Tema 11: SQL Avanzado

Introducción a la Ingeniería del Software y los Sistemas de Información I

Ingeniería Informática – Tecnologías Informáticas

Departamento de Lenguajes y Sistemas Informáticos

Índice

Introducción

La mayoría de los motores de los SGBD permiten implementar operaciones avanzadas. Las ventajas de esta función son las siguientes:

- **Rendimiento**: Al ejecutarse directamente en el motor se consigue un mejor rendimiento que si hiciéramos las operaciones en lenguajes de alto nivel como Java, C, etc.
- **Consistencia e integridad:** Podemos incorporar reglas de integridad directamente en el motor de manera que tenemos mayores garantías de conservar la integridad y consistencia de los datos para todas las aplicaciones que trabajen con la Base de datos.

Lenguajes de los motores de SGBD relacionales más populares:

- MS SQL Server: T-SQL
- Oracle: PL/SQL
- PostgreSQL: PL/pgSQL
- MySQL/MariaDB: SQL/compound statements

Introducción: Tipos de objetos

Procedimientos (stored procedures)

- Pueden implementar lógica que **modifica datos en las tablas**.
- No se pueden usar en un `SELECT`, sino mediante `CALL.`

Funciones (stored functions)

- Procedimiento que **devuelve un valor**.
- Usado para realizar **cálculos que no modifican datos en tablas**.
- Se pueden usar en un `SELECT` (y `WHERE`).

Disparadores (triggers)

- Código asociado a **eventos** (como `INSERT, UPDATE, DELETE` en tablas).
- Se disparan automáticamente **antes o después del evento** al que se asocian.
- Usados para **validación compleja de datos**, **reglas de negocio**, **mantenimiento de integridad** de datos.
- **Pueden llamar a funciones** en su cuerpo, pero no procedimientos.

Vistas (views)

- **Simplifican** el acceso a datos complejos y consultas repetitivas.
- Actúan como ***tablas virtuales*** (pueden usarse en `SELECT, JOIN`, etc.).
- **No almacenan datos propios**, son una representación de una consulta sobre otras tablas.

Introducción: Tipos de objetos

Tipo de objeto	Parámetros	INSERT/UPDATE/DELETE	Llamada	Return
Procedimientos	SÍ	SÍ	CALL sp_name;	NO (1)
Funciones	SÍ	NO (2)	func_name()	SÍ
Triggers	NO (3)	SÍ	NO	NO
Vistas	NO	NO	Select nombre_vista;	NO

(1) Pueden tener parámetros de entrada/salida

(2) Sólo pueden hacer selects

(3) Variables:

- **'old'** (cuando están asociados a eventos de operaciones UPDATE o DELETE). Para acceder a los valores de la fila previos a la actualización o el borrado (valores *antiguos*)
- **'new'** (para eventos de operaciones INSERT o UPDATE). Para acceder a los valores de la fila que se van a insertar o actualizar (valores *nuevos*)

Introducción: Sintaxis general

```
DELIMITER //
CREATE OR REPLACE …
BEGIN
   stmt_sql1;
   stmt_sql2;
   …
END//
-- Otras
funciones/proc/triggers
DELIMITER ;
```

Indica que las siguientes instrucciones acaban con el delimitador **//**

Bloque de instrucciones BEGIN-END, separadas por el delimitador **;**

Indica el final de la instrucción, ya que aparece el delimitador **//**, definido al principio.

A partir de este punto, se vuelve a utilizar como delimitador de instrucciones el **;**

Procedimientos: Sintaxis

```
CREATE
    [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MariaDB data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    Valid SQL procedure statement
```

Introducción: Palabras reservadas

Compound Statements:

- BEGIN END
- CASE
- DECLARE CONDITION
- DECLARE HANDLER
- DECLARE nombreVariable
- FOR
- GOTO
- IF
- ITERATE
- :labels
- LEAVE
- LOOP
- REPEAT LOOP
- RESIGNAL
- RETURN
- SELECT INT
- SET vble
- SIGNAL
- WHILE

