Vistas e Indices en Bases de Datos MySQL

Vistas en Bases de Datos (MySQL)

Introducción

Las vistas en MySQL son una herramienta poderosa que permite a los usuarios interactuar con los datos de una manera más sencilla y segura. Una vista es una tabla virtual cuyo contenido se define mediante una consulta. A diferencia de una tabla, una vista no almacena datos físicamente, sino que genera los datos dinámicamente cada vez que se consulta.

¿Qué es una Vista?

Una vista es una consulta almacenada que se puede tratar como una tabla. Las vistas se utilizan para simplificar la complejidad de las consultas, proporcionar seguridad y presentar los datos de una manera específica.

Las vistas tienen varios usos, entre ellos:

Simplicidad: Simplificar consultas complejas al encapsularlas en una vista que se puede consultar fácilmente.

Seguridad: Restringir el acceso a datos sensibles proporcionando solo la información necesaria a través de la vista.

Mantenimiento: Facilitar el mantenimiento de consultas complejas al centralizarlas en una vista.

Consistencia: Asegurar que las consultas reutilizadas sean consistentes en toda la aplicación.

Características de las Vistas

1. **Es una Tabla Virtual**: Una vista no almacena datos físicamente, sino que muestra los datos de las tablas subyacentes.
2. **Actualización Dinámica**: Los datos en una vista siempre están actualizados, ya que se generan en tiempo real a partir de las tablas base.
3. **Seguridad**: Las vistas pueden restringir el acceso a datos sensibles al mostrar solo las columnas y filas necesarias.
4. **Simplificación**: Facilitan la escritura de consultas complejas al encapsular la lógica en una sola vista.
5. **Compatibilidad**: Proporciona una interfaz consistente incluso si cambia el esquema de las tablas subyacentes.
6. **Rendimiento**: Aunque MySQL no soporta vistas indexadas, las vistas pueden mejorar la organización y legibilidad de las consultas. También es importante tener en cuenta alternativas como las tablas temporales y la optimización de consultas, en casos de vistas muy complejas.

Tipos de Vistas

1. **Vistas Simples**: Basadas en una sola tabla. Pueden ser "actualizables", es decir usar update, delete e insert a través de ellas, pero solo si están basadas en 1 sola tabla y no usan: join/unión/disctint/subquerys/funciones de agregado.

2. **Vistas Complejas**: Basadas en múltiples tablas y pueden incluir funciones agregadas. No son "actualizables" (complejidad para asegurar consistencia e integridad).

3. **Vistas Materializadas**: MySQL no soporta vistas materializadas/persistentes nativamente, pero se pueden simular mediante tablas temporales y triggers.

Limitaciones de las Vistas

1. *No Almacenan Datos:* Las vistas no almacenan datos físicamente, lo que puede afectar el rendimiento en consultas muy complejas.
2. *Actualización:* No todas las vistas son "actualizables", especialmente las que incluyen funciones agregadas o combinaciones complejas.
3. *Rendimiento:* Consultas a vistas complejas pueden tener un rendimiento inferior en comparación con consultas directas a las tablas subyacentes, ya que la vista necesita ser recalculada cada vez que se accede. Es en estos (y otros casos), el **optimizador de consultas** puede que tenga alguna dificultad al momento de realizar su trabajo debido a que existen factores subyacentes a la vista, que afectan el rendimiento. Por ejemplo: complejidad de la consulta, índices, funciones agregadas, group by y tamaño de las tablas involucradas.
4. *Restricciones:* Algunas vistas no pueden contener ciertas cláusulas, como ORDER BY, a menos que se utilicen con LIMIT para garantizar el orden de los resultados.
5. *Seguridad y Permisos:* Las vistas pueden restringir el acceso a ciertas columnas de las tablas subyacentes, pero no pueden ocultar completamente los datos subyacentes si un usuario tiene permisos suficientes para acceder a las tablas directamente.
6. *Índices:* Las vistas no pueden tener índices propios (ya que no almacenan datos físicamente), lo que puede limitar el rendimiento en consultas que se beneficiarían del uso de índices.
7. *Dependencia de las Tablas Subyacentes:* Los cambios en las tablas subyacentes, como la modificación de la estructura o eliminación de columnas, pueden invalidar las vistas que dependen de esas tablas.
8. *Limitaciones en Consultas de Unión:* Vistas que usan UNION o UNION ALL pueden tener limitaciones en su capacidad para actualizar datos a través suyo.
9. *Limitaciones para depurar consultas c/vistas complejas:* Esto se debe a que la lógica de las vistas esta oculta en su definición y por ende esto dificulta el seguimiento de una consulta que utilice una vista.

