Unknown Title

vamos a repasar todo esto. Empezaremos por el primer CASO. Quiero que los expliques de manera detallada:

Contenido

CASO 6. 5

CASO 7. 6

CASO 9. 7

CASO 10. 8

CASO 11. 8

CASO 12. 9

Crea

las tablas y/o índices necesarios para cada uno de los siguientes casos.

CASO 1

Situación: Una tienda online tiene una tabla pedidos con 10 millones de filas. Las consultas

que buscan pedidos de un cliente específico tardan varios minutos en

completarse.

Asumiendo

que la columna donde están las “id” de los usuarios sea “id_cliente”.

CREATE INDEX idx_pedidos_cliente

ON pedidos (id_cliente);

CASO 2

Situación: Una plataforma inmobiliaria permite

filtrar por ciudad y precio simultáneamente. Las consultas suelen ser: SELECT *

FROM viviendas WHERE ciudad = 'Madrid' AND precio < 300000 ORDER BY precio;.

CREATE

INDEX idx_ciudad_precio ON viviendas (ciudad, precio);

CASO 3

Situación: Una nueva red social necesita asegurar

que ningún usuario registre el mismo correo electrónico. Además, quieren que

los usuarios puedan buscar palabras clave dentro de sus "biografías"

(campos de texto largo).

Para

garantizar que no registren el mismo correo usamos “Índice Único (UNIQUE)”:

CREATE

UNIQUE INDEX idx_email_unico ON usuarios (email);

Y para

búsquedas dentro de texto usamos “Índice de Texto Completo (FULLTEXT)”:

CREATE

FULLTEXT INDEX idx_bio_user ON usuarios (biografia);

Las

búsquedas se deben hacer ahora con “Match” y “Against”[1][2].

SELECT *

FROM usuarios WHERE MATCH (biografia) AGAINST ('economía');

Aquí

tienes una ampliación de la actividad con 4 casos complejos adicionales.

Estos están diseñados para que el alumno no solo decida "poner un

índice", sino que tenga que analizar el orden de las columnas, el tipo de

motor y el impacto en el almacenamiento.

CASO 4

Situación: Una plataforma de gestión de empleados

tiene una tabla nomina con 5 millones de registros. Las consultas más

frecuentes son:

SELECT

* FROM nomina WHERE departamento_id = 5 AND año = 2023;

SELECT

* FROM nomina WHERE departamento_id = 5;

SELECT

* FROM nomina WHERE año = 2023; (Esta es la que más tarda).

El

administrador propone crear un único índice compuesto: CREATE

INDEX idx_dep_año ON nomina (departamento_id, año);.

- El Reto: ¿Por

qué este índice no acelera la consulta número 3? ¿Cómo debería reestructurarse

la estrategia de indexación para cubrir las tres consultas de forma eficiente

sin penalizar demasiado las inserciones?

Porque los

índices compuestos siguen la regla de “Regla del Prefijo de Izquierda”,

por lo cual se filtra desde la columna de la izquierda a la derecha; pero no

sirve de nada si solo estás usando el de la derecha.

En este caso

sirve para las dos primeras. Para la tercera, se debería crear una propia:

CREATE

INDEX idx_año ON nomina (año);

CASO 5

Situación: Un periódico digital tiene una tabla articulos con un campo contenido de tipo LONGTEXT. Actualmente, usan la siguiente consulta

para buscar palabras clave: SELECT titulo FROM articulos WHERE

contenido LIKE '%economía%';

A pesar

de tener un índice normal en la columna contenido

(contenido_index), la

consulta tarda más de 10 segundos y el servidor se bloquea cuando hay muchos

usuarios buscando. Borra y crea los índices que consideres necesarios.

Están usando

un “Índice Normal (B-Tree)” para hacer búsquedas en un campo “LONGTEXT”,

el cual no está diseñado para eso. Debe usar un “Índice de Texto Completo

(FULLTEXT)”.

Borrar el

índice anterior: DROP INDEX contenido_index ON articulos;

Crear uno

nuevo: CREATE FULLTEXT INDEX idx_contenido_ft ON articulos (contenido);

Ahora se

recomienda cambiar la consulta[3]:

SELECT titulo FROM articulos WHERE MATCH

(contenido) AGAINST ('economía');

CASO 6

Situación: Un alumno recibe una base de datos

heredada de un desarrollador anterior. Al ejecutar SHOW

INDEX FROM productos;,

observa lo siguiente:

PRIMARY

KEY en id

INDEX

idx_categoria en categoria_id

INDEX

idx_cat_prov en (categoria_id,

proveedor_id)

UNIQUE

INDEX idx_slug en

slug_producto

- ¿Cambiarías

algo? Hazlo

Si, el segundo, “INDEX idx_categoria en categoria_id”, es innecesario porque el 3 (que es compuesto), ya cubre

esa opción con “Regla del Prefijo de Izquierda”.