Procedimientos: Ejemplos

```
-- Insertar un nuevo Departamento
DELIMITER //
CREATE OR REPLACE PROCEDURE
  pInsertarDepartamento (n VARCHAR(32), l VARCHAR(64))
BEGIN
  INSERT INTO Departamentos (nombreDep, localidad) VALUES (n, l);
END//
DELIMITER ;

-- Llamada a procedimiento
CALL pInsertarDepartamento('Economía', 'Almería');
```

Procedimientos: Asignación de valores

```
-- Insertar un nuevo Empleado.
-- Si la fechaInicial es null, poner la fecha actual del sistema
DELIMITER //
CREATE OR REPLACE PROCEDURE
  pInsertarEmpleado (d INT, j INT, n VARCHAR(64), s DECIMAL, fi DATE,
                     ff DATE,c DOUBLE)
BEGIN
  IF (fi IS NULL) THEN
    SET fi = SYSDATE();
  END IF;
  INSERT INTO Empleados (departamentoId, jefe, nombre, salario,
                         fechaInicial, fechaFinal, comision)
   VALUES (d, j, n, s, fi, ff, c);
END //
DELIMITER ;


-- Llamada al procedimiento
CALL pInsertarEmpleado (1, NULL, 'Daniel', 500, NULL, '2020-09-15', 0.2);
```

Asignación de un valor a una variable

Procedimientos: Declaración de variables

```
-- Procedimiento para igualar las comisiones de todos los empleados a
-- la media de las comisiones.
DELIMITER //
CREATE OR REPLACE PROCEDURE
  pIgualarComisiones()
BEGIN
  DECLARE comisionMedia DOUBLE;
  SET comisionMedia = (SELECT AVG(comision) FROM Empleados);
  UPDATE Empleados SET comision = comisionMedia;
 END //
DELIMITER ;
-- Llamada al procedimiento
CALL pIgualarComisiones();
```

Se obtiene **UN** valor para asignarlo a la variable

Declaración de una variable

empleadoId	departamentoId	jefe	nombre	salario	fechaInicial	fechaFinal	comision
1	1	(NULL)	Pedro	2.500,00	2017-09-15	(NULL)	0,19999999999999998
2	1	(NULL)	José	2.500,00	2018-08-15	2019-08-15	0,19999999999999998
3	2	(NULL)	Lola	2.300,00	2018-08-15	(NULL)	0,19999999999999998
4	1	1	Luis	1.300,00	2018-08-15	2018-11-15	0,19999999999999998
5	(NULL)	1	Ana	1.300,00	2018-08-15	2018-11-15	0,19999999999999998
6	1	(NULL)	Daniel	500,00	2019-10-28	2020-09-15	0,19999999999999998

Procedimientos: Tipo fila “ROW TYPE OF”

```
-- Aplicar un aumento dado a la comisión de empleado concreto
DELIMITER //
CREATE OR REPLACE PROCEDURE
  pSubirComision(id INT, aumento DOUBLE)
BEGIN
  DECLARE empleado ROW TYPE OF Empleados;
  DECLARE nuevaComision DOUBLE;
  SELECT * INTO empleado
    FROM Empleados
    WHERE empleadoId = id;
  SET nuevaComision = empleado.comision + aumento;
  UPDATE Empleados
    SET comision = nuevaComision
    WHERE empleadoId = id;
END //
DELIMITER ;
-- Llamada al procedimiento
CALL pSubirComision(1,0.1);
```

Se obtiene **UNA FILA** para asignarla a la variable

Declaración de una variable de tipo fila

empleadoId	departamentoId	jefe	nombre	salario	fechaInicial	fechaFinal	comision
1	1	(NULL)	Pedro	2.500,00	2017-09-15	(NULL)	0,3
2	1	(NULL)	José	2.500,00	2018-08-15	2019-08-15	0,19999999999999998
3	2	(NULL)	Lola	2.300,00	2018-08-15	(NULL)	0,19999999999999998
4	1	1	Luis	1.300,00	2018-08-15	2018-11-15	0,19999999999999998
5	(NULL)	1	Ana	1.300,00	2018-08-15	2018-11-15	0,19999999999999998
6	1	(NULL)	Daniel	500,00	2019-10-28	2020-09-15	0,19999999999999998