Crear Vistas en MySQL

1- create view comprasXcliente as
select
    c.CLI_ID,
    c.APELLIDO,
    c.NOMBRES,
    v.FECHA_COMPRA
from
    clientes c,
    ventas v
where
    c.CLI_ID = v.CLI_ID

2- CREATE
    [OR REPLACE]
    VIEW nombre_vista [(columna1, columna2, ...)] //por defecto, los mismos nombres que las columnas de las tablas
    AS
    sentencia_select
    [WITH [CASCADED | LOCAL] CHECK OPTION];

CREATE OR REPLACE VIEW clientes_activos (id_cliente, nombre_completo, email) AS
SELECT cli_id, CONCAT(apellido, ', ', nombres), correo_electronico
FROM clientes
WHERE estado = 'activo';

With CHECK OPTION: se utiliza al crear o modificar una vista para controlar cómo se manejan las inserciones y actualizaciones de datos a través de esa vista. Su objetivo principal es asegurar que cualquier cambio realizado a través de la vista cumpla con las condiciones definidas en la consulta SELECT que define la vista.

LOCAL:

- Cuando se utiliza LOCAL, las comprobaciones de la cláusula WHERE (o cualquier otra condición de filtrado) se aplican solo a la vista específica en la que se define CHECK OPTION.
- Si la vista se basa en otra vista (una vista anidada), las comprobaciones de otras vistas anidadas no se tienen en cuenta.

CASCADED:

- Cuando se utiliza CASCADE, las comprobaciones se aplican no solo a la vista actual, sino también a todas las vistas subyacentes (vistas anidadas) que tengan CHECK OPTION definido.
- Esto asegura que cualquier cambio realizado a través de la vista cumpla con las condiciones de todas las vistas en la cadena de vistas anidadas.

Vamos con ejemplos…

```
CREATE OR REPLACE VIEW clientes_por_ciudad AS
SELECT CONCAT(nombre, ' ', apellido) AS nombre_completo, email
FROM Clientes
WHERE ciudad = 'Madrid'
WITH CHECK OPTION;
```

Aquí definimos una vista que establece como condición en el **where**, que la ciudad sea *Madrid*. La cláusula WITH CHECK OPTION (por defecto: LOCAL) asegura que cualquier inserción o actualización en la vista cumpla con la condición definida en el WHERE de la vista.

¿Que sucede en este caso?

```
INSERT INTO clientes_por_ciudad (nombre_completo, email) VALUES ('Pedro Sánchez', 'pedro.sanchez@email.com');
```

La instrucción falla, porque al no insertar la ciudad, no es posible verificar que la misma sea Madrid; por lo tanto, la inserción no cumple con la condición de la vista (establecida en el where).

Por otra parte, si la vista fuese la siguiente:

```
CREATE OR REPLACE VIEW clientes_por_ciudad AS
SELECT CONCAT(nombre, ' ', apellido) AS nombre_completo, email, ciudad
FROM Clientes
WHERE ciudad = 'Madrid'
WITH CHECK OPTION;
```

Podríamos ejecutar…

```
INSERT INTO clientes_por_ciudad (nombre_completo, email, ciudad)
VALUES ('Pedro Sánchez', 'pedro.sanchez@email.com', 'Madrid');
```

En este caso, la instrucción NO fallara, ya que podemos establecer la ciudad y la misma, al ser Madrid, no infringe la condición del **where** de la vista.

*¿Qué sucede en este **otro** caso?*

```
CREATE OR REPLACE VIEW clientes_madrid AS
SELECT nombre, apellido, ciudad
FROM Clientes
WHERE ciudad = 'Madrid'
WITH CHECK OPTION CASCADE;
```

```
CREATE OR REPLACE VIEW clientes_primera_letra_a AS
SELECT nombre, apellido, ciudad
FROM clientes_madrid
WHERE nombre LIKE 'A%'
WITH CHECK OPTION CASCADE;
```

En este caso Con CASCADE, cualquier operación sobre la vista derivada (clientes_primera_letra_a) no solo verifica las condiciones de esa vista, sino también las condiciones de la vista base (clientes_madrid).