Así, que lo borraría:

DROP INDEX idx_categoria ON productos;

CASO 7

Situación: Una multinacional quiere generar un

reporte: SELECT * FROM ventas WHERE pais = 'España'

AND estado_pedido = 'Completado';. La columna pais tiene 200 valores distintos (alta cardinalidad

relativa), mientras que estado_pedido solo tiene 3 valores: 'Pendiente', 'Cancelado',

'Completado' (baja cardinalidad).

El alumno

dispone de dos opciones:

- Opción A:

Crear un índice en estado_pedido.

- Opción B:

Crear un índice en pais.

- Opción C:

Crear un índice compuesto (pais, estado_pedido).

- Razona

cada una de las opciones

Opción A: Una mala opción porque apenas mejoraría el rendimiento. “Evitar índices en columnas con baja cardinalidad (pocos valores distintos como género o booleanos). No aportan ventajas significativas.”

Opción B: Mejor que la anterior. Al contrario que la “A”, esta tiene una alta cardinalidad y mejoraría el rendimiento.

Opción C: La mejor opción, porque es capaz de filtrar por ambos criterios. Además, siguiendo la Regla del Prefijo Más a la Izquierda, filtrará primero por “país”, quitándose de encima muchos registros innecesarios.

CASO 8

Situación: Una aplicación de marketing ejecuta miles de veces por segundo la consulta: SELECT email FROM suscriptores WHERE estado = 'activo';

Actualmente hay un índice en la columna estado. El servidor tiene mucha carga de I/O (lectura de disco) porque, tras encontrar el registro en el índice, MySQL debe ir a la tabla principal a buscar el email.

Eso es

porque está usando un “Índice Normal” en vez de uno compuesto. Debería crear uno así:

CREATE INDEX

idx_estado_email ON suscriptores (estado, email);

CASO 9

Situación: Tenemos una tabla usuarios donde el id es INT y una tabla logs_actividad donde el campo usuario_id fue creado por error como VARCHAR(20). Al hacer un JOIN entre ambas, la consulta es

extremadamente lenta, a pesar de que ambos campos están indexados.

- ¿Qué propones para mejorar esto?

Convertir

“usuario_id” a INT[4].

ALTER

TABLE logs_actividad MODIFY usuario_id INT NOT NULL;

Indicamos el

“NOT NULL”, porque al ser una “PRIMARY KEY”, si lo modificamos se puede

“perder” esa característica, por eso es bueno definirla de nuevo; por si acaso.

CASO 10

Situación: Un panel de control financiero muestra

las ventas de un mes específico ordenadas por fecha: SELECT *

FROM facturas WHERE empresa_id = 45 AND fecha BETWEEN '2023-01-01' AND

'2023-01-31' ORDER BY fecha DESC;

Crear un índice compuesto que recoja ambos campos[5].

Ayuda a mejorar el rendimiento: “Mejora significativa en operaciones

WHERE, JOIN, ORDER BY y GROUP BY, las más utilizadas en consultas complejas.”

CREATE INDEX idx_empresa_fecha ON facturas (empresa_id,

fecha DESC);

O sin “DESC”.

CASO 11

Situación: Una base de datos de ciberseguridad

almacena millones de URLs en una columna url_detectada (VARCHAR 2048). Quieren indexar esta columna para búsquedas rápidas, pero el índice resultante ocupa más que la propia tabla y ralentiza todo el sistema.

Usamos “Prefijos de Columna”, los cuales permiten indexar solo los primeros “N” caracteres, permitiendo tener un índice más pequeño.

Sería algo así (si la tabla se llamase “tabla_url”).

```
CREATE INDEX idx_url_corto ON tabla_url (url_detectada(100));
```

El único problema es que no distingue URLs que solo difieren al final.

CASO 12

Situación: El equipo de desarrollo quiere borrar un índice que parece no usarse (idx_antiguo), pero tienen miedo de que al borrarlo la base de datos de producción colapse si alguna consulta crítica sí lo usaba.

- El Reto:

Proponer un método para "desactivar" el índice sin borrarlo físicamente, permitiendo que el optimizador de MySQL lo ignore pero que se pueda recuperar instantáneamente si algo sale mal.

- Concepto clave: Invisible Indexes (disponible desde MySQL 8.0) y monitorización del rendimiento.