Funciones: Sintaxis

```
CREATE [OR REPLACE]
    [DEFINER = {user | CURRENT_USER | role | CURRENT_ROLE }]
    [AGGREGATE] FUNCTION [IF NOT EXISTS] func_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...]
    RETURN func_body

func_parameter:
    param_name type

type:
    Any valid MariaDB data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

func_body:
    Valid SQL procedure statement
```

Funciones: Ejemplos

```
-- Devuelve el número de empleados de una localidad
DELIMITER //
CREATE OR REPLACE FUNCTION
  fNumEmpleados(loc VARCHAR(64)) RETURNS INT
BEGIN
  RETURN (
    SELECT COUNT(*)
    FROM empleados E JOIN departamentos D
    ON (E.departamentoId = D.departamentoId)
    WHERE D.localidad = loc
  );
END//
DELIMITER ;
```

```
SELECT D.descripción, D.localidad,
fNumEmpleados(D.localidad)
FROM Departamentos
WHERE localidad LIKE 'Sevilla';
```

Funciones: Uso en procedimientos

```
-- Ejemplo de uso de funciones dentro de procedimientos
-- deben almacenarse en variables
DELIMITER //
CREATE OR REPLACE FUNCTION
  fComisionMedia() RETURNS DOUBLE
BEGIN
  RETURN (
    SELECT AVG(comision)
    FROM empleados
  );
END //
DELIMITER ;
-- El procedimiento puede usar la función
DELIMITER //
CREATE OR REPLACE PROCEDURE
  pIgualarComisiones()
BEGIN
  DECLARE cm DOUBLE;
  SET cm = fComisionMedia();
  UPDATE Empleados SET comision = cm;
END//
DELIMITER ;
```

empleados (5r × 2c)

nombre	comision
Pedro	0,155520...
José	0,155520...
Lola	0,155520...
Luis	0,155520...
Ana	0,155520...

Trigger: Sintaxis

```
CREATE [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    TRIGGER [IF NOT EXISTS] trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW
   [{ FOLLOWS | PRECEDES } other_trigger_name ]
   trigger_stmt
```

trigger_time = {BEFORE, AFTER}

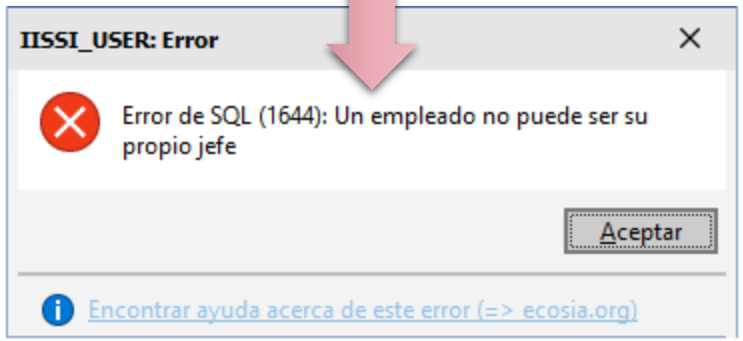
trigger_event = {INSERT, UPDATE, DELETE}

FOLLOWS | PRECEDES other_trigger_name: Añade “other_trigger” después/antes del actual

Trigger: Ejemplo simple

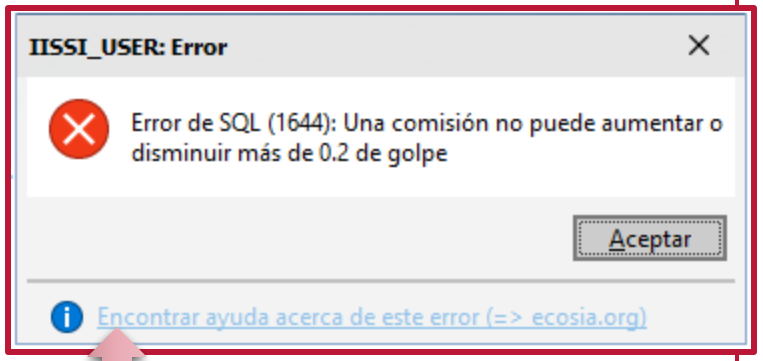
```
DELIMITER //
-- Comprueba que un empleado no es su propio jefe
CREATE OR REPLACE TRIGGER
  tEmpleadoPropioJefe
BEFORE UPDATE ON Empleados FOR EACH ROW
BEGIN
     IF (new.empleadoId = new.jefe) THEN
          SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
          'Un empleado no puede ser su propio jefe';
     END IF;
END //
DELIMITER ;
```

```
UPDATE Empleados SET jefe=1 WHERE empleadoId = 1;
```