Entonces la siguiente instrucción fallara:

```
INSERT INTO clientes_primera_letra_a (nombre, apellido, ciudad)
VALUES ('Ana', 'Lopez', 'Barcelona');
```

Si bien el nombre comienza con 'A' (condición de la vista derivada); no cumple con que la ciudad sea 'Madrid'. No así con el siguiente ejemplo, en el cual se cumple con ambas condiciones y por ende la instrucción NO fallara.

```
INSERT INTO clientes_primera_letra_a (nombre, apellido, ciudad)
VALUES ('Ana', 'Lopez', 'Madrid');
```

Consultar la vista (como cualquier tabla)

```
select
        *
from
        comprasxcliente
```

Ejemplos

Vista "Simple"

1-
```
CREATE VIEW ResumenDeVentas AS
SELECT
   IDOrdenDeVenta AS SalesOrderID,
   FechaDeOrden AS OrderDate,
   TotalDebido AS TotalDue
FROM
   EncabezadoDeOrdenDeVenta;
```

2-
```
CREATE TABLE empleados (
   id INT AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(50),
   departamento VARCHAR(50),
   salario DECIMAL(10, 2)
);

CREATE VIEW vista_empleados AS
SELECT nombre, departamento
FROM empleados;

-- Actualizar el departamento de un empleado
UPDATE vista_empleados
SET departamento = 'Recursos Humanos'
WHERE nombre = 'Juan';
```

Vista "Compleja"

1-
```
CREATE VIEW VentasDeProductos AS
SELECT
   p.Nombre,
   SUM(s.CantidadOrdenada) AS TotalVendido
FROM
   Producto AS p
JOIN
   DetalleDeOrdenDeVenta AS s ON p.IDProducto = s.IDProducto
GROUP BY
   p.Nombre;
```

2-
```
CREATE TABLE productos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50),
    precio DECIMAL(10, 2)
);

CREATE TABLE ventas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    producto_id INT,
    cantidad INT,
    FOREIGN KEY (producto_id) REFERENCES productos(id)
);

CREATE VIEW vista_productos_ventas AS
SELECT p.nombre, p.precio, SUM(v.cantidad) AS total_ventas
FROM productos p
JOIN ventas v ON p.id = v.producto_id
GROUP BY p.nombre, p.precio;

-- Intentar actualizar a través de la vista generaria un error
UPDATE vista_productos_ventas
SET precio = 25.00
WHERE nombre = 'Producto A';
```

Modificar una vista

```
ALTER VIEW vista_empleados AS
SELECT nombre, departamento, salario
FROM empleados;
```

Eliminar una vista

```
DROP VIEW vista_empleados;
```

Índices en MySQL

Introducción

En una base de datos, los índices son estructuras que mejoran la velocidad de recuperación de datos en las consultas. Cuando trabajamos con grandes volúmenes de datos, la búsqueda de información puede ser muy lenta si no tenemos índices definidos. Un índice actúa como el índice de un libro: nos permite encontrar la información rápidamente sin tener que recorrer todo el contenido.

Concepto Básico

Un índice en MySQL es un objeto de base de datos que puede ser creado sobre una o más columnas de una tabla. El objetivo es permitir un acceso más rápido a los datos en las consultas que usan esas columnas.

¿Por qué usar Índices?

Imaginemos que tenemos una tabla de clientes con miles o millones de registros. Si deseamos encontrar rápidamente a un cliente por su apellido, MySQL tendríamos que revisar fila por fila para encontrar el valor deseado si no existe un índice. Este proceso es ineficiente. Sin embargo, al crear un índice en la columna "Apellido", MySQL puede localizar el dato de manera mucho más rápida, evitando leer todas las filas de la tabla.

Ventajas principales:

1. **Mejora en la velocidad de consulta:** Un índice bien diseñado puede hacer que las consultas sobre una gran cantidad de datos sean mucho más rápidas.
2. **Ordenamiento más eficiente:** Los índices facilitan el ordenamiento rápido de resultados.
3. **Optimización de búsquedas:** Las búsquedas que usan las columnas indexadas son más eficientes.

Desventajas o Consideraciones:

1. **Uso de espacio:** Los índices ocupan espacio adicional en la base de datos.
2. **Impacto en inserciones, actualizaciones y eliminaciones:** Cada vez que los datos de una tabla cambian, el índice también debe actualizarse, lo que puede ralentizar las operaciones de modificación de datos.

Tipos de Índices en MySQL

Índice Clustered (Agrupado)

Un índice clustered o índice agrupado determina el orden físico de los datos en la tabla. Solo puede haber un índice agrupado por tabla, ya que solo puede haber un orden físico en los datos.

- La tabla misma se organiza de acuerdo con el índice agrupado.
- Cuando realizas una consulta que utiliza este índice, MySQL puede encontrar los datos mucho más rápido, ya que están ordenados físicamente.

