Bases de datos I

Módulo 4. Buenas prácticas y optimización

Profesor: Lucas Catardo

Sesión 20. Buenas prácticas

Septiembre 23 de 2021

CREATE

- Utilizar VARCHAR en lugar de TEXT (hasta 8000 caracteres)
- Evaluar el uso de CHAR y VARCHAR. Utilizar CHAR cuando la longitud de los registros tiene poca variación.
- No usar columnas con tipos de datos FLOAT, REAL O DATETIME como FOREIGN KEY.
- Utilizar CONSTRAINT para mantener la integridad de los datos.
- Evitar claves primarias compuestas.

SELECT

- Evitar el uso de SELECT * FROM tabla. Especificar los campos requeridos.
- Especificar el alias de la tabla delante de cada campo definido en el SELECT.
- Evitar el uso de GROUP BY, DISTINCT y ORDER BY. Evaluar si estas tareas las puede hacer la aplicación que recibe los datos.

WHERE

- Evitar el uso de wildcards en LIKE como "%valor%", ya que el uso del comodín al inicio de la cadena obliga al DBMS a buscar en todos los registros.
- En subconsultas, utilizar EXISTS y NOT EXISTS en lugar de IN y NOT IN.
- Evitar utilizar funciones dentro de las condiciones del WHERE.

UNION

- Utilizar UNION ALL en lugar de UNION cuando se sabe que los registros no se repiten, para evitar la ejecución implícita de un DISTINCT.

CRUD

- Utilizar SET NOCOUNT ON en operaciones CRUD para evitar el conteo de las filas afectadas.

Orden de Procesamiento de una Query

¿Cómo se escribe?
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY

¿Cómo se ejecuta?
FROM
WHERE
GROUP BY
HAVING
SELECT
ORDER BY

En detalle:
FROM
ON
JOIN
WHERE (El motor no interpreta los alias de columnas porque aún no existen).
GROUP BY (Ídem)
HAVING
SELECT (Se crean las columnas con alias)
DISTINCT
ORDER BY (Se pueden utilizar alias)
LIMIT (Es una opción de visualización, no de cálculo)

Índices

Es una estructura de datos que mejora la velocidad de las consultas, por medio de un identificador único de cada fila de una tabla, permitiendo un rápido acceso a los registros.

Ventajas
- Mejora el rendimiento de las consultas
- Mejora sustancialmente el rendimiento de las consultas que contienen agregaciones y combinaciones (GROUP BY, JOIN).

Desvantajas
- Las tablas utilizadas para almacenar los índices consumen espacio.
- Cuando se realizan operaciones de actualización, inserción o borrado, se requiere actualizar las tablas de los índices asociadas.

Recomendaciones:
- Evitar crear índices en tablas que se actualizan frecuentemente
- Usar claves cortas en lso índices agrupados. Los índices agrupados mejoran si se crean en columnas únicas o que no admiten valores NULL.

Tipos de índices

- Simple.
Definido sobre una sola columna.

CREATE INDEX "I_libros_autor"
ON "libros" (autor);

- Compuesto.
Formado por varias columnas de la misma tabla.

CREATE INDEX "I_libros_autoreditorial"
ON "libros" (autor, editorial);

- Agrupado (CLUSTERED)
Almacena los datos de las filas en orden. Solo se puede crear un único índice agrupado en una tabla de base de datos. Esto funciona de manera eficiente únicamente si los datos se ordenan en orden creciente o decreciente.

CREATE CLUSTERED INDEX "I_libros_autor"
ON "libros" (autor);

- No agrupado
Organiza los datos de forma aleatoria, pero el ínidice especifica internamente un orden lógico. El orden del índice no es el mismo que el ordenamiento físico de los datos. Los índices no agrupados funcionan bien con tablas donde los datos se modifican con frecuencia y el índice se crea en las columnas utilizadas en orden por las declaraciones WHERE y JOIN.

CREATE NONCLUSTERED INDEX "I_libros_autor"
ON "libros" (autor);

Sintaxis:

Crear un índice:
CREATE INDEX "nombre_indice"
ON "nombre_tabla" (nombre_columna);

Eliminar un índice:
ALTER TABLE "nombre_tabla"
DROP INDEX "nombre_indice";

Analizar y almacenar la distribución de claves para una tabla:
ANALYZE TABLE nombre_tabla;

Preguntas

- ¿Los textos largos y formateados de la base de datos se almacenan en la base de datos?
- ¿qué vemos en Bases de datos II?
- ¿qué función tienen los índices, cómo y cuándo se usan?
- ¿qué elementos adicionales hay que estudiar de bases de datos relacionales?
- ¿cómo se migra una base de datos de mysql, por ej, a postgreSQL?
- ¿qué diferencias significativas hay entre bases de datos relacionales?

