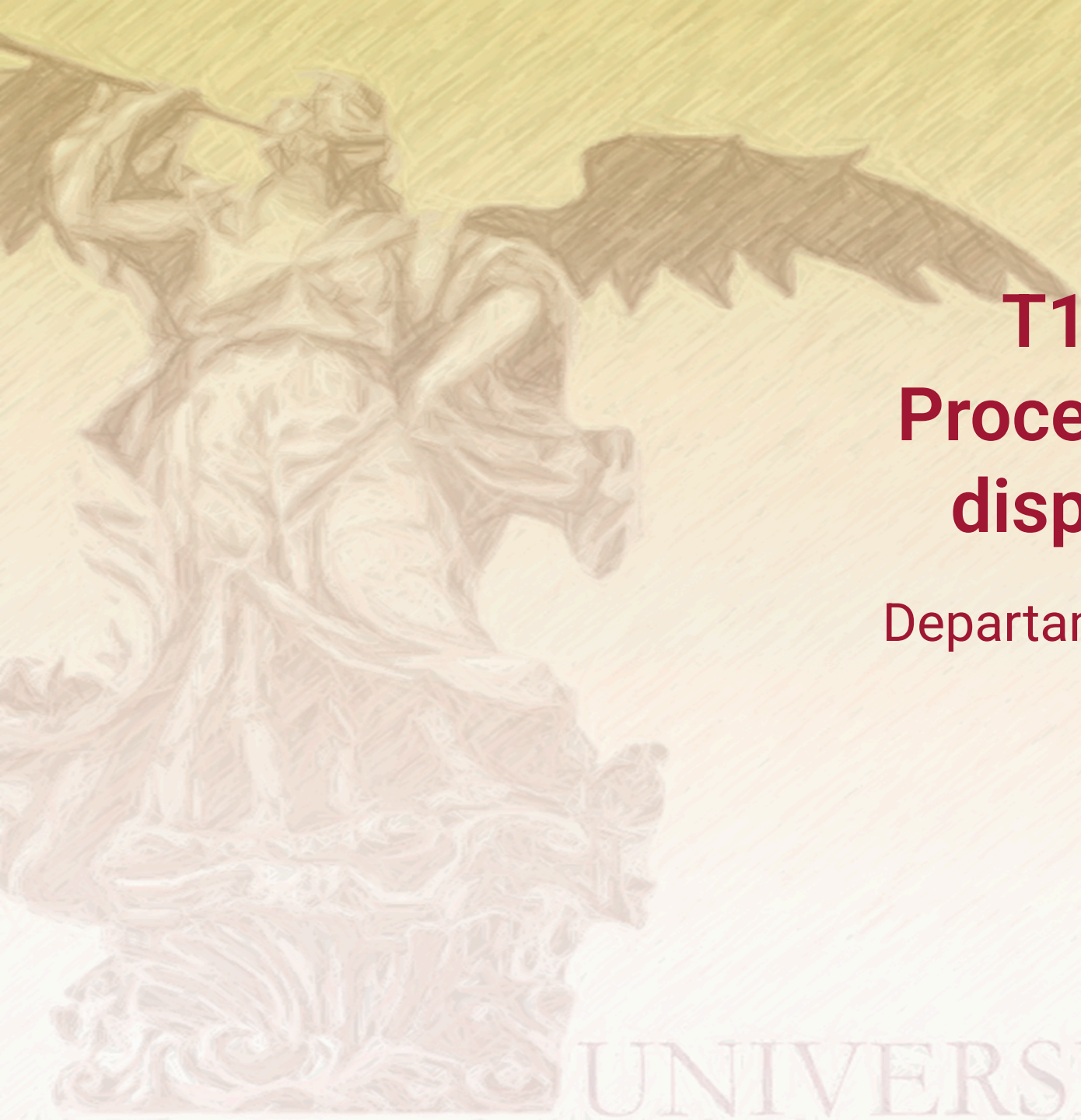


T11 - SQL Avanzado: Procedimientos, funciones, disparadores y cursores

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla

Objetivos


Sintaxis
Procedimiento
Introducción
Trigger
Contenido
Manejo
Funciones
Propio
Procedimientos
Disparadores
Lógica
Declaración
Usar
Cursores
Lenguaje
Directamente
Permitiendo
Pueden
Datos
Variables