Ejemplo de Creación de un Índice Agrupado:

```
CREATE CLUSTERED INDEX IX_Clientes_Apellido ON Clientes (Apellido);
```

Índice Non-Clustered (No Agrupado)

Un índice non-clustered o índice no agrupado no afecta el orden físico de los datos en la tabla, sino que crea una estructura separada que apunta a las filas correspondientes en la tabla base.

- Podemos tener varios índices no agrupados en una tabla.
- Este índice mantiene una referencia a la ubicación física de los datos.

Ejemplo de Creación de un Índice No Agrupado:

```
CREATE INDEX IX_Clientes_Email ON Clientes (Email);
```

Índices Compuestos

Un índice compuesto es un índice que se crea sobre más de una columna de una tabla. Es útil cuando las consultas filtran u ordenan datos en más de una columna.

Ejemplo de Creación de un Índice Compuesto:

```
CREATE INDEX IX_Clientes_Apellido_Nombre ON Clientes (Apellido, Nombre);
```

Cómo Utilizan los Índices las Consultas

Consulta sin Índices: Imaginemos una tabla de clientes sin ningún índice. Si realizamos una consulta para buscar clientes con un apellido específico, MySQL debe revisar todas las filas de la tabla para encontrar los resultados:

```
SELECT * FROM Clientes WHERE Apellido = 'González';
```

Consulta con Índices: Al agregar un índice en la columna Apellido, MySQL puede usar el índice para buscar rápidamente la fila o filas que contienen "González":

```
SELECT * FROM Clientes WHERE Apellido = 'González';
```

Mantenimiento de Índices

Debido a que los índices afectan el rendimiento de las consultas, pero también impactan las operaciones de modificación de datos, es importante realizar mantenimiento regular de los índices. MySQL ofrece herramientas como OPTIMIZE TABLE para optimizar los índices.

Reorganizar y Reconstruir Índices:

```
OPTIMIZE TABLE Clientes;
```

Buenas Prácticas

- No crear índices en todas las columnas. Esto aumenta el uso de espacio y puede afectar el rendimiento de las operaciones de inserción, actualización y eliminación.
- Evaluar la frecuencia de uso de columnas en las consultas antes de crear un índice.
- Realizar un mantenimiento regular de los índices, especialmente en tablas que reciben muchas inserciones y actualizaciones.
- Utilizar índices compuestos en columnas que frecuentemente se consultan juntas.

Modificar un Índice

Para modificar un índice, generalmente tendríamos que eliminar el índice existente y luego crear uno nuevo con las modificaciones deseadas. MySQL no permite modificar directamente un índice existente. Se haría asi:

1- DROP INDEX nombre_del_indice ON nombre_de_la_tabla;

Ejemplo: DROP INDEX IX_Clientes_Apellido ON Clientes;

2- CREATE INDEX nuevo_nombre_del_indice ON nombre_de_la_tabla (nueva_columna1, nueva_columna2);

Ejemplo: CREATE INDEX IX_Clientes_Nombre_Apellido ON Clientes (Nombre, Apellido);

Eliminar un Índice

Para eliminar un índice en MySQL, podemos usar el comando DROP INDEX. Aquí la sintaxis y un ejemplo:

DROP INDEX nombre_del_indice ON nombre_de_la_tabla;

Ejemplo: DROP INDEX IX_Clientes_Email ON Clientes;

Índices B-Tree (B: binario?)

Los índices B-Tree (árboles balanceados) son versátiles y efectivos para una amplia gama de operaciones de búsqueda y clasificación.

¿Cómo funcionan?
Un índice B-Tree organiza los datos de manera jerárquica en forma de un **árbol balanceado** (*recordemos el concepto*). Esto significa que todos los caminos desde la raíz hasta las hojas del árbol tienen la misma longitud. El árbol se ajusta automáticamente cuando se insertan, actualizan o eliminan datos para mantener el balance.
El B-Tree permite búsquedas rápidas porque divide el espacio de búsqueda en múltiples ramas, descartando partes grandes de los datos con cada nivel del árbol que se examina (*¿recuerdan "búsqueda binaria"?*). Este tipo de índice es *particularmente útil para buscar valores específicos, rangos de valores y para operaciones como ORDER BY*.

¿a que se refiere?

