Cómo Crear Índices en MySQL

La creación de índices es un paso crucial para optimizar el rendimiento de las bases de datos. A continuación, se detalla la sintaxis SQL para los tipos de índices más comunes:

- **Índice Normal (B-Tree)**

El tipo más común. Permite valores duplicados y es ideal para búsquedas generales.

CREATE INDEX nombre_del_indice ON nombre_tabla (columna);

- **Índice Único (UNIQUE)**

Garantiza que todos los valores en la columna sean distintos, reforzando la integridad de los datos.

CREATE UNIQUE INDEX nombre_del_indice ON nombre_tabla (columna);

- **Índice Compuesto**

Creado sobre múltiples columnas, optimiza consultas que filtran por varias de ellas a la vez.

CREATE INDEX nombre_del_indice ON nombre_tabla (columna1, columna2);

- **Índice de Texto Completo (FULLTEXT)**

Especializado en búsquedas de texto natural en columnas extensas (CHAR, VARCHAR, TEXT).

CREATE FULLTEXT INDEX nombre_del_indice ON nombre_tabla (columna);

Reglas Clave para la Optimización de Índices

- **Regla del Prefijo de Izquierda**

Al usar índices compuestos (ej. (columna_A, columna_B, columna_C)), MySQL solo puede utilizarlos eficientemente si la consulta empieza a filtrar por la columna más a la izquierda y avanza hacia la derecha.

Un índice en (Apellido, Nombre, Ciudad) es útil para búsquedas por:
• Apellido
• Apellido y Nombre
• Apellido, Nombre y Ciudad.

No será usado si buscas solo por Nombre o Ciudad.

- **Prefijos de Columna**

Para columnas con textos largos (VARCHAR, TEXT), puedes indexar solo los primeros "N" caracteres para ahorrar espacio y acelerar la gestión del índice.

CREATE INDEX idx_descripcion ON productos (descripcion(10));

- **Ventaja:** Índice más pequeño y actualizaciones más rápidas.
- **Desventaja:** Menos efectivo si muchos registros comparten el mismo prefijo (baja cardinalidad), ya que MySQL aún necesitará escanear más filas.

Creación de Índices Durante la Definición de la Tabla

Es posible incorporar la definición de índices directamente en la sentencia CREATE TABLE, lo que facilita la configuración inicial de la tabla y sus optimizaciones desde el primer momento.

```
CREATE TABLE usuarios (
  id INT NOT NULL,
  email VARCHAR(100),
  nombre VARCHAR(100),
  PRIMARY KEY (id),
  UNIQUE INDEX idx_email (email),
  INDEX idx_nombre (nombre)
);
```

En este ejemplo, se define un índice primario (PRIMARY KEY) para la columna id, un índice único (UNIQUE INDEX) llamado idx_email para asegurar la unicidad del email, y un índice normal (INDEX) llamado idx_nombre para optimizar búsquedas por nombre.

Consultar Índices Existentes en MySQL

Puedes obtener una lista detallada de todos los índices definidos en una tabla específica utilizando el siguiente comando SQL:

SHOW INDEX FROM nombre_tabla;

Detalles Clave: Este comando te proporcionará una tabla con información valiosa, como el nombre del índice, si es primario o único, las columnas que lo componen, y la cardinalidad, que indica la unicidad de los valores en la columna indexada.

mysql> show index from pedidos;

+---------+------------+--------------------+--------------+------------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |

+---------+------------+--------------------+--------------+------------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+

| pedidos | 0 | PRIMARY | 1 | id_pedido | A | 49855 | NULL | NULL | | BTREE | | | YES | NULL |

| pedidos | 1 | idx_estado | 1 | estado_envio | A | 3 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_cliente | 1 | cliente_email | A | 1008 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_categoria_estado | 1 | categoria_producto | A | 4 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_categoria_estado | 2 | estado_envio | A | 12 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_fecha | 1 | fecha_pedido | A | 361 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_estado_ingresos | 1 | estado_envio | A | 3 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_estado_ingresos | 2 | cantidad | A | 30 | NULL | NULL | YES | BTREE | | | YES | NULL |

| pedidos | 1 | idx_estado_ingresos | 3 | precio_unitario | A | 42311 | NULL | NULL | YES | BTREE | | | YES | NULL |

+---------+------------+---------------------+-------------+------------------+-----------+-------------+----------+--------+------+--------

1. Columnas de Identificación

- **Table:** El nombre de la tabla (pedidos).
- **Non_unique:** * 0: El índice **no permite duplicados** (como la Primary Key).
    - 1: El índice **permite duplicados** (índices normales).
- **Key_name:** El nombre del índice. Fíjate que idx_estado_ingresos aparece 3 veces porque es un **índice compuesto** de tres columnas.
- **Seq_in_index:** El orden de la columna dentro del índice.
    - *Ejemplo:* En idx_estado_ingresos, estado_envio es la posición 1, cantidad la 2 y precio_unitario la 3.