Contenido

- Introducción
- Procedimientos
- Funciones
- Disparadores
- Cursores

Contenido

Introducción

Procedimientos

Funciones

Disparadores

Cursores

Introducción

MS SQL Server: T-SQL (lenguaje propio)

- Declaración de variables
- Control de transacciones
- Manejo de excepciones
- Manejo de Strings, Fechas, ...

Oracle: PL/SQL (lenguaje propio)

- Parecido a T-SQL pero algo más complicado

PostgreSQL: PL/pgSQL (lenguaje propio)

- Destaca el manejo de las funciones

MySQL/MariaDB: SQL/compound statements

- Bastantes limitaciones

Tipos de objetos

Procedimientos (stored procedures)

- Programa almacenado en la BD
- Se ejecuta directamente en el motor del SGBD
- Pueden implementar lógica
- No se pueden usar desde un `SELECT`, sino desde `CALL`

Funciones (stored functions)

- Procedimiento que devuelve un valor
- Se pueden usar en un `SELECT`

Disparadores (triggers)

- Código asociado a eventos
- Validación compleja de datos y reglas de negocio

Cursores (cursors)

- Iteradores sobre resultados devueltos por `SELECT`

Comparación

	Param	INS/UPD/DEL	Llamada	Return
Procedimientos	SI	SI	`CALL sp_name;`	NO (1)
Funciones	SI	NO (2)	`func_name()`	SI
Triggers	NO (3)	SI	NO	NO
Vistas	NO	NO	`SELECT nombre_vista;`	NO

(1) Pueden tener parámetros de entrada/salida
(2) Sólo pueden hacer `SELECT`
(3) Variables `OLD` (UPD/DEL) y `NEW` (INS/UPD)

Palabras reservadas(MariaDB, MySQL)

```
BEGIN  END
CASE
DECLARE CONDITION
DECLARE HANDLER
DECLARE vble
FOR
GOTO
IF
ITERATE
:labels
```

```
LEAVE
LOOP
REPEAT LOOP
RESIGNAL
RETURN
SELECT INT
SET vble
SIGNAL
WHILE
```

Sintaxis general

Son instrucciones SQL dentro de unbloque `BEGIN` - `END` deparadas por `;` por lo que hay que usar `DELIMITER //` para indicar dónde empieza el bloque de instrucciones y donde termina `DELIMITER ;`

```
DELIMITER //
CREATE OR REPLACE …
BEGIN
  stmt_sql1;
  stmt_sql2;
  …
END//
DELIMITER ;
```

Contenido

- Introducción
- **Procedimientos**
- Funciones
- Disparadores
- Cursores

Ejemplo: BD empleados (Repo GIT)

```sql
-- Código disponible en reposito Git
CREATE OR REPLACE TABLE departments (
  department_id INT NOT NULL AUTO_INCREMENT,
  name_dep VARCHAR(32),
  city VARCHAR(64),
  UNIQUE (name_dep, city),
  PRIMARY KEY(department_id)
);

CREATE OR REPLACE TABLE employees (
  employee_id INT NOT NULL AUTO_INCREMENT ,
  department_id INT,
  boss_id INT,
  name_emp VARCHAR(64) NOT NULL,
  start_date DATE,
  end_date DATE,
  salary DOUBLE DEFAULT 2000,
  fee DOUBLE,
  PRIMARY KEY(employee_id),
  FOREIGN KEY (department_id) REFERENCES departments (department_id)
    ON DELETE RESTRICT,
  FOREIGN KEY (boss_id) REFERENCES employees (employee_id),
  UNIQUE(name_emp),
  CONSTRAINT valid_fee CHECK (fee>=0 AND fee<=1),
  CONSTRAINT valid_dates CHECK (start_date < end_date),
  CONSTRAINT valid_salary CHECK (salary > 0)
);
```