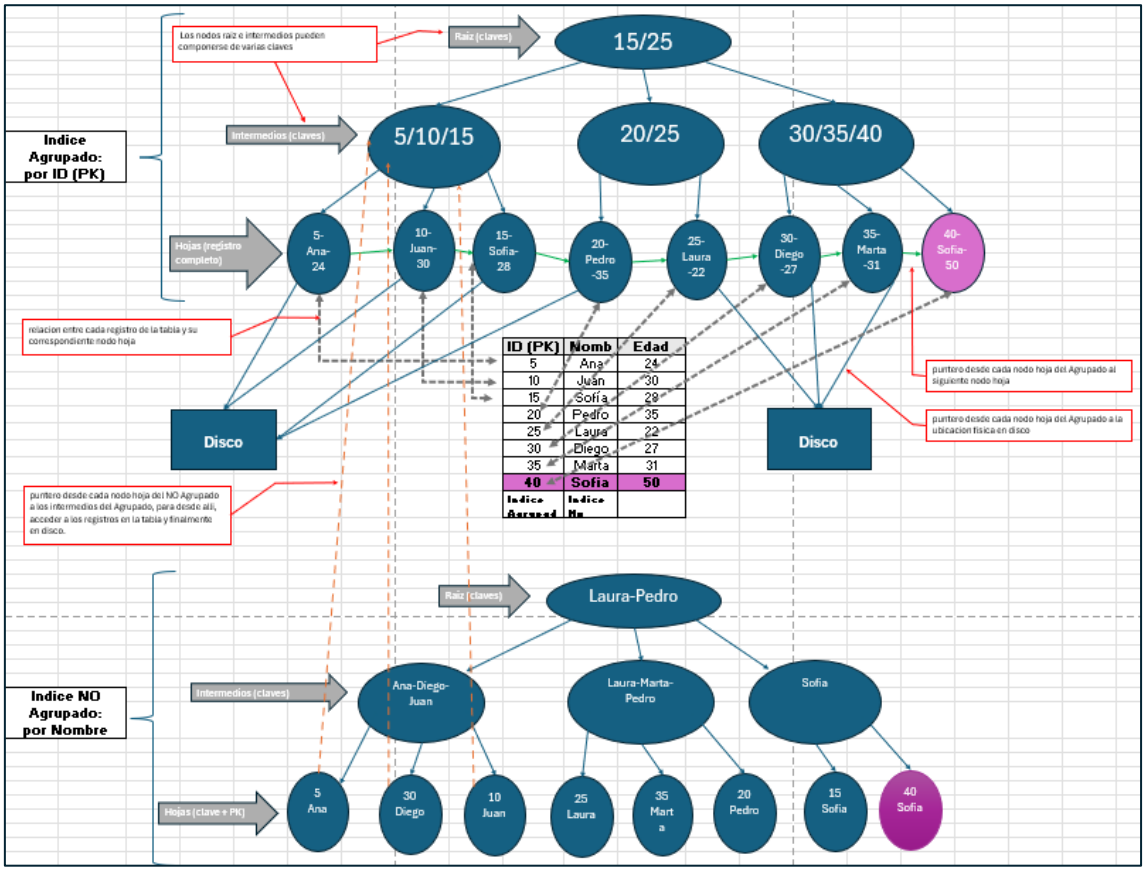
Características:
- Soportan búsquedas exactas (=) y por rango (<, <=, >, >=, BETWEEN).
- Están organizados de manera que las operaciones de lectura sean eficientes.
- Sirven como base para muchos índices (la mayoría) en MySQL.

Limitaciones:
- Aunque son rápidos en consultas de lectura, mantener un B-Tree puede ser **costoso** en términos de rendimiento durante las **operaciones de inserción, eliminación o actualización**, ya que el árbol debe reequilibrarse.

Preguntas hasta aquí:

1. ¿Porque decimos que estos índices (**B-Tree**) sirven de base para muchos otros índices? ¿a que "otros" índices nos referimos?
2. ¿Qué sucede en nuestras tablas al declarar una **pk**? ¿de quien depende lo que suceda en el caso anterior?
3. ¿Qué sucede si no declaramos una pk? (**unique/rowid**).
4. ¿Qué diferencia hay para el motor, si tiene que leer datos basándose en un índice clustered o en uno que no lo es? (acceso a disco)
5. ¿Por qué un índice no-agrupado, pasa/apunta al índice agrupado de la tabla? (lo que hace que el agrupado sea una especie de "**intermediario**")?
6. ¿Qué impacto tendría en las relaciones entre tablas, que mis tablas tengan como índice agrupado, la columna **rowid**?
7. ¿Cómo afectan los índices a las **FK**?
8. ¿Cómo impacta en un índice agrupado, el definir una **pk compuesta**?