2. Columnas de Estructura y Contenido

- **Column_name:** El nombre de la columna física de la tabla.
- **Collation:** Cómo se ordena. A significa *Ascendente*. Es lo más común.
- **Cardinality: ¡Esta es clave!** Es una estimación del número de valores únicos en el índice.
    - En PRIMARY, la cardinalidad es 49855 (casi cada fila es única).

    - En idx_estado, la cardinalidad es solo 3. Esto significa que solo hay 3 estados posibles (ej: 'Pendiente', 'Enviado', 'Cancelado'). **Es una cardinalidad baja.**

- **Index_type:** El método de almacenamiento. Casi siempre verás BTREE, que es la estructura de árbol balanceado que usa MySQL por defecto.

3. Columnas de Estado

- **Null:** YES si la columna puede contener valores nulos.
- **Visible:** YES o NO. Indica si el optimizador de consultas puede usar el índice (visto en el **Caso 12**).

Análisis de "Detectives" de tu tabla

Si miramos tu captura como expertos, podemos notar varias cosas interesantes:

1. **Redundancia detectada:** Tienes un índice llamado idx_estado (solo columna estado_envio) y otro índice compuesto idx_estado_ingresos que **empieza** por estado_envio.

Conclusión: El índice idx_estado sobra (Redundancia, como en el **Caso 6**), ya que el compuesto puede hacer su trabajo.

2. **Alta vs Baja Cardinalidad:** * idx_fecha tiene 361. Es moderada, buena para filtrar por días.
    - idx_estado_ingresos termina con precio_unitario, que tiene una cardinalidad de 42311. Esto lo hace un índice **muy potente** para búsquedas exactas combinadas.

¿Qué significan los resultados de idx_categoria_estado?

Vemos que tiene Seq_in_index 1 y 2.

- Si buscas solo por categoria_producto, el índice **funciona**.
- Si buscas por categoria_producto Y estado_envio, el índice **vuela**.
- Si buscas solo por estado_envio, el índice **no se usa** (Regla del prefijo).

¿Te gustaría que analizáramos si alguno de estos índices está penalizando tus INSERTs por ser demasiados, o prefieres ver cómo borrar el que hemos detectado que sobra?

Eliminar Índices Existentes en MySQL

- **Método 1: Usando DROP INDEX**

Esta sentencia es directa y permite eliminar un índice de una tabla específica. Es útil cuando se desea eliminar un índice sin alterar explícitamente la definición de la tabla.

DROP INDEX nombre_del_indice ON nombre_tabla;

- **Método 2: Usando ALTER TABLE**

La sentencia ALTER TABLE se utiliza para modificar la estructura de una tabla, y esto incluye la adición o eliminación de índices. Es una alternativa a DROP INDEX, con el mismo efecto.

ALTER TABLE nombre_tabla DROP INDEX nombre_del_indice;

Siempre es recomendable evaluar el impacto de la eliminación de un índice antes de ejecutar la operación en un entorno de producción, ya que puede afectar el rendimiento de las consultas.

EXPLAIN ANALYZE: Planificación y Ejecución en Tiempo Real

Mientras que EXPLAIN tradicionalmente ofrece una estimación del plan de ejecución de una consulta, EXPLAIN ANALYZE va un paso más allá al ejecutar la consulta y proporcionar el plan de ejecución real junto con estadísticas de tiempo de ejecución.

EXPLAIN ANALYZE SELECT columna1, columna2 FROM nombre_tabla WHERE columna3 = 'valor' ORDER BY columna1;

El resultado incluye no solo la información de EXPLAIN (cómo planea ejecutar la consulta) sino también detalles sobre los tiempos de ejecución y las filas procesadas en cada etapa, lo que permite una optimización mucho más informada.

