre las filas que en teoría cree que va a consultar (44.821) y las que al final termina usando (50.000).

d) Pedidos por rango de fechas

```
SELECT COUNT(*)
FROM pedidos_brutos
WHERE fecha_pedido BETWEEN '01/06/2023' AND '30/06/2023';
```

```
mysql> SELECT COUNT(*)
    -> FROM pedidos_brutos
    -> WHERE fecha_pedido BETWEEN '01/06/2023' AND '30/06/2023';
+----------+
| COUNT(*) |
+----------+
|    47554 |
+----------+
1 row in set (0,08 sec)

mysql> SELECT COUNT(*) FROM pedidos_brutos WHERE fecha_pedido BETWEEN '01/06/9999' AND '30/06/9999';
+----------+
| COUNT(*) |
+----------+
|    47418 |
+----------+
1 row in set (0,09 sec)
```

En este caso, además hay además un problema derivado de usar VARCHAR en la columna fecha_pedido. MySQL no está comparando fechas, sino cadenas de texto, y por lo tanto no está ordenando de manera "numérica" si no por orden alfabético[1] ("*If table_date is a varchar, the order by will order alphabetically*").

Usa la primera cifra de cada texto para establecer el rango mínimo y máximo con el que filtrar:
01/06/9999' está el 0 y 30/06/9999' el 3. Esto hace que cualquier texto que tenga 0, 1, 2 o 3 se aceptado en la búsqueda.

[1] https://stackoverflow.com/questions/5080375/order-by-date-varchar

EXPLAIN FORMAT=TREE
SELECT COUNT(*)
FROM pedidos_brutos
WHERE fecha_pedido BETWEEN '01/06/2023' AND '30/06/2023';

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT COUNT(*)
    -> FROM pedidos_brutos
    -> WHERE fecha_pedido BETWEEN '01/06/2023' AND '30/06/2023';
+----------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------+
| EXPLAIN
                                                                     |
+----------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------+
| -> Aggregate: count(0)  (cost=5019 rows=1)
    -> Filter: (pedidos_brutos.fecha_pedido between '01/06/2023' and '30/06/2023')  (cost=4527 rows=4914)
        -> Table scan on pedidos_brutos  (cost=4527 rows=44232)
 |
+----------------------------------------------------------------------------------------------------------------
mysql> EXPLAIN ANALYZE FORMAT=TREE
    -> SELECT COUNT(*)
    -> FROM pedidos_brutos
    -> WHERE fecha_pedido BETWEEN '01/06/2023' AND '30/06/2023';
+----------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------+
| EXPLAIN

                                              |
+----------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------+
| -> Aggregate: count(0)  (cost=5019 rows=1) (actual time=103..103 rows=1 loops=1)
    -> Filter: (pedidos_brutos.fecha_pedido between '01/06/2023' and '30/06/2023')  (cost=4527 rows=4914) (actual time=0.2..96.6 rows=47554 loops=1
)
        -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.19..77 rows=50000 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------
```

En el "filter" hay un error de estimación bastante alto (4.914) frente al actual (47.554).

2. Registra el **tamaño en disco** de la base de datos (usando `information_schema`).

```
SELECT
   table_schema AS bd,
   ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS tamaño_mb
FROM information_schema.tables
WHERE table_schema = 'logistica_db'
GROUP BY table_schema;
```