“Invisible indexes make it possible to test the effect of removing an index on query performance, without making a destructive change that must be undone should the index turn out to be required. Dropping and re-adding an

index can be expensive for a large table, whereas making it invisible and

visible are fast, in-place operations.”[6]

Gestión de Índices Invisibles:

- Podemos usar primero un “ALTER TABLE”

y convertirlo en invisible para probar que no pasa nada:

ALTER TABLE nombre_tabla ALTER INDEX idx_antiguo

INVISIBLE;

- Más adelante podemos borrarla si no surgen

problemas:

DROP INDEX idx_antiguo ON nombre_tabla;

- Si hay problemas, se recupera en un

segundo con:

ALTER TABLE nombre_tabla ALTER INDEX idx_antiguo

VISIBLE;

[1] https://codersfree.com/courses-status/aprende-php-y-mysql-desde-cero/indice-fulltext-en-mysql

[2] https://www.datacamp.com/doc/mysql/mysql-match-against

[3] https://codersfree.com/courses-status/aprende-php-y-mysql-desde-cero/indice-fulltext-en-mysql

[4] https://forum.espocrm.com/forum/general/58093-how-to-change-the-column-type-id-from-varchar-to-int

[5] https://use-the-index-luke.com/sql/sorting-grouping/indexed-order-by

[6] https://dev.mysql.com/doc/refman/9.1/en/invisible-indexes.html

¡Excelente iniciativa! Vamos a diseccionar estos casos. La optimización de bases de datos no es solo escribir código, sino entender cómo el motor de la base de datos "piensa" para encontrar la información más rápido.

Comencemos con el análisis detallado del **Caso 1**, que es la base fundamental de la indexación.

CASO 1: Búsqueda por ID de Cliente en Grandes Volúmenes

La Situación

Tenemos una tabla de `pedidos` con **10 millones de filas**. Sin un índice, cuando pides los registros de un cliente (`id_cliente = 500`), MySQL tiene que hacer lo que se llama un **Full Table Scan**.

> **Analogía:** Es como buscar un nombre específico en un libro de 1.000 páginas que no tiene índice al final y cuyos nombres no están en orden alfabético. Tienes que leer cada página, una por una, hasta el final.

El Problema

El tiempo de respuesta es de minutos porque el disco duro tiene que leer 10 millones de registros para asegurarse de que no se olvidó ninguno. Esto genera una carga de I/O (lectura/escritura) altísima.

La Solución: El Índice B-Tree

Al ejecutar:

```
CREATE INDEX idx_pedidos_cliente ON pedidos (id_cliente);
```

Lo que ocurre internamente es:

1. **Creación de una estructura paralela:** MySQL crea un árbol balanceado (B-Tree).
2. **Orden lógico:** Guarda los `id_cliente` de forma ordenada junto con un "puntero" (la dirección física) hacia la fila completa en la tabla original.
3. **Velocidad de búsqueda:** En lugar de leer 10 millones de filas, el motor realiza una búsqueda logarítmica. Para 10 millones de filas, ¡un índice puede encontrar el dato en apenas **20-25 saltos** de lectura!

Beneficios y Costes

- **Ventaja:** Las consultas pasan de tardar minutos a milisegundos.
- **Coste:** * **Espacio en disco:** El índice ocupa espacio adicional.
    - **Escritura (INSERT/UPDATE/DELETE):** Cada vez que añades un pedido, MySQL debe actualizar también el índice, lo que hace las inserciones un poquito más lentas (aunque en este caso compensa totalmente).

¿Cómo quieres proceder?

He detallado el concepto de búsqueda lineal vs. búsqueda indexada. **¿Te gustaría que pasemos ahora al Caso 2 (Índices Compuestos) o tienes alguna duda sobre cómo se almacena físicamente el índice en el Caso 1?**

CASO 2

El **CASO 2** nos introduce en el mundo de los **índices compuestos** (o multicolumna), que son vitales cuando las consultas dependen de varios factores a la vez.

CASO 2: Filtrado por Ciudad y Precio (Inmobiliaria)

La Situación

La consulta es: `SELECT * FROM viviendas WHERE ciudad = 'Madrid' AND precio < 300000 ORDER BY precio;`

Aquí el motor de búsqueda tiene tres retos en una sola sentencia:

1. **Filtrar** por una igualdad (`ciudad = 'Madrid'`).
2. **Filtrar** por un rango (`precio < 300000`).
3. **Ordenar** los resultados (`ORDER BY precio`).

La Solución Propuesta

```
CREATE INDEX idx_ciudad_precio ON viviendas (ciudad, precio);
```

¿Por qué esta estructura es tan eficiente?

Para entenderlo, imagina que el índice es como una **guía telefónica** (donde primero se ordena por Apellido y luego por Nombre):

1. **Regla del Prefijo de Izquierda:** El índice agrupa todas las propiedades de 'Madrid' juntas. Como 'Madrid' es una coincidencia exacta, el motor "salta" directamente a esa sección del índice.
2. **Orden Físico:** Dentro de la sección de 'Madrid', los precios ya están **almacenados de forma ordenada** (de menor a mayor).
3. **Doble Beneficio (Filtrado + Orden):**
    - **Filtrado:** El motor recorre los precios hasta llegar al 300,000 y se detiene. No tiene que mirar el resto.
    - **Ordenación:** Como el índice ya está ordenado por precio, MySQL **se ahorra el paso de "Sort"** (ordenar en memoria). Los datos salen listos para mostrarse.