Trigger: Ejemplo para reglas de negocio complejas

```
-- Ejemplo de Trigger para comprobar que las variaciones sobre
-- la comisión de los Empleados no puede cambar en más 0.2 puntos
-- OPCIÓN 1: No se permite realizar el cambio en la comisión
DELIMITER //
CREATE OR REPLACE TRIGGER tCambiosComision
BEFORE UPDATE ON empleados FOR EACH ROW
BEGIN
  IF((new.comision - old.comision) > 0.2 OR ((new.comision - old.comision) < -0.2)) THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
      'Una comisión no puede aumentar o disminuir más de 0.2 de golpe';
    END IF;
END //
DELIMITER ;
-- OPCIÓN 2: Se permita realizar el cambio al valor máximo permitido
DELIMITER //
CREATE OR REPLACE TRIGGER tCambiosComision
BEFORE UPDATE ON empleados FOR EACH ROW
BEGIN
  IF((new.comision - old.comision) > 0.2) THEN
    SET new.comision = old.comision + 0.2;
  END IF;
  IF((new.comision - old.comision) < -0.2) THEN
    SET new.comision = old.comision - 0.2;
  END IF;
END //
```

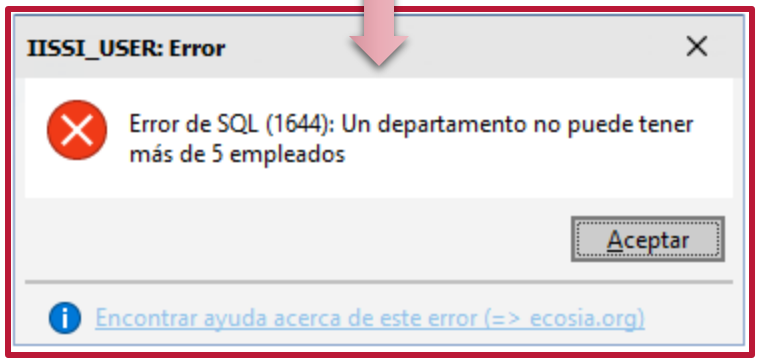



```
UPDATE Empleados SET comision=0.9 WHERE empleadoId = 1;
```

Trigger: Ejemplo para reglas de negocio complejas

```
-- RN003. Los departamentos no tienen más de 5 empleados
DELIMITER //
CREATE OR REPLACE TRIGGER tMaxEmpleadosDepartamento
BEFORE INSERT ON Empleados
FOR EACH ROW
BEGIN
     DECLARE n INT;
     SET n = (SELECT COUNT(*) FROM Empleados WHERE
     departamentoId = new.departamentoId);
     IF (n >= 5) THEN
          SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
          'Un departamento no puede tener más de 5 empleados';
     END IF;
END //
DELIMITER ;
```

```
CALL createEmpleado (1, NULL, 'Rafael', 500, NULL, '2020-09-15', 0.2);
```