Sintaxis de procedimientos

```
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
```

Procedimiento: `p_populate`

Se suele usar un procedimiento para hacer la **carga incial** de datos, que también sirve como **tests positivos**.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_populate()
BEGIN
  SET FOREIGN_KEY_CHECKS = 0;
  DELETE FROM employees;
  DELETE FROM departments;
  SET FOREIGN_KEY_CHECKS = 1;

  INSERT INTO departments (department_id, name_dep, city) VALUES
    (1, 'Arte', 'Cádiz'),
    (2, 'Historia', NULL),
    (3, 'Informática', 'Sevilla')
  ;
  INSERT INTO employees (employee_id, department_id, boss_id, name_emp, salary, start_date, end_date, fee) VALUES
    (1, 1, NULL, 'Pedro', 2300.00, '2017-09-15', NULL, 0.2),
    (2, 1, NULL, 'Jose', 2500.00, '2018-08-15', NULL, 0.5),
    (3, 2, NULL, 'Lola', 2300.00, '2018-08-15', NULL, 0.3),
    (4, 1, 1, 'Luis', 1300.00, '2018-08-15', '2018-11-15', 0),
    (5, 1, 1, 'Ana', 1300.00, '2018-08-15', '2018-11-15', 0)
  ;
END;
//
DELIMITER ;
```

Procedimiento: `p_insert_department`

```
DELIMITER //
CREATE OR REPLACE PROCEDURE
  p_insert_department(
    p_name_dep VARCHAR(32),
    p_city VARCHAR(64))
BEGIN
  INSERT INTO departments (name_dep, city) VALUES (p_name_dep, p_city);
END//
DELIMITER ;
```

```
CALL p_populate();
CALL p_insert_department('Economía', 'Almeria');

-- Insertar departamento duplicado:
-- CALL p_insert_department ('Economía', 'Almeria');
```

Procedimiento: `p_insert_employee`

Asignación de valores

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_insert_employee (
  p_employee_id INT,
  p_department_id INT,
  p_boss_id INT,
  p_name_emp VARCHAR(64),
  p_start_date DATE,
  p_end_date DATE,
  p_salary DOUBLE,
  p_fee DOUBLE)
BEGIN
  IF (p_start_date IS NULL) THEN
    SET p_start_date = SYSDATE();
  END IF;
  INSERT INTO employees (employee_id, department_id, boss_id, name_emp, salary, start_date, end_date, fee)
  VALUES (p_employee_id, p_department_id, p_boss_id, p_name_emp, p_salary, p_start_date, p_end_date, p_fee);
END //
DELIMITER ;

-- CALL p_populate();
-- CALL p_insert_employee(6, 1, NULL, 'Daniel', '2020-09-15', NULL, 2500.0, 0.2);
```

Procedimiento: `p_equate_fees`

Declaración de variables, set

```
-- Igualar la comisión de TODOS al valor de la comisión media
DELIMITER //
CREATE OR REPLACE PROCEDURE
  p_equate_fees()
BEGIN
  DECLARE avgFee DOUBLE;
  SET avgFee = (SELECT AVG(fee) FROM Employees);
  -- Modifica TODOS los Employees, no tiene WHERE
  UPDATE Employees SET fee = avgFee;
END //
DELIMITER ;
-- CALL p_populate();
-- CALL p_equate_fees();
```