¿Qué pasaría si el índice fuera al revés? (`precio, ciudad`)

Si creáramos el índice como `(precio, ciudad)`, sería mucho menos eficiente para esta consulta. El motor tendría que mirar todos los precios menores a 300,000 de **todas** las ciudades del país y luego buscar cuáles son de 'Madrid'.

> **Regla de oro:** En un índice compuesto, coloca primero las columnas que usen comparaciones de igualdad (=) y después las que usen rangos (<, >, `BETWEEN`) o necesiten ordenación.

¿Continuamos con el Caso 3?

En el siguiente caso veremos cómo manejar **correos únicos** y cómo buscar dentro de **biografías largas** (Full-Text). ¿Te parece bien que avancemos o quieres profundizar en el orden de los índices compuestos?

CASO 3

El **CASO 3** es muy interesante porque nos enfrenta a dos necesidades totalmente distintas en una misma tabla: la **integridad de los datos** (que no se repitan correos) y la **búsqueda de lenguaje natural** (buscar palabras en biografías).

CASO 3: Red Social (Emails Únicos y Búsquedas en Biografías)

Parte A: El Índice Único (UNIQUE)

Situación: Queremos evitar que dos personas se registren con el mismo email.

```
CREATE UNIQUE INDEX idx_email_unico ON usuarios (email);
```

- **Cómo funciona:** A diferencia de un índice normal, el índice `UNIQUE` actúa como un **vigilante**. Cada vez que alguien intenta insertar un nuevo correo, MySQL comprueba en el árbol del índice si ya existe.
- **El beneficio:** No necesitas programar una consulta previa tipo `SELECT` para ver si el correo existe antes de registrar al usuario. La propia base de datos garantiza la integridad. Si el correo existe, lanzará un error y rechazará la inserción.

Parte B: Búsqueda en Biografías (FULLTEXT)

Situación: Los usuarios escriben textos largos en sus biografías y queremos buscar palabras como "economía", "música" o "viajes".

El error común: Usar un índice normal y hacer un `LIKE '%economía%'`.

- El problema del `LIKE` con el símbolo `%` al principio es que **inutiliza cualquier índice normal**. El motor tiene que leer cada biografía palabra por palabra (Full Table Scan).

La solución:

```
CREATE FULLTEXT INDEX idx_bio_user ON usuarios (biografia);
```

- **Cómo funciona:** En lugar de un árbol B-Tree estándar, el índice FULLTEXT crea un **índice invertido**. Es muy parecido al índice temático que encuentras al final de un libro de texto:
  - Palabra "economía" -> aparece en los usuarios 5, 20 y 45.
  - Palabra "viajes" -> aparece en los usuarios 1, 5 y 100.
- **La consulta especial:**

  ```
  SELECT * FROM usuarios
  WHERE MATCH (biografia) AGAINST ('economía');
  ```

  Esta sintaxis es muchísimo más rápida porque el motor no "lee" el texto, sino que consulta directamente su lista de palabras clave.

Diferencias Clave:

1. **Índice Normal:** Ideal para valores exactos o rangos (números, fechas, nombres cortos).
2. **Índice Full-Text:** Diseñado para bloques de texto (LONGTEXT, TEXT) donde buscamos palabras sueltas dentro del contenido.

¿Pasamos al CASO 4? Aquí es donde la cosa se pone difícil, porque analizaremos por qué un índice compuesto puede "fallar" dependiendo de cómo hagamos la consulta (la famosa **Regla del Prefijo de Izquierda**).

CASO 4

El **CASO 4** es fundamental para entender el "talón de Aquiles" de los índices compuestos. Aquí es donde muchos desarrolladores junior cometen errores de rendimiento.

CASO 4: Nóminas y la Regla del Prefijo de Izquierda

La Situación

Tienes una tabla de `nomina` con 5 millones de registros y tres consultas frecuentes:

1. `SELECT * FROM ... WHERE departamento_id = 5 AND año = 2023;` (Filtra por ambos).
2. `SELECT * FROM ... WHERE departamento_id = 5;` (Filtra solo por departamento).
3. `SELECT * FROM ... WHERE año = 2023;` (**La que más tarda**).

El Problema del Índice Único

El administrador creó este índice:

```
CREATE INDEX idx_dep_año ON nomina (departamento_id, año);
```

¿Por qué falla en la consulta 3?

Imagina que este índice es un **directorio telefónico** organizado primero por **Ciudad** (departamento) y luego por **Nombre** (año).

- Si buscas a alguien en "Madrid" y que se llame "Juan" (Consulta 1), lo encuentras rápido.
- Si buscas a todos los de "Madrid" (Consulta 2), vas a la sección de Madrid y los tienes todos juntos.
- **Pero**, si buscas a todos los que se llamen "Juan" independientemente de la ciudad (Consulta 3), el directorio no te sirve. Tienes que leer **todas las ciudades** para encontrar a los "Juanes" repartidos por todo el libro.