Resumen ilustrativo y comparativo de índices Agrupados y No-Agrupados



Índice Agrupado:

- **Nodos Intermedios (Raíz y Nivel 1):**
  - Contienen claves primarias (IDs) que dividen el rango de IDs en ramas.
  - Tienen punteros a los nodos hijos (otros nodos intermedios u hojas).
  - No tienen punteros que los relacionen entre sí al mismo nivel.
- **Nodos Hoja:**
  - Contienen los datos completos de las filas (todas las columnas).
  - Están enlazados secuencialmente mediante punteros para búsquedas de rango.
  - Apuntan directamente a las ubicaciones físicas de los registros en el disco (o simulan apuntar a bloques de disco).

Índice No Agrupado:

- **Nodos Intermedios (Raíz y Nivel 1):**
  - Contienen los valores de la columna indexada (nombres) que dividen el rango de valores en ramas.
  - Tienen punteros a los nodos hijos (otros nodos intermedios u hojas).
  - No tienen punteros que los relacionen entre sí al mismo nivel.
- **Nodos Hoja:**
  - Contienen pares de valores: la clave del índice (nombre) y la clave primaria (ID) correspondiente.
  - La clave primaria (ID) actúa como un puntero al registro correspondiente en el índice agrupado.
  - Tienen punteros a los nodos intermedios del índice agrupado que contienen las claves primarias correspondientes.

Comparación de Punteros:

- **Agrupado:**
  - Hojas: punteros a registros físicos.
  - Intermedios: punteros a nodos hijos.
- **No Agrupado:**
  - Hojas: punteros a claves primarias en el índice agrupado.
  - Intermedios: punteros a nodos hijos.

Comparación de Contenido de Nodos:

- **Agrupado:**
  - Hojas: datos completos de las filas.
  - Intermedios: claves primarias (IDs).
- **No Agrupado:**
  - Hojas: clave del índice (nombre) y clave primaria (ID).
  - Intermedios: claves del índice (nombres).

Índices Full-Text (Texto Completo)

Los índices de texto completo están diseñados específicamente para búsquedas en grandes bloques de texto. Son útiles en escenarios como *motores de búsqueda para sitios web* o aplicaciones donde necesitamos buscar términos específicos en campos de texto largos.

¿Cómo funcionan?

En lugar de buscar coincidencias exactas, este tipo de índice "tokeniza" el texto en palabras y las almacena en una **estructura** que permite búsquedas basadas en **relevancia**. Esto permite realizar búsquedas avanzadas como encontrar textos que "contengan las palabras clave" o que "tengan una relación semántica".

"**tokeniza**": proceso de dividir o fragmentar un bloque de texto completo en **unidades más pequeñas llamadas "tokens"**, donde cada token generalmente representa una palabra o término relevante dentro del texto.

¿Qué significa "token"?

Un "token" es una pieza elemental de información. En el caso de un índice Full-Text, los tokens suelen ser palabras individuales que se extraen del texto completo.

Aplicación/uso:

Estos índices usan la cláusula **MATCH / AGAINST** para realizar búsquedas:

- **MATCH** especifica las columnas donde se realiza la búsqueda.
- **AGAINST** contiene las palabras clave que se están buscando.

Ejemplo: en esta tabla…

```
CREATE TABLE articulos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    titulo VARCHAR(255),
    contenido TEXT
);
```

1. **CREATE FULLTEXT INDEX** idx_contenido ON articulos (contenido);

2. SELECT * FROM articulos
   WHERE **MATCH**(contenido) **AGAINST** ('tecnología');

Ventajas:

- Realizan búsquedas rápidas y precisas en textos largos.
- Soportan búsquedas semánticas, con opciones para clasificar resultados por relevancia.

Limitaciones:

- Solo funcionan con tablas de tipo InnoDB o MyISAM (dependiendo de la versión de MySQL).
- No son útiles para datos pequeños o consultas con igualdad estricta.

*Ahora bien, ¿Cómo es el proceso de "**tokenizacion**"?*

1. Extracción de palabras:
- Se toma el contenido de una columna (por ejemplo, una descripción o un artículo de texto largo).
- El texto se analiza y se separa en palabras individuales (**tokens**), omitiendo espacios, puntuación u otros caracteres innecesarios.

2. Eliminación de palabras irrelevantes (stop words):
- Palabras comunes como "y", "de", "el", "la", etc., que no aportan significado al contexto de búsqueda, se eliminan.
- Así se reduce el "*ruido*" y mejora la eficiencia de las búsquedas.