Sincrónico

Subconsultas:

```
select * from video as v
where idVideo not in (select Video_idVideo from playlist_video where Video_idVideo = v.idVideo)
```

Es preferible usar not exists

```
select * from video as v
where not exists (select idVideo from playlist_video where Video_idVideo = v.idVideo)
```

Para eliminar índices con FK:

Refrescar los índices para que sean más performantes (se pueden haber fragmentado):
Reorganize
Rebuild
(Funciona en SQL Server)

CLUSTERED INDEX / NONCLUSTERED INDEX no se pueden generar en MySQL.

Sesión 21.

Sincrónico.

Todo lo que está en el SELECT debe estar en el GROUP BY.

Para filtrar datos nulos, la versión estándar es:

WHERE dato IS NULL;

La siguiente versión:
WHERE ISNULL(dato); puede no funcionar en algunos motores.

Sesión 22. Profundicemos.

Septiembre 28 de 2021

Sesión 23. Stored Procedures

Septiembre 30 de 2021

Stored procedure:
Son un conjunto de instrucciones en formato SQL que se almacenan, compilan y ejecutan dentro del servidor de bases de datos.
Por lo general, se los utiliza para definir la lógica del negocio dentro de la base de datos y reducir la necesidad de codificar dicha lógica en programas clientes.

Para crear un stored procedure:

```
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento()
BEGIN
	-- Bloque de instrucciones SQL
END $$
```

Borrar un stored procedure:

```
DROP PROCEDURE [IF EXISTS] sp_nombre_procedimiento();
```

Variables
Las variables son elementos que almacenan datos que pueden ir cambiando a lo largo de la ejecución.

Para declarar una variable

```
DECLARE nombre_variable TIPO_DE_DATO [DEFAULT valor];
```

Para asignar un valor a una variable:

```
SET nombre_variable = expresión;
```

Ejemplo:

```
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento()
BEGIN
	DECLARE salario FLOAT DEFAULT 1000.00;
	SET salario = 25700.50;
END $$
```

Parámetros
Son variables por medio de las cuales se envían y se reciben datos de programas clientes. Se definen dentro de la cláusula CREATE. Existen tres tipos de parámetros:

IN: Recibe datos
OUT: Devuelve datos
INOUT: Sirve para entrada y para salida

Ejemplo:

```
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(INOUT aumento FLOAT)
BEGIN
	SET aumento = aumento + 25700.50;
END $$
// La variable aumento es de Entrada/Salida
```

Para llamar el stored procedure:

```
SET @salario = 2000.00;
CALL sp_nombre_procedimiento(@salario);
SELECT @salario;
```

Ventajas y desventajas:

Ventajas
- Gran velocidad de respuesta. Todo se procesa dentro del servidor.
- Mayor seguridad. Se limita e impide el acceso directo a las tablas donde están almacenados los datos, evitando la manipulación por parte de las aplicaciones clientes.
- Independencia
- Reutilización de código
- Mantenimiento más sencillo. Disminuye el costo de modificación cuando cambian las reglas del negocio.*** (Sería transparente para el cliente. Si se modifica el código del BackEnd, todos los usuarios requerirían actualizar la aplicación).

Desventajas
- Difícil modificación. Si se requiere modificarlo, su definición tiene que ser reemplazada totalmente
- Aumentan el uso de memoria
- Restringidos para una lógica de negocio compleja. (En algunos casos es preferible usar el backend).

Sincrónico

Los Stored Procedures, como las vistas, se asocian a la base de datos. Los índices se asocian a las tablas.