En bases de datos, esto se llama la **Regla del Prefijo de Izquierda**: Un índice compuesto solo es útil si la consulta utiliza la **primera columna** definida en el índice. Si te saltas la primera y vas directo a la segunda, el motor de la base de datos no puede "saltar" a los datos y acaba haciendo un escaneo completo de la tabla.

La Estrategia de Reestructuración

Para cubrir las tres consultas de forma eficiente sin crear índices innecesarios:

1. **Mantener el índice compuesto:** `(departamento_id, año)`. Este cubre perfectamente las consultas 1 y 2.
2. **Crear un índice adicional:**

   ```
   CREATE INDEX idx_año ON nomina (año);
   ```

   Este índice específico acelerará la consulta 3 (la que más tardaba).

   Nota de experto: Aunque ahora tenemos dos índices, la mejora en las búsquedas compensa el pequeño coste extra de almacenamiento y el ligero impacto en la velocidad al insertar nuevas nóminas.

¿Pasamos al CASO 5? Vamos a ver qué sucede cuando intentamos usar un índice "normal" en un campo de texto gigante (LONGTEXT) y por qué eso puede bloquear tu servidor.

CASO 5

El **CASO 5** es un escenario de "emergencia" clásico. Aquí vemos cómo una mala elección del tipo de índice puede no solo ralentizar una consulta, sino llegar a **colapsar un servidor** entero debido al consumo de recursos.

CASO 5: El Periódico Digital y el LONGTEXT

El Escenario

Un periódico tiene miles de artículos. El campo `contenido` es de tipo LONGTEXT (puede guardar libros enteros dentro). Actualmente, usan: `SELECT titulo FROM articulos WHERE contenido LIKE '%economía%';`

A pesar de tener un índice normal (`B-Tree`), el servidor se bloquea.

¿Por qué el índice normal no funciona aquí?

Hay dos razones técnicas por las cuales el índice actual es inútil:

1. **El problema del comodín inicial (%):** Cuando usas `LIKE '%palabra%'`, el motor de búsqueda no puede usar un índice B-Tree. Los B-Tree están diseñados para buscar "desde el principio" de la cadena. Si pones el porcentaje al inicio, obligas a MySQL a escanear cada byte de cada artículo.
2. **Límites de tamaño:** Un índice B-Tree normal tiene un límite de longitud (generalmente 767 o 3072 bytes). Un LONGTEXT puede medir gigabytes; el índice normal solo estaría guardando el "principio" del texto, lo cual no sirve para buscar palabras en medio del cuerpo del artículo.

La Solución Técnica

Para arreglar esto, debemos cambiar la **estrategia de indexación**:

1. **Eliminar lo que no sirve:**

```
DROP INDEX contenido_index ON articulos;
```

2. **Crear un índice de lenguaje natural:**

```
CREATE FULLTEXT INDEX idx_contenido_ft ON articulos (contenido);
```

3. **Cambiar la sintaxis de la consulta:**

```
SELECT titulo FROM articulos
WHERE MATCH (contenido) AGAINST ('economía');
```

El resultado del cambio

- **Velocidad:** El índice FULLTEXT funciona como el índice de palabras clave al final de un libro. MySQL ya sabe exactamente en qué filas aparece la palabra "economía" sin tener que leer los textos.
- **Estabilidad:** Al dejar de hacer escaneos completos (Full Table Scans) en campos tan pesados, el uso de CPU y Disco cae drásticamente, evitando que el servidor se bloquee.

¿Pasamos al CASO 6? En el siguiente caso, actuaremos como "detectives de código" para limpiar una base de datos heredada que tiene índices redundantes (que sobran). ¿Listo para optimizar?

CASO 6

En el **CASO 6**, te pones el traje de "consultor senior". A menudo, heredarás bases de datos de desarrolladores que, por miedo o desconocimiento, crearon índices de más. Esto se conoce como **redundancia de índices**.

CASO 6: Limpieza de una Base de Datos Heredada

El Diagnóstico

Al ejecutar `SHOW INDEX FROM productos;`, encontramos esta configuración:

1. `PRIMARY KEY` en `id`. (Correcto, es la identidad).
2. `INDEX idx_categoria` en `categoria_id`.
3. `INDEX idx_cat_prov` en `(categoria_id, proveedor_id)`.
4. `UNIQUE INDEX idx_slug` en `slug_producto`. (Correcto, para URLs amigables únicas).

El Problema: El Índice Redundante

El índice número 2 (`idx_categoria`) es **totalmente innecesario**.

¿Por qué? Por la **Regla del Prefijo de Izquierda** que vimos en el Caso 4. El índice número 3 es un índice compuesto que empieza precisamente por `categoria_id`.

- Si alguien busca por `categoria_id`, el motor de MySQL puede usar la primera parte del índice compuesto (`idx_cat_prov`).