3. Indexación de los tokens:
- Cada **token (palabra)** se indexa y se almacena junto con información sobre su posición en el texto, lo que permite realizar búsquedas eficientes.
    - NOTA 1: los token se almacenan en una estructura que forma parte de la misma tabla. Asocia cada palabra con las filas donde aparece.
    - NOTA 2: se utiliza un "mapa de relaciones" que contiene la info sobre los tokens y los id's de filas donde se encuentran esos tokens.

Ejemplo 1:

- Supongamos que tenemos una tabla con un campo TEXT que en algún registro almacena algo como: "*Los avances en la inteligencia artificial son sorprendentes.*"
- Luego de aplicar el proceso (**tokenizacion**), obtendríamos lo siguiente:
    - Los **tokens** serían: ["avances", "inteligencia", "artificial", "sorprendentes"]. Se almacenan en 1 sola estructura global para todos los registros de la tabla.
    - Palabras **irrelevantes** como "Los", "en", "la", "son" serían descartadas (stop words).
    - Los tokens se almacenan en el índice **Full-Text** para facilitar búsquedas como:
    -
        - SELECT * FROM articulos
        - WHERE MATCH(contenido) AGAINST ('inteligencia artificial');

Ejemplo 2: Tenemos una tabla con un índice Full-Text sobre una columna de texto. Si en la columna aparece la palabra "tecnología" en las filas 1, 3 y 7, el índice global incluiría algo como lo siguiente:

Token: "tecnología" → *almacenado en la "estructura global de token's".*
Mapeo: [fila 1, fila 3, fila 7] → *almacenado en el "mapa de relaciones".*

Optimización de búsquedas Full-Text

1. **Uso eficiente de índices Full-Text:** Tengamos en cuenta que las columnas donde definimos índices Full-Text sean aquellas con grandes volúmenes de texto (como contenido o descripción) y donde se realicen búsquedas frecuentes.
2. **Evitar palabras comunes (stop words):** MySQL ignora palabras comunes como "el", "la", "y", "de", etc. Si necesitamos incluirlas, podemos personalizar la lista de stop words o deshabilitar esta funcionalidad (requiere configuración avanzada).
3. **Consulta segmentada:** Si trabajamos con grandes volúmenes de datos, consideremos dividir las consultas Full-Text en subconsultas más pequeñas para evitar problemas de rendimiento.
4. **Actualizar estadísticas:** Usemos OPTIMIZE TABLE para mantener los índices Full-Text actualizados y optimizados cuando la tabla sufra muchas inserciones o actualizaciones.
5. **Evitar consultas genéricas:** Definir búsquedas específicas para reducir el número de resultados irrelevantes y mejorar el tiempo de ejecución.

*Para cerrar, ¿por qué es importante la **tokenización**?*

- Velocidad: Permite buscar directamente en las palabras clave ya indexadas, sin recorrer todo el texto cada vez.

- Búsquedas avanzadas: Hace posible encontrar textos relevantes basándose en las palabras contenidas, su frecuencia y posición.

- Flexibilidad: Soporta búsquedas por palabras individuales, frases completas o incluso búsquedas semánticas dependiendo del motor.

Índices Hash

1. ¿Qué son los índices Hash?

Un índice Hash es una estructura que utiliza una **función hash** para asignar claves específicas a posiciones dentro de una tabla. En lugar de recorrer toda la tabla o una estructura jerárquica como un B-Tree, el índice hash encuentra datos directamente mediante el cálculo de un "hash" para la clave buscada.

2. Cómo funcionan los índices Hash

- **Generación de Hashes:** Cuando se crea un índice Hash para una columna, MySQL utiliza una función hash para convertir el valor de esa columna en un identificador único, llamado "clave hash".
- **Ubicación Directa:** Durante una consulta, MySQL aplica la misma función hash al valor buscado y utiliza la clave hash resultante para identificar rápidamente la ubicación de los datos asociados. Esto elimina la necesidad de recorrer las filas una por una.
- **Acceso Exacto:** Los índices Hash son ideales para búsquedas con igualdad (= o IN), ya que la clave hash permite localizar directamente el registro con el valor deseado.