Procedimiento: `p_raise_fee`

Declaración de variables `ROW TYPE OF`, select-into

```
-- Procedimiento para aplicar un aumento en la comisión de un empleado
CREATE OR REPLACE PROCEDURE
  p_raise_fee(id INT, amount DOUBLE)
BEGIN
  DECLARE e ROW TYPE OF employees;
  DECLARE new_fee DOUBLE;
  SELECT * INTO e -- el resultado del select lo almacena en la variable
    FROM employees
    WHERE employee_id = id;
  SET new_fee = e.fee + amount;
  UPDATE employees
    SET fee = new_fee
    WHERE employee_id = id;
END //
```

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

Contenido

- Introducción
- Procedimientos
- **Funciones**
- Disparadores
- Cursores

Sintaxis de funciones

```
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
```

Función: `f_num_employees`

```
-- Devuelve el número de empleados de una localidad
CREATE OR REPLACE FUNCTION
  f_num_employees(c VARCHAR(64)) RETURNS INT
BEGIN
  RETURN (
    SELECT COUNT(*)
    FROM employees e JOIN departments d
    ON (e.department_id = d.department_id)
    WHERE d.city = c
  );
```

Atención: `f_num_employees(null) = 0`

Función: `f_num_employees`

```
SELECT city, COUNT(*)
FROM employees NATURAL JOIN departments
GROUP BY city;
```

departamentos (2r × 2c)

localidad	COUNT(*)
(NULL)	3
Sevilla	1

```
SELECT city, f_num_employees(city)
FROM employees NATURAL JOIN departments
GROUP BY city;
```

departamentos (2r × 2c)

localidad	fNumEmpleados(localidad)
(NULL)	0
Sevilla	1

Uso de funciones en procedimientos

empleados (5r × 2c)

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

```
-- Ejemplo de uso de funciones dentro de procedimientos
-- Definición de la función
CREATE OR REPLACE FUNCTION
  f_avg_fee() RETURNS DOUBLE
BEGIN
  RETURN (
    SELECT AVG(fee)
    FROM employees
  );

-- El procedimiento puede usar la función
DELIMITER //
CREATE OR REPLACE PROCEDURE
  p_equate_fees()
BEGIN
  DECLARE af DOUBLE;
  SET af = f_avg_fee();
  UPDATE employees SET fee = af;
```

Contenido

- Introducción
- Procedimientos
- Funciones
- **Disparadores**
- Cursores

Sintaxis de disparadores

```
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` }

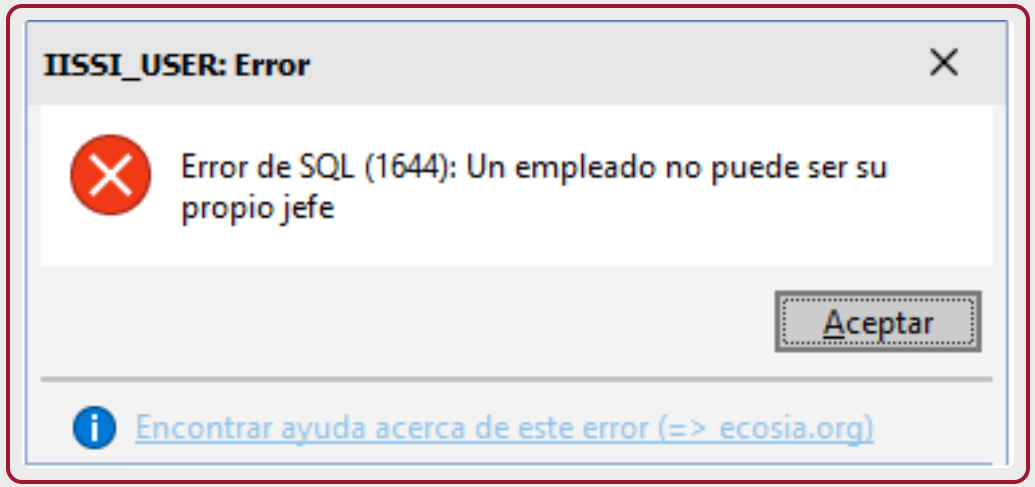
trigger_event = { `INSERT` , `UPDATE` , `DELETE` }

`FOLLOWS` | `PRECEDES` other_trigger: Añade “other_trigger” después/antes

Ejemplo: `t_biu_self_boss`