```
mysql> SELECT
    ->     table_schema AS bd,
    ->     ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS tamaño_mb
    -> FROM information_schema.tables
    -> WHERE table_schema = 'logistica_db'
    -> GROUP BY table_schema;
+--------------+------------+
| bd           | tamaño_mb  |
+--------------+------------+
| logistica_db |       6.52 |
+--------------+------------+
1 row in set (0,00 sec)
```

3. Optimización de la Estructura

Basándose en el análisis anterior y las deficiencias detectadas, propón y aplica los cambios estructurales que consideres necesarios. Explica el motivo de cada cambio.

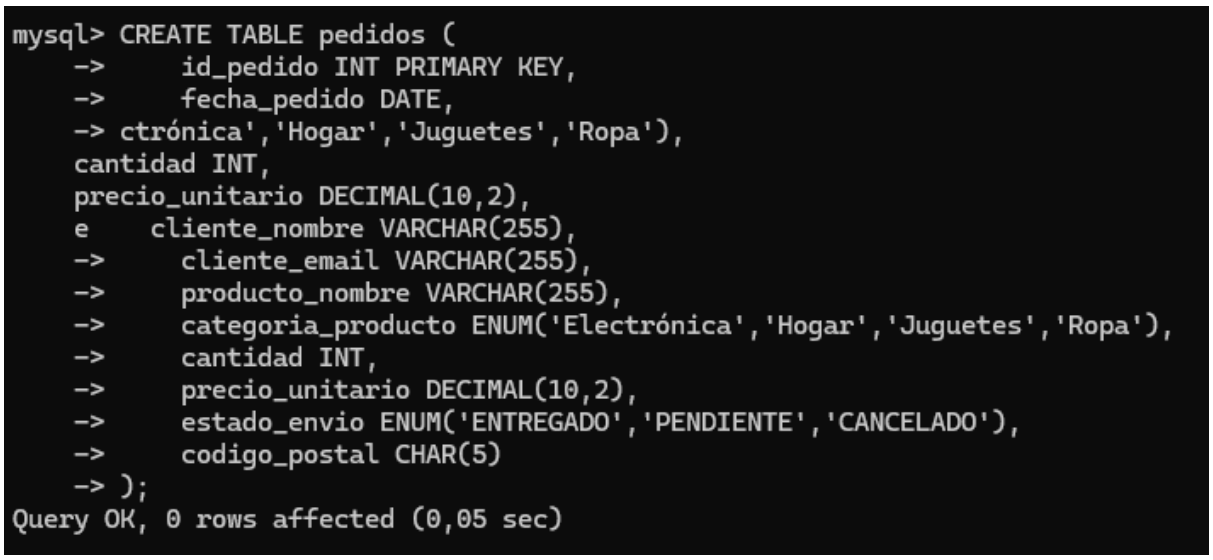
Los principales fallos detectados son:

- Tipos de datos incorrectos.
- Sin claves primarias.
- Sin índices.
- Campos repetitivos en columnas categoria_producto y estado_envio.

El principal cambio es migrar los datos a una tabla nueva:

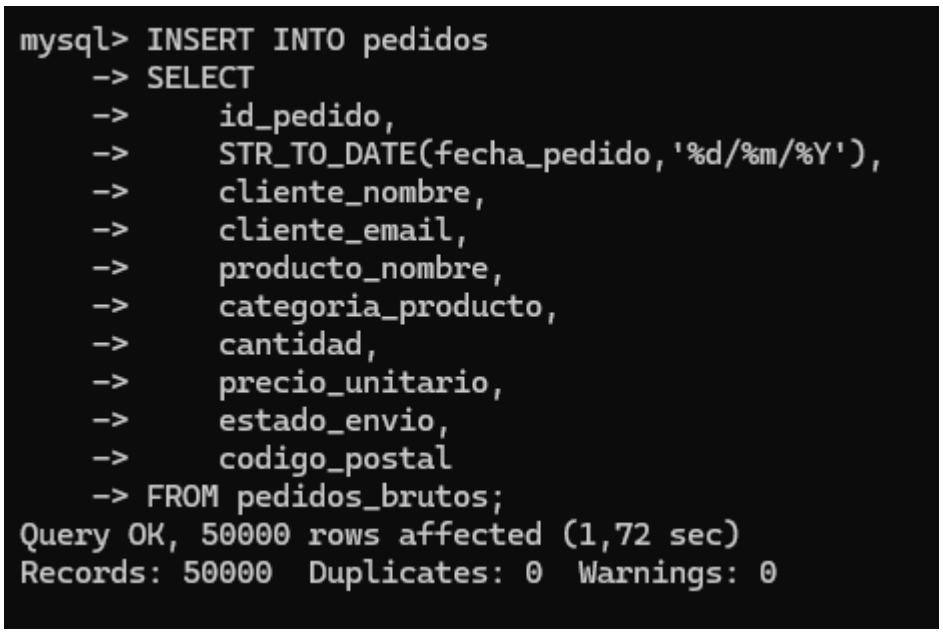
```
CREATE TABLE pedidos (
   id_pedido INT PRIMARY KEY,
   fecha_pedido DATE,
   cliente_nombre VARCHAR(50),
   cliente_email VARCHAR(100),
   producto_nombre VARCHAR(200),
   categoria_producto ENUM('Electrónica','Hogar','Juguetes','Ropa'),
   cantidad INT,
   precio_unitario DECIMAL(10,2),
   estado_envio ENUM('ENTREGADO','PENDIENTE','CANCELADO'),
   codigo_postal CHAR(5)
);
```

- Se establece una PRIMARY KEY.
- Los **campos cantidad** y **precio_unitario** pasan a usar números (INT y DECIMAL).
- Se usa ENUM para **categoria_producto** y **estado_envio** porque usan valores predefinidos, de esta manera ahorra espacio (MySQL lo almacena como un número y necesita procesar menos en las consultas.
- **fecha_pedido** pasa a usar DATE.



Ahora los migramos:

```
INSERT INTO pedidos
SELECT
   id_pedido,
   STR_TO_DATE(fecha_pedido,'%d/%m/%Y'),
   cliente_nombre,
   cliente_email,
   producto_nombre,
   categoria_producto,
   cantidad,
   precio_unitario,
   estado_envio,
   codigo_postal
FROM pedidos_brutos;
```



STR_TO_DATE(fecha_pedido,'%d/%m/%Y') → Esto convierte de texto (String) a fecha (Date) en un formato de día, mes, año.

Ahora vamos a crear algunos índices:

CREATE INDEX idx_estado ON pedidos (estado_envio);
CREATE INDEX idx_cliente ON pedidos (cliente_email);
CREATE INDEX idx_categoria_estado ON pedidos (categoria_producto, estado_envio);
CREATE INDEX idx_fecha ON pedidos (fecha_pedido);
CREATE INDEX idx_estado_ingresos ON pedidos (estado_envio, cantidad, precio_unitario);

```
mysql> CREATE INDEX idx_estado ON pedidos (estado_envio);
NDEX idx_categoria_estado ON pedidos (categoria_producto, estado_envio);
CREATE INDEX idx_fecha ON pedidos (fecha_pedido);
CREATE INDEX idx_estado_ingresos ON pedidos (estado_envio, cantidad, precio_unitario);
Query OK, 0 rows affected (0,35 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_cliente ON pedidos (cliente_email);
Query OK, 0 rows affected (0,74 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_categoria_estado ON pedidos (categoria_producto, estado_envio);
Query OK, 0 rows affected (0,35 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_fecha ON pedidos (fecha_pedido);
Query OK, 0 rows affected (0,32 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_estado_ingresos ON pedidos (estado_envio, cantidad, precio_unitario);
Query OK, 0 rows affected (0,50 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

SHOW INDEX FROM pedidos_brutos;

```
+---------+--------------+---------+------------+
 Table   | Non_unique | Key_name             | Seq_in_index | Column_name         |
| Comment | Index_comment | Visible | Expression |
---------+------------+----------------------+--------------+---------------------+
+---------+--------------+---------+------------+
 pedidos |          0 | PRIMARY              |            1 | id_pedido           |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_estado           |            1 | estado_envio        |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_cliente          |            1 | cliente_email       |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_categoria_estado |            1 | categoria_producto  |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_categoria_estado |            2 | estado_envio        |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_fecha            |            1 | fecha_pedido        |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_estado_ingresos  |            1 | estado_envio        |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_estado_ingresos  |            2 | cantidad            |
|         |            | YES     | NULL       |
 pedidos |          1 | idx_estado_ingresos  |            3 | precio_unitario     |
|         |            | YES     | NULL       |
---------+------------+----------------------+--------------+---------------------+
```

4. Validación y Comparativa

Es el momento de comprobar si los cambios han surtido efecto.