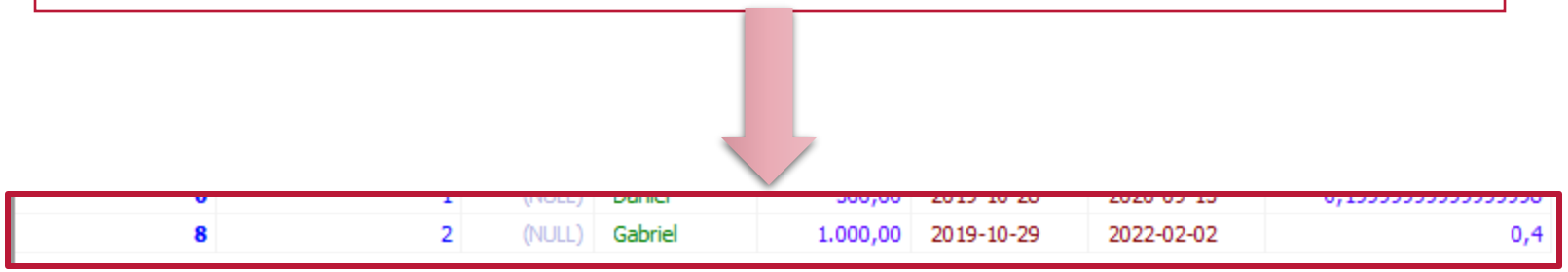
Trigger: Ejemplo para reglas de negocio complejas

```
-- RN003. Los departamentos no tienen más de 5 empleados
DELIMITER //
CREATE OR REPLACE TRIGGER tMaxEmpleadosDepartamentoUpdate
BEFORE UPDATE ON Empleados
FOR EACH ROW
BEGIN
     DECLARE n INT;
     SET n = (SELECT COUNT(*) FROM Empleados WHERE
          departamentoId = new.departamentoId);
     IF (n >= 5) THEN
          SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
          'Un departamento no puede tener más de 5 empleados';
     END IF;
END //
DELIMITER ;
```

Trigger: Sustitución de valores

```
DELIMITER //
-- Si un empleado no tiene fecha inicial, usa la actual
CREATE OR REPLACE TRIGGER fechaInicialDefault
BEFORE INSERT ON Empleados
FOR EACH ROW
BEGIN
     IF (new.fechaInicial IS NULL) THEN
          SET new.fechaInicial = SYSDATE();
     END IF;
END //
DELIMITER ;
```

```
INSERT INTO Empleados
VALUES (NULL, 2, NULL, 'Gabriel', 1000, NULL, '2022-02-02', 0.4);
```



8	2	(NULL)	Gabriel	1.000,00	2019-10-29	2022-02-02	0,4

Vistas

Consulta almacenada que presenta una ***tabla virtual*** basada en los datos de **una o más tablas**.

No almacenan datos propios, se consultan las tablas subyacentes cuando se accede a la vista.

Puede utilizarse en consultas `SELECT` como una tabla real.

No se pueden realizar operaciones de modificación (`INSERT, UPDATE` o `DELETE`) sobre la vista.

Si la **estructura** de las tablas subyacentes se modifica, la vista **no se actualiza automáticamente**.

Ventajas:

- **Facilidad de acceso a datos complejos**, encapsulando la lógica en una estructura reutilizable.
- **Seguridad y control de acceso,** mostrando solo las columnas y filas necesarias para cada usuario de la BD.
- **Consistencia**, evitando duplicaciones de la lógica.

Vistas: Ejemplo – creación de tablas subyacentes

```sql
-- Crear tabla de Productos
CREATE TABLE Productos
(
	id INT PRIMARY KEY AUTO_INCREMENT,
	nombre VARCHAR(100),
	precio DECIMAL(10, 2)
);
-- Crear tabla de Ventas
CREATE TABLE Ventas
(
	id INT PRIMARY KEY AUTO_INCREMENT,
	producto_id INT,
	cantidad INT,
	fecha DATE,
	FOREIGN KEY (producto_id) REFERENCES productos(id)
);
```

Vistas: Ejemplo – creación de vista y uso

```sql
-- Crear la vista que calcula los productos más populares
CREATE VIEW ProductosMasPopulares AS
SELECT
	p.id AS id_producto,
	p.nombre AS nombre_producto,
	SUM(v.cantidad) AS total_vendido
FROM
	Productos p
JOIN
	Ventas v ON p.id = v.producto_id
GROUP BY p.id, p.nombre
ORDER BY total_vendido DESC;
```

```sql
-- Consultar los productos más populares
SELECT * FROM ProductosMasPopulares;
-- Filtrar para obtener productos con más de 15 unidades vendidas
SELECT * FROM ProductosMasPopulares WHERE total_vendido > 15;
```

Tema 11: SQL Avanzado

Introducción a la Ingeniería del Software y los Sistemas de Información I

Ingeniería Informática – Tecnologías Informáticas

Departamento de Lenguajes y Sistemas Informáticos