```
CREATE OR REPLACE TRIGGER t_biu_self_boss
BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN
  IF (NEW.employee_id = NEW.boss_id) THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
      'An employee cannot be his own boss';
  END IF;
END
```

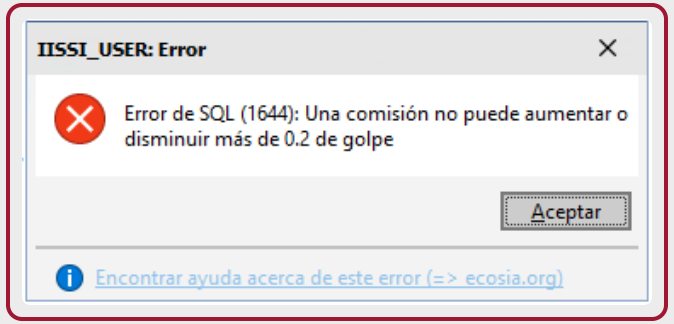
```
UPDATE Empleados SET jefe=1 WHERE empleadoId = 1;
```



Ejemplo: `t_change_fee`

```
-- OPCIÓN 1: No se permite realizar el cambio en la comisión
CREATE OR REPLACE TRIGGER t_change_fee_1
BEFORE UPDATE ON employees FOR EACH ROW
BEGIN
  IF((new.fee - old.fee) > 0.2 OR ((new.fee - old.fee) < -0.2)) THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
      'A fee cannot increase or decrease more than 0.2';
  END IF;
END
-- OPCIÓN 2: Se permite realizar el cambio al valor máximo permitido

CREATE OR REPLACE TRIGGER t_change_fee_2
BEFORE UPDATE ON employees FOR EACH ROW
BEGIN
  IF((new.fee - old.fee) > 0.2) THEN
    SET new.fee = old.fee + 0.2;
  END IF;
  IF((new.fee - old.fee) < -0.2) THEN
    SET new.fee = old.fee - 0.2;
  END IF;
END
```

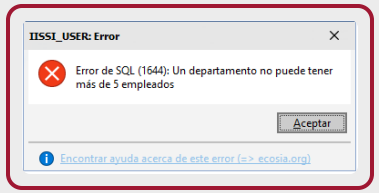



```
UPDATE Empleados SET comision=0.9 WHERE empleadoId = 1;
```

Ejemplo: `t_max_employees_department`

```
CREATE OR REPLACE TRIGGER t_max_employees_department
BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN
  DECLARE n INT;
  SET n = (
    SELECT COUNT(*)
    FROM employees
    WHERE department_id = new.department_id
  );
  IF (n > 4) THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
      'A department cannot have more than 5 employees';
    END IF;
  END
```

```
CALL p_populate_db();
CALL p_insert_employee(6, 1, NULL, 'quinto empleado departamento 1', null, NULL, 1500, 0);
CALL p_insert_employee(7, 1, NULL, 'sexto empleado departamento 1', null, NULL, 1500, 0);
```



Sustitución de valores