Parámetros
Son variables por medio de las cuales se reciben datos de los programas clientes

IN: Recibe datos
OUT: Devuelve datos
INOUT: Sirve para entrada y para salida

```sql
DELIMITER $$
CREATE PROCEDURE sp_productos(IN filtro_categoria VARCHAR(15))
BEGIN
    SELECT ProductoNombre, PrecioUnitario FROM productos p
    JOIN categorias C ON p.CategoriaID = c.CategoriaID
    WHERE CategoriaNombre = filtro_categoria;
END $$
```

```sql
CALL sp_productos('Seafood');
```

Variables:

```sql
DELIMITER $$
CREATE PROCEDURE sp_cantidad_productos(IN filtro_categoria VARCHAR(15), OUT cantidad INT)
BEGIN
    SELECT count(*) INTO cantidad FROM productos p
    JOIN categorias C ON p.CategoriaID = c.CategoriaID
    WHERE CategoriaNombre = filtro_categoria;
END $$
```

```sql
CALL sp_cantidad_productos('Seafood', @cant_seafood);
SELECT @cant_seafood;
```

La variable almacena un solo valor.

```sql
CREATE PROCEDURE sp_nombre_procedimiento(INOUT param1 TIPO_DE_DATO, INOUT param2 TIPO_DE_DATO);
```

Ejemplo:

```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(INOUT aumento FLOAT)
BEGIN
    SET aumento = aumento + 25700.50;
END $$
```

Ejecución:

```sql
SET @salario = 2000.00; -- Declaración y asignación de variable (dato)
CALL sp_nombre_procedimiento(@salario); -- Ejecución y envío de dato (2000.00)
SELECT @salario; -- Muestra el resultado
```

El INTO solo funciona cuando se ejecuta un SELECT; mientras que el SET se puede utilizar en cualquier caso.

No solo funciona con funciones de agregación. Pero sí requiere que se consulte solo un datos de un solo registro; además debe coincidir los tipos de datos. Ejemplo

```sql
SET @Precio = 0;

SELECT PrecioUnitario INTO @Precio FROM productos WHERE ProductoID = 6;

SELECT @Precio;
```

Otra opción:

```sql
SET @Precio = 0;

SET @Precio = (SELECT PrecioUnitario FROM productos WHERE ProductoID = 6);

SELECT @Precio;
```

Herramientas posiblemente más usadas en BD:

1. Funciones
2. Stored procedure
3. Vistas

```
CREATE PROCEDURE `actualizaprecioDeUnoC1` (IN porcAumento INT, IN id INT, OUT precioAntes DOUBLE, OUT precioActualizado D
BEGIN
    -- Lucas Catardo. 30/09/2021
    SELECT PrecioUnitario INTO precioAntes FROM productos WHERE ProductoID = id;


    UPDATE productos
    SET PrecioUnitario = ((PrecioUnitario * porcAumento) / 100) + PrecioUnitario
    WHERE ProductoID = id;

    SET precioActualizado = (SELECT PrecioUnitario FROM productos WHERE ProductoID = id);
END
```

Y para llamar la función:

```
CALL actualizaprecioDeUnoC1(25, 27, @antes, @despues);

SELECT @antes, @despues;
```

Integración de Instrucciones DDL y DML:

Crear una tabla:

```
DELIMITER $$
CREATE PROCEDURE sp_crear_tabla()
BEGIN
	CREATE TABLE nombre_tabla (
		id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
		descripcion VARCHAR(200));
END $$

CALL sp_crear_tabla(); --De forma similar se pueden utilizar ALTER TABLE y DROP TABLE.
```

Insertar datos:

```
DELIMITER $$
CREATE PROCEDURE sp_agregar_usuario(
	IN nombre VARCHAR(30), IN apellido VARCHAR(30))
BEGIN
	INSERT INTO usuario (nombre, apellido) VALUES (nombre, apellido);
END $$

CALL sp_agregar_usuario('DIEGO', 'PEREZ'); -- Los datos a insertar se pasan como parámetros IN. De forma similar se pueden utilizar UPDATE y DELETE.
```

Instrucción SELECT con IN – OUT

```
DELIMITER $$
CREATE PROCEDURE sp_dame_nombre_usuario(
        INOUT id INT, OUT nom VARCHAR(30))
BEGIN
        SELECT nombre INTO nom FROM usuario WHERE id_usuario = id;
END $$

CALL sp_dame_nombre_usuario(1,@nombre); -- Ejecución del SP y envía “1” como dato
SELECT @nombre; -- Muestra el resultado
```

Instrucción SELECT con IN – OUT

También se puede utilizar un mismo parámetro para la entrada y la devolución del resultado:

```
DELIMITER $$
CREATE PROCEDURE sp_dame_nombre_usuario(INOUT valor VARCHAR(30))
BEGIN
        SELECT COUNT(*) INTO valor FROM usuario
        WHERE nombre LIKE CONCAT('%', valor, '%');
END $$

SET @letra = 'a'; -- Declaración y asignación de una variable (dato)
CALL sp_dame_nombre_usuario(@letra); -- Ejecución del SP y envía “a” como dato
SELECT @letra;
```