- Tener el índice individual (`idx_categoria`) solo sirve para ocupar espacio en el disco y ralentizar las operaciones de `INSERT` y `UPDATE`, ya que MySQL tiene que actualizar dos índices que contienen la misma información inicial.

La Acción

Debemos eliminar el índice que sobra para aligerar la base de datos:

```
DROP INDEX idx_categoria ON productos;
```

Resumen de la lógica de optimización:

- Si tienes un índice en `(A, B)`, ya tienes cubiertas las búsquedas solo por A.
- **No** tienes cubiertas las búsquedas solo por B.
- Por lo tanto, `INDEX (A)` es redundante si ya existe `INDEX (A, B)`.

¿Pasamos al CASO 7? Este es muy interesante porque trata sobre la **cardinalidad**. Aprenderás por qué indexar una columna como "Género" o "Estado de pedido" a veces es una pérdida de tiempo. ¿Seguimos?

CASO 7

El **CASO 7** nos introduce en un concepto vital: la **Cardinalidad**. No todos los datos son iguales ante los ojos de un índice, y saber elegir cuál poner primero puede marcar la diferencia entre una consulta instantánea y una lenta.

CASO 7: Multinacional, Países y Cardinalidad

La Situación

Queremos optimizar la consulta: `SELECT * FROM ventas WHERE pais = 'España' AND estado_pedido = 'Completado';`

Tenemos dos columnas candidatas:

- **pais:** 200 valores distintos (**Alta cardinalidad relativa**).
- **estado_pedido:** Solo 3 valores ('Pendiente', 'Cancelado', 'Completado') (**Baja cardinalidad**).

Análisis de las Opciones

Opción A: Índice en `estado_pedido` (Mala)

Si indexamos solo por estado, cuando busquemos "Completado", el índice nos devolverá aproximadamente **1/3 de la tabla** (millones de filas). El motor de la base de datos tendrá que trabajar muchísimo filtrando después por país. Los índices en columnas con muy pocos valores distintos no suelen ser eficientes.

Opción B: Índice en `pais` (Buena)

Como hay 200 países, al filtrar por "España" estamos descartando el **99.5% de los datos** de golpe. Es mucho más selectivo que la Opción A y reduce drásticamente el trabajo del procesador.

Opción C: Índice compuesto `(pais, estado_pedido)` (La mejor)

Esta es la opción ganadora. ¿Por qué?

1. **Filtro total:** El índice nos lleva directamente a los registros que cumplen **ambas** condiciones.
2. **Orden estratégico:** Al poner `pais` a la izquierda (el de mayor cardinalidad), el motor de búsqueda "poda" el árbol de datos mucho más rápido desde el primer paso.

   Regla de oro: Al crear índices compuestos, coloca las columnas **más selectivas** (las que tienen más valores únicos) a la **izquierda**. Esto ayuda al optimizador a descartar la mayor cantidad de filas irrelevantes lo antes posible.

¿Seguimos con el CASO 8? En el próximo caso veremos un truco avanzado: cómo hacer que una consulta ni siquiera tenga que tocar la tabla principal usando un **Índice de Cobertura (Covering Index)**. ¿Te interesa?

CASO 8

El **CASO 8** es una de mis técnicas favoritas porque es el "hack" definitivo para reducir la carga de lectura en el disco: el **Índice de Cobertura (Covering Index)**.

CASO 8: La Aplicación de Marketing y el Índice de Cobertura

La Situación

La consulta se ejecuta miles de veces por segundo: `SELECT email FROM suscriptores WHERE estado = 'activo';`

Actualmente hay un índice en `estado`. El problema es que el servidor sufre mucha carga de **I/O (Entrada/Salida)**.

¿Qué está pasando "bajo el capó"?

Cuando MySQL usa el índice actual en `estado`:

1. Busca en el índice quiénes están "activos".
2. El índice le dice: *"El usuario está en la posición física X del disco"*.
3. MySQL tiene que **ir a la tabla principal** (hacer un "Bookmark Lookup") para leer la columna `email`, porque el índice de `estado` no sabe cuál es el email.

Con miles de consultas por segundo, el cabezal del disco (o el bus de datos SSD) se vuelve loco saltando del índice a la tabla una y otra vez.

La Solución: El Índice de Cobertura

Creamos un índice compuesto que incluya **todo lo que la consulta necesita**:

```
CREATE INDEX idx_estado_email ON suscriptores (estado, email);
```

¿Por qué esto es mágico?

Ahora, cuando MySQL busca los suscriptores "activos", encuentra el `email` **dentro del propio índice**.

- El motor dice: *"Ya tengo el estado y ya tengo el email aquí mismo, no necesito ir a buscar la fila real a la tabla"*.
- La tabla principal ni se toca. Todo se resuelve en la memoria del índice.

  Resultado: La carga de I/O desaparece casi por completo y la respuesta es instantánea. Es como si en lugar de ir a la sección de "Frutería" del supermercado para ver el precio de las manzanas, el mapa de la entrada ya te dijera el precio directamente.