Ejecución Real

Muestra el tiempo exacto de ejecución y las filas procesadas en cada paso de la consulta, revelando si las estimaciones iniciales eran correctas.

Plan Detallado

Proporciona un desglose exhaustivo de cómo SGBD procesa la consulta, incluyendo uniones (JOINs), filtros y el uso de índices, con métricas de costo y tiempo reales.

Identificación de Ineficiencias

Facilita la detección de operaciones costosas o ineficientes que no se habrían identificado solo con el plan estimado, guiando hacia mejoras específicas en los índices o la consulta.

Análisis Detallado con EXPLAIN ANALYZE

Al interpretar los resultados de EXPLAIN ANALYZE, es crucial fijarse en métricas clave que revelan la eficiencia real de la consulta.

Tiempos Reales vs. Estimados

Compara actual time con los cost estimados. Discrepancias grandes indican que el optimizador pudo haber tomado decisiones subóptimas.

Filas y Bucles Procesados

Observa rows y loops para cada etapa. Un alto número de filas procesadas o bucles repetitivos sugiere posibles cuellos de botella.

Identificación de Operaciones

Busca operaciones costosas como Seq Scan (escaneo completo de tabla) o tipos de uniones ineficientes (Hash Join sin índices adecuados).

Ejemplo EXPLAIN

EXPLAIN - Coste estimado

- Cost: es una **unidad de medida arbitraria** (no son segundos ni milisegundos) que representa el esfuerzo que le cuesta a la CPU y al disco realizar esa operación.
- Rows= filas estimadas que el optimizador crea que va a procesar

EXPLAIN FORMAT=TREE

```
SELECT first_name, last_name, SUM(amount) AS total
FROM staff INNER JOIN payment
  ON staff.staff_id = payment.staff_id
    AND
    payment_date LIKE '2005-08%'
GROUP BY first_name, last_name;
```

```
-> Table scan on <temporary>
   -> Aggregate using temporary table
      -> Nested loop inner join  (cost=1757.30 rows=1787)
         -> Table scan on staff  (cost=3.20 rows=2)
         -> Filter: (payment.payment_date like '2005-08%')  (cost=117.43 rows=894)
            -> Index lookup on payment using idx_fk_staff_id (staff_id=staff.staff_id)
(cost=117.43 rows=8043)
```

Operación	Rendimiento	Descripción	Recomendación
Index Lookup	⚡ Excelente	Busca datos usando un índice específico.	Minimizar la lectura de disco.
Table Scan	🐢 Lento	Lee todas las filas de la tabla de arriba a abajo.	Evitarlo en tablas grandes usando índices.
Nested Loop	⚖️ Variable	Cruza filas de dos tablas mediante bucles anidados.	Eficiencia mediante índices en tabla interna.
Filter	🔍 Necesario	Evalúa condiciones (WHERE) sobre los datos.	Filtrar lo más pronto posible.
Aggregate	📊 Medio	Resume múltiples filas en un valor (SUM, COUNT).	Agrupar y procesar datos.
Using filesort	⚠️ Pesado	El motor debe ordenar los datos tras obtenerlos.	Evitarlo indexando las columnas de ORDER BY.
Index Scan	📈 Medio	Lee todo el índice (más ligero que toda la tabla).	Mejorar un Table Scan si no hay filtros exactos.

Ejemplo EXPLAIN ANALYZE

- Se usa para entender el comportamiento real de una consulta
    - actual time: Se mide en **milisegundos (ms)** y suele mostrar dos valores: actual time=inicio..fin. Por cada loop
    - rows: filas reales que se procesan.
    - Loops: **cuántas veces se repitió ese paso**

EXPLAIN ANALYZE FORMAT=TREE

```
SELECT first_name, last_name, SUM(amount) AS total
FROM staff INNER JOIN payment
  ON staff.staff_id = payment.staff_id
    AND
    payment_date LIKE '2005-08%'
GROUP BY first_name, last_name;
```

-> Table scan on <temporary> (actual time=0.001..0.001 rows=2 loops=1)

-> Aggregate using temporary table (actual time=58.104..58.104 rows=2 loops=1)

-> Nested loop inner join (cost=1757.30 rows=1787) (actual time=0.816..46.135 rows=5687 loops=1)

-> Table scan on staff (cost=3.20 rows=2) (actual time=0.047..0.051 rows=2 loops=1)