3. Características clave

- **Altísima velocidad:** Los índices Hash son extremadamente rápidos para búsquedas exactas porque no requieren navegar estructuras complejas como los B-Tree. Solo hacen un cálculo de hash y acceden directamente al dato.
- **Menor uso de recursos:** Comparados con otros índices, los índices Hash son más ligeros porque no almacenan estructuras complejas, solo claves hash vinculadas a las filas.
- **Limitaciones en búsquedas por rango:** Dado que los hashes no tienen un orden lógico, no pueden ser utilizados para búsquedas como BETWEEN, <, >, o ORDER BY. Solo buscan coincidencias exactas.

4. Aplicación de índices Hash

Los índices Hash son más comunes en tablas que:

- Necesitan búsquedas rápidas por igualdad (= o IN).
- Utilizan el motor de almacenamiento **MEMORY** (ya que este motor los soporta directamente).
- Contienen datos de alta frecuencia de consulta exacta y relativamente estáticos.

Ejemplo

```
CREATE TABLE productos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255),
    precio DECIMAL(10, 2),
    tipo_memoria VARCHAR(50)
) ENGINE=MEMORY;
```

- **CREATE INDEX** idx_tipo_hash ON productos (tipo_memoria) **USING HASH**;

- SELECT * FROM productos WHERE tipo_memoria = 'SSD';

5. Ventajas y limitaciones

Ventajas:

- **Velocidad extrema:** Ideal para tablas con consultas frecuentes de igualdad, ya que el acceso es directo.
- **Eficiencia de recursos:** En tablas MEMORY, los índices Hash reducen el uso de memoria al evitar estructuras más pesadas como los B-Tree.

Limitaciones:

- **Sólo tablas MEMORY:** Los índices Hash no son soportados por otros motores de almacenamiento como InnoDB o MyISAM.
- **Sin búsquedas por rango:** No pueden resolver consultas como por ejemplo:
    - SELECT * FROM productos WHERE precio BETWEEN 50 AND 100;
- **Colisiones de Hash:** Aunque es raro, dos valores diferentes pueden generar el mismo hash (llamado "colisión"), lo que reduce la eficiencia. El motor maneja estas colisiones, pero puede impactar el rendimiento en casos extremos.
- **No soportan ordenamiento:** No se pueden usar para consultas que requieren ORDER BY.

6. Algunas Consideraciones

Cómo se generan las claves hash:

Los índices Hash dependen de la función hash utilizada por MySQL. Esta función convierte un valor de entrada en un identificador único. Sin embargo, como las funciones hash no tienen un orden lógico, no son aplicables para búsquedas secuenciales.

Colisiones y cómo se manejan:

En el raro caso de que dos valores diferentes generen la misma clave hash:

- MySQL utiliza estructuras adicionales para almacenar y resolver las colisiones.
- Aunque esto asegura la precisión de las consultas, puede impactar el rendimiento si hay demasiadas colisiones.

Comparación con índices B-Tree:

Los índices Hash son mejores para búsquedas exactas (= o IN), mientras que los B-Tree son más versátiles y soportan búsquedas por rango (BETWEEN, <, >), ordenamiento (ORDER BY) y búsquedas parciales.

7. ¿Cuándo usar índices Hash?

Usar índices Hash tiene sentido si:

1. Estamos trabajando con tablas **MEMORY**.
2. Las consultas son principalmente de igualdad (WHERE columna = valor).
3. No necesitamos soporte para operaciones por rango ni ordenamiento.

Para cualquier otra situación, los índices basados en B-Tree suelen ser más adecuados.

Motor	Características	Ventajas	Desventajas
InnoDB	- Soporte para transacciones ACID. - Persistencia de datos en disco. - Soporta claves foráneas (FK). - Maneja bloqueos a nivel de fila.	- Alta confiabilidad y recuperación ante fallos. - Relaciones entre tablas mediante FK. - Ideal para datos críticos y complejos.	- Mayor uso de recursos (CPU y memoria). - Más lento para lectura en tablas grandes sin índices.
MyISAM	- No soporta transacciones. - Almacena datos en disco. - Bloqueo a nivel de tabla. - Índices basados en B-Tree.	- Más rápido para consultas de solo lectura. - Menor uso de memoria. - Adecuado para almacenamiento simple y búsquedas rápidas.	- No soporta FK ni transacciones. - Menos confiable: pérdida de datos en fallos del sistema.
MEMORY	- Datos almacenados en memoria RAM. - No persiste los datos (volátil). - Índices basados en Hash o B-Tree.	- Altísima velocidad para consultas. - Ideal para datos temporales. - Bajo uso de recursos en operaciones rápidas.	- Los datos se pierden al reiniciar el servidor. - No soporta grandes volúmenes ni relaciones complejas.