Tareas:

1. **Repetición de Pruebas:** Ejecutar exactamente las mismas consultas del apartado 2.
2. **Comparación de métricas:**
    - Comparar el tamaño en disco de la base de datos antes y después (usando `information_schema`).
    - Comparar los nuevos planes de ejecución con `EXPLAIN`.
3. **¿Qué impacto han tenido los cambios?**

a) Pedidos entregados por cliente:

```
EXPLAIN ANALYZE FORMAT=TREE
SELECT cliente_email, COUNT(*)
FROM pedidos
WHERE estado_envio = 'ENTREGADO'
GROUP BY cliente_email;
```

Original:

```
-> Table scan on <temporary>  (actual time=130..131 rows=1000 loops=1)
    -> Aggregate using temporary table  (actual time=130..130 rows=1000 loops=1)
        -> Filter: (pedidos_brutos.estado_envio = 'ENTREGADO')  (cost=4527 rows=4423) (actual time=0.097..99.3 rows=16583 loops=1)
            -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.0721..82.3 rows=50000 loops=1)
```

Optimizado:

```
-> Table scan on <temporary>  (actual time=103..103 rows=1000 loops=1)
    -> Aggregate using temporary table  (actual time=103..103 rows=1000 loops=1)
        -> Index lookup on pedidos using idx_estado (estado_envio='ENTREGADO'), with index condition: (pedidos.estado_envio = 'ENTREGADO')  (cost=2
757 rows=24927) (actual time=0.313..71.2 rows=16583 loops=1)
```

Ahora se usa un acceso por índice (**idx_estado**), en lugar de escanear las 50.000 filas de la tabla, solo necesita leer aquellas que no son necesarias (reduce el número de filas leídas en un 67%. Ahora no tiene que usar “Table scan”.

También reducción del tiempo por ser un campo que usa ENUM.

Mejora del tiempo (130 ms vs 103 ms).

b) Coste de ventas canceladas por categoría:

```
EXPLAIN ANALYZE FORMAT=TREE
SELECT categoria_producto,
     SUM(cantidad * precio_unitario) AS cancelaciones
FROM pedidos
WHERE estado_envio = 'CANCELADO'
GROUP BY categoria_producto;
```

Original:

```
-> Table scan on <temporary>  (actual time=138..138 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=138..138 rows=4 loops=1)
        -> Filter: (pedidos_brutos.estado_envio = 'CANCELADO')  (cost=4527 rows=4423) (actual time=0.435..108 rows=16697 loops=1)
            -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.336..91.6 rows=50000 loops=1)
```

Optimizada:

```
-> Table scan on <temporary>  (actual time=106..106 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=106..106 rows=4 loops=1)
        -> Index lookup on pedidos using idx_estado (estado_envio='CANCELADO'), with index condition: (pedidos.estado_envio = 'CANCELADO')  (cost=2
757 rows=24927) (actual time=0.413..70.3 rows=16697 loops=1)
```

De igual manera que el anterior, hace uso de un index (**idx_estado**), evitando tener que leer toda la tabla, solo las 16.697 líneas necesarias. Se evita usar Table scan y se mejora el tiempo (138 ms vs 106 ms). Además, ambas columnas han pasado de usar VARCHAR a INT y Decimal, lo que mejora el rendimiento.

c) Ingresos Totales:

Ejemplo que no usa índice.

```
EXPLAIN ANALYZE FORMAT=TREE
SELECT
   SUM(cantidad * precio_unitario) AS ingresos_reales
FROM pedidos
WHERE estado_envio != 'CANCELADO';
```

Original:

```
-> Aggregate: sum((pedidos_brutos.cantidad * pedidos_brutos.precio_unitario))  (cost=9566 rows=1) (actual time=115..115 rows=1 loops=1)
    -> Filter: (pedidos_brutos.estado_envio <> 'CANCELADO')  (cost=5084 rows=44821) (actual time=0.0838..102 rows=33438 loops=1)
        -> Table scan on pedidos_brutos  (cost=5084 rows=49801) (actual time=0.0783..83.5 rows=50000 loops=1)
```