-> Filter: (payment.payment_date like '2005-08%') (cost=117.43 rows=894) (actual time=0.464..22.767 rows=2844 loops=2)

-> Index lookup on payment using idx_fk_staff_id (staff_id=staff.staff_id) (cost=117.43 rows=8043) (actual time=0.450..19.988 rows=8024 loops=2)

Optimización de Tipos de Datos

Para maximizar el rendimiento y minimizar el consumo de recursos, es fundamental seleccionar los tipos de datos más adecuados. Esto afecta directamente el almacenamiento, la memoria RAM (Buffer Pool) y la velocidad de procesamiento de la CPU.

Integridad Numérica

Asegúrate de usar el tipo entero más pequeño posible (TINYINT, SMALLINT, INT, BIGINT) que pueda contener los valores esperados. Por ejemplo, TINYINT para edades y INT para IDs estándar.

Números Sin Signo (UNSIGNED)

Para columnas que nunca contendrán valores negativos (como claves primarias o cantidades), utiliza la opción UNSIGNED. Esto duplica el rango positivo del tipo de dato, permitiendo almacenar valores más grandes sin aumentar el espacio.

Cadenas de Texto: VARCHAR vs CHAR

- Usa CHAR para cadenas de longitud fija y corta (ej. códigos ISO de país).
- Limita VARCHAR a un tamaño realista en lugar de usar VARCHAR(255) por defecto, ya que cada byte cuenta en la gestión de memoria.

Valores Fijos con ENUM

Para columnas con un conjunto predefinido de valores (ej. 'activo', 'inactivo'), ENUM es ideal. MySQL lo almacena internamente como un número, ahorrando significativamente espacio.

Fechas y Horas

Emplea DATE, DATETIME o TIMESTAMP en lugar de VARCHAR para las columnas de fecha/hora. Esto permite la aritmética de fechas, una ordenación eficiente y optimizaciones de búsqueda.

Precisión Decimal con DECIMAL

Para datos financieros o cualquier valor que requiera precisión exacta, usa DECIMAL(M,D) en lugar de FLOAT o DOUBLE para evitar errores de redondeo.

DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.

Tipos de datos en SQL.

https://www.w3schools.com/sql/sql_datatypes.asp

Data type	**Description**
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'

Ejemplo de UNSIGNED

```
CREATE TABLE pedidos (
   -- El ID nunca será negativo, usamos UNSIGNED para duplicar el rango
   id_pedido INT UNSIGNED AUTO_INCREMENT,

   -- El ID del cliente tampoco es negativo
   id_cliente INT UNSIGNED NOT NULL,

   -- No existen "cantidades negativas" de productos en un pedido
   cantidad TINYINT UNSIGNED NOT NULL,

   -- El precio siempre es positivo
   total_pago DECIMAL(10,2) UNSIGNED NOT NULL,

   -- Definimos la llave primaria
   PRIMARY KEY (id_pedido)
) ENGINE=InnoDB;
```

\\\\\\\\\\\\

```
CREATE TABLE products(
product_id INT AUTO_INCREMENT,
product_item varchar(255) NOT NULL,
category varchar(255) NOT NULL,
total_amount int UNSIGNED,
PRIMARY KEY (product_id)
);
```

Diseño y Normalización (Lógica de Estructura)

El diseño de la estructura de la base de datos es fundamental para garantizar la eliminación de la redundancia y asegurar la integridad de los datos, elementos clave para un rendimiento óptimo.

Eliminación de Redundancia

¿Se ha aplicado al menos la 3ª Forma Normal (3FN)? Evita repetir información, como nombres de clientes o categorías, en cada fila de ventas para optimizar el almacenamiento y la coherencia.

Claves Primarias (PK)

Asegúrate de que cada tabla tenga una clave primaria (PK) clara y única, preferiblemente numérica y auto-incremental, para una identificación eficiente de los registros.

Claves Ajenas (FK)

Define explícitamente las claves ajenas (FK) para permitir que el motor de la base de datos optimice las operaciones JOIN y mantenga la integridad referencial entre tablas.

Atributos Atómicos

Verifica que cada columna contenga un solo dato atómico. Por ejemplo, evita guardar "Calle, Número, Piso" en una única columna para facilitar la búsqueda y manipulación de datos.

Índices

Optimización del uso de índices