¿Pasamos al CASO 9? Entraremos en el terreno de los errores de diseño de tipos de datos: cuando intentas juntar (JOIN) un número con un texto. ¡Un clásico de los quebraderos de cabeza!

CASO 9

El **CASO 9** nos lleva a uno de los problemas de rendimiento más "silenciosos" y frustrantes: la **incompatibilidad de tipos de datos** en las uniones (JOINs).

CASO 9: El Conflicto entre `INT` y `VARCHAR`

La Situación

Tenemos dos tablas que queremos unir:

1. `usuarios`: El campo `id` es un **ENTERO (INT)**.
2. `logs_actividad`: El campo `usuario_id` es una **CADENA (VARCHAR)**.

Aunque ambos campos tienen índices, la consulta de unión (`JOIN`) es extremadamente lenta.

¿Por qué los índices no funcionan aquí?

Cuando intentas unir un `INT` con un `VARCHAR`, MySQL se ve obligado a realizar una **conversión implícita de tipos** en cada fila para poder compararlos. Es decir, tiene que transformar el texto a número o viceversa sobre la marcha.

- **El desastre:** Al transformar el valor de la columna para compararlo, MySQL **inhabilita el índice**. Es como si tuvieras un diccionario ordenado alfabéticamente, pero te piden buscar palabras basándote en su número de vocales; el orden alfabético deja de servirte y tienes que leer todo el libro.

La Solución Técnica

La única solución real y definitiva es normalizar los tipos de datos para que coincidan. Como estamos hablando de IDs, lo correcto es que ambos sean números enteros:

```
ALTER TABLE logs_actividad MODIFY usuario_id INT NOT NULL;
```

¿Qué ganamos con esto?

1. **Comparación Nativa:** El procesador compara números de forma mucho más rápida que cadenas de texto.
2. **Activación de Índices:** Al no haber conversión, el motor puede usar el índice B-Tree de ambas tablas directamente.
3. **Ahorro de Espacio:** Un `INT` ocupa generalmente 4 bytes, mientras que un `VARCHAR(20)` puede ocupar mucho más, reduciendo el tamaño de la tabla y del índice.

   Consejo Pro: Siempre que diseñes una base de datos, asegúrate de que las llaves foráneas (`Foreign Keys`) tengan exactamente el mismo tipo y longitud que la llave primaria (`Primary Key`) a la que apuntan.

¿Pasamos al CASO 10? Veremos cómo optimizar un panel financiero donde el orden de las fechas es crítico para el usuario. ¿Seguimos?

CASO 10

El **CASO 10** nos sitúa en un entorno financiero donde el tiempo es dinero. Aquí el desafío no es solo encontrar los datos, sino entregarlos ya **ordenados**, evitando que el servidor tenga que hacer un esfuerzo extra después de filtrar.

CASO 10: Facturas, Filtros de Fecha y Ordenamiento

La Situación

La consulta es: `SELECT * FROM facturas WHERE empresa_id = 45 AND fecha BETWEEN '2023-01-01' AND '2023-01-31' ORDER BY fecha DESC;`

El motor de la base de datos tiene que realizar tres pasos:

1. **Filtrar** por la empresa específica (`empresa_id = 45`).
2. **Filtrar** por un rango de tiempo (`BETWEEN`).
3. **Ordenar** los resultados de la factura más reciente a la más antigua (`ORDER BY fecha DESC`).

El Problema: "Filesort"

Si solo tuviéramos un índice en `empresa_id`, MySQL encontraría las facturas de la empresa, pero luego tendría que cargarlas todas en la memoria RAM y ordenarlas por fecha. Si la empresa tiene miles de facturas, esto genera una operación lenta llamada **Filesort**.

La Solución: El Índice de Rango y Orden

Creamos un índice compuesto que "guíe" al motor en el orden exacto de la consulta:

```
CREATE INDEX idx_empresa_fecha ON facturas (empresa_id, fecha DESC);
```

¿Por qué funciona tan bien?

1. **Agrupación:** El índice coloca todas las facturas de la `empresa_id = 45` juntas.
2. **Pre-ordenamiento:** Dentro de ese grupo, las fechas ya están guardadas físicamente en orden descendente.
3. **Eficiencia:** El motor lee el índice, encuentra el bloque de la empresa 45, y simplemente "copia" los resultados en el orden en que aparecen. No necesita usar la CPU para ordenar nada; los datos ya están listos.

Un detalle técnico sobre el `DESC`

En versiones antiguas de MySQL, el `DESC` en los índices era ignorado (se leía hacia atrás). Sin embargo, desde **MySQL 8.0**, los índices descendentes son reales y permiten que el motor lea el índice de forma natural hacia adelante, lo cual es ligeramente más rápido para el hardware.