Optimizado:

```
-> Aggregate: sum((pedidos.cantidad * pedidos.precio_unitario))  (cost=8397 rows=1) (actual time=64.3..64.3 rows=1 loops=1)
    -> Filter: (pedidos.estado_envio <> 'CANCELADO')  (cost=5074 rows=33237) (actual time=0.139..50.8 rows=33438 loops=1)
        -> Covering index scan on pedidos using idx_estado_ingresos  (cost=5074 rows=49855) (actual time=0.133..35.8 rows=50000 loops=1)
```

No hace “Table scan”, si no que usa el index para realizar la búsqueda. El número de filas leídas es casi el mismo, tanto en index como en el Filter, pero vemos una enorme diferencia en el tiempo usado (83 vs 35,8 y 102 vs 50.8). Esto se debe a que ambas columnas usan campos de números (INT y DECIMAL), por lo tanto, MySQL no tiene que hacer conversiones de texto a número.

d) Pedidos por rango de fechas:

```
EXPLAIN ANALYZE FORMAT=TREE
SELECT COUNT(*)
FROM pedidos
WHERE fecha_pedido BETWEEN '2023-06-01' AND '2023-06-30';
```

Hay que hacer un pequeño ajuste en la consulta, porque al usar DATE, el formato es diferente.

Original:

```
-> Aggregate: count(0)  (cost=5019 rows=1) (actual time=108..108 rows=1 loops=1)
    -> Filter: (pedidos_brutos.fecha_pedido between '01/06/2023' and '30/06/2023')  (cost=4527 rows=4914) (actual time=0.431..101 rows=47554 loops=1)
        -> Table scan on pedidos_brutos  (cost=4527 rows=44232) (actual time=0.407..82 rows=50000 loops=1)
```

Optimizado:

```
-> Aggregate: count(0)  (cost=1231 rows=1) (actual time=6.59..6.59 rows=1 loops=1)
    -> Filter: (pedidos.fecha_pedido between '2023-06-01' and '2023-06-30')  (cost=822 rows=4086) (actual time=0.286..6.06 rows=4086 loops=1)
        -> Covering index range scan on pedidos using idx_fecha over ('2023-06-01' <= fecha_pedido <= '2023-06-30')  (cost=822 rows=4086) (actual time=0.266..2.66 rows=4086 loops=1)
```

Este es el que mayor mejoría muestra, pasando de 108 ms a 6.59 ms. Con el uso del índice pasa de mirar las 50.000 a solo 4.086 filas. También cuenta el uso de DATE para el campo, que ahora si permite hacer bien las búsquedas de fechas. El coste estimado también es menor (ahora el optimizador sí puede estimar bien).

Comparar el tamaño en disco de la base de datos antes y después (usando `information_schema`).

Original:

```
SELECT
    table_schema AS bd,
    ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS tamaño_mb
FROM information_schema.tables
WHERE table_schema = 'logistica_db'
GROUP BY table_schema;
```

```
mysql> SELECT
    ->     table_schema AS bd,
    ->     ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS tamaño_mb
    -> FROM information_schema.tables
    -> WHERE table_schema = 'logistica_db'
    -> GROUP BY table_schema;
+--------------+------------+
| bd           | tamaño_mb  |
+--------------+------------+
| logistica_db |       6.52 |
+--------------+------------+
1 row in set (0,00 sec)
```

En esta ocasión, como hemos pasado los datos a una nueva tabla, para que se vea bien la diferencia he borrado la original.

Optimizada:

```
mysql> SELECT
    ->     table_schema AS bd,
    ->     ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS tamaño_mb
    -> FROM information_schema.tables
    -> WHERE table_schema = 'logistica_db'
    -> GROUP BY table_schema;
+--------------+------------+
| bd           | tamaño_mb  |
+--------------+------------+
| logistica_db |       5.52 |
+--------------+------------+
1 row in set (0,00 sec)
```