¿Pasamos al CASO 11? Este es muy curioso: ¿qué haces cuando la URL que quieres indexar es tan larga que el índice acaba ocupando más espacio que la propia tabla? ¡Vamos a ver la técnica del prefijo!

CASO 11

El **CASO 11** nos enfrenta al reto del almacenamiento y la eficiencia cuando manejamos datos masivos, como las URLs en una base de datos de ciberseguridad.

CASO 11: URLs Gigantes y el Índice de Prefijo

La Situación

Tienes una columna `url_detectada` de tipo `VARCHAR(2048)`. Indexar 2048 caracteres por cada fila es una receta para el desastre:

1. **Índice obeso:** El índice puede llegar a ser más grande que la tabla misma.
2. **Rendimiento pobre:** Al ser tan grande, el índice no cabe en la memoria RAM (`Buffer Pool`), lo que obliga a MySQL a leer constantemente del disco.

La Solución: Índice de Prefijo

En lugar de indexar toda la URL, indexamos solo los primeros caracteres. La lógica es que la mayoría de las URLs se diferencian en sus primeros 100 caracteres.

```
CREATE INDEX idx_url_corto ON tabla_url (url_detectada(100));
```

¿Cómo funciona?

MySQL solo guardará en el árbol del índice los primeros 100 caracteres de cada URL.

- **Si buscas una URL específica:** El motor busca en el índice el prefijo. Si encuentra una coincidencia, va a la tabla principal para verificar si el resto de la URL también coincide.
- **Ahorro masivo:** Has reducido el tamaño del índice en un **95%** aproximadamente (de 2048 a 100 caracteres).

El Reto: ¿Cuántos caracteres elegir?

No puedes elegir un número demasiado pequeño (como 5), porque muchas URLs empezarían igual (ej. `https://...`) y el índice perdería su utilidad (baja selectividad). Debes elegir un número donde la mayoría de tus datos sean únicos.

> **Dato curioso:** Esta técnica es la que permite indexar campos de tipo TEXT o BL0B, los cuales MySQL te **obliga** a indexar mediante prefijos porque no puede manejar su tamaño completo de forma eficiente en un B-Tree.

¿Terminamos con el CASO 12? Aprenderás a ser un administrador de bases de datos "Ninja": cómo desactivar un índice sin borrarlo para estar seguro de que no vas a romper nada en producción. ¿Listo?

CASO 12

El **CASO 12** es la red de seguridad de cualquier Administrador de Bases de Datos (DBA). En entornos de producción con millones de registros, borrar un índice es una operación de "no retorno" inmediato: si te equivocas, reconstruirlo puede tardar horas y mantener el servidor colapsado mientras tanto.

CASO 12: Índices Invisibles (El "Modo Avión" de los Índices)

La Situación

Tienes un índice llamado `idx_antiguo`. Todo indica que nadie lo usa, pero existe el miedo de que una consulta crítica que se ejecuta una vez al mes lo necesite. Si lo borras y esa consulta aparece, el sistema caerá.

El Concepto: Invisible Indexes

Introducidos en **MySQL 8.0**, los índices invisibles permiten que el índice siga existiendo y actualizándose, pero le prohíben al **Optimizador de Consultas** utilizarlo. Es como si el índice estuviera ahí, pero fuera invisible para los ojos del motor de búsqueda.

El Protocolo de Actuación Seguro

1. Hacerlo Invisible (La Prueba de Estrés)

En lugar de borrarlo, lo "apagamos":

```
ALTER TABLE nombre_tabla ALTER INDEX idx_antiguo INVISIBLE;
```

- **¿Qué sucede?** El índice se mantiene actualizado (si insertas datos, el índice se entera), pero ninguna consulta lo usará. Si el rendimiento del servidor no empeora tras unos días, confirmas que no era necesario.

2. La Marcha Atrás (En caso de emergencia)

Si de repente una consulta se vuelve lentísima tras el cambio, lo recuperas en **milisegundos**:

```
ALTER TABLE nombre_tabla ALTER INDEX idx_antiguo VISIBLE;
```

- A diferencia de un `CREATE INDEX` (que tendría que leer toda la tabla y tardar horas), esto es una operación instantánea en los metadatos.

3. El Borrado Definitivo

Tras un periodo de observación (por ejemplo, una semana), si todo va bien, puedes borrarlo físicamente para ahorrar espacio en disco:

```
DROP INDEX idx_antiguo ON nombre_tabla;
```

Resumen de Ventajas

- **Seguridad:** Eliminas el riesgo de "romper" la producción.
- **Mantenimiento:** El índice se mantiene al día mientras es invisible, por lo que al volverlo visible, los datos son 100% precisos de inmediato.

¡Hemos completado los 12 casos! 🚀

Has repasado desde la indexación simple hasta estrategias avanzadas de visibilidad y cardinalidad. ¿Te gustaría que hagamos un **resumen rápido de las 3 reglas de oro** para indexar o prefieres que probemos a crear un ejercicio práctico con una tabla inventada?