```
-- Ejemplo de Trigger para almacenar en la startDate del contrato
-- de un empleado a la fecha actual del sistema en que caso de que la
-- startDate proporcionada sea NULL
DELIMITER //
CREATE OR REPLACE TRIGGER t_default_start_date
BEFORE INSERT ON employees FOR EACH ROW
BEGIN
  IF (new.start_date IS NULL) THEN
    SET new.start_date = SYSDATE();
  END IF;
END //
DELIMITER ;
```

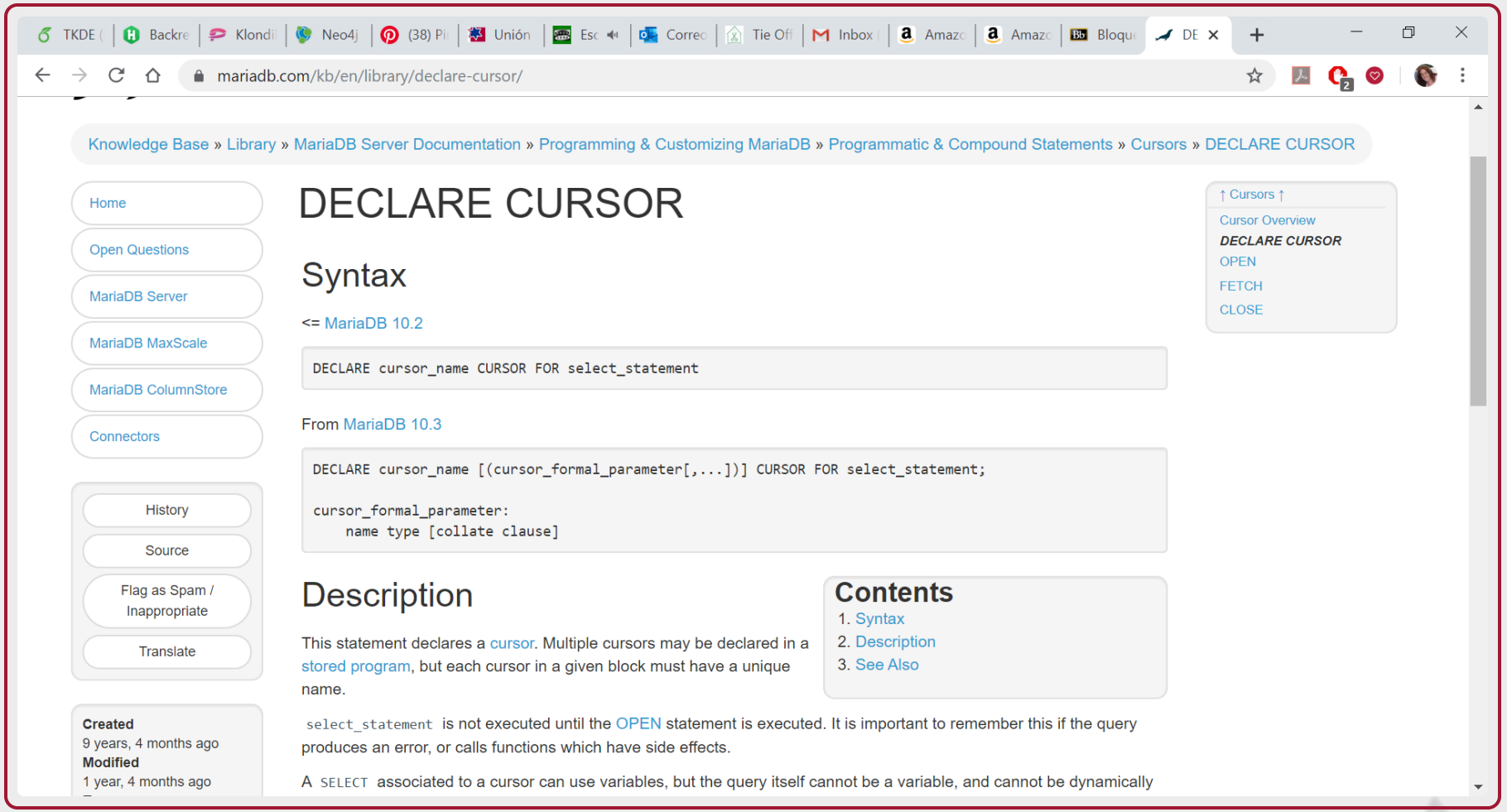
```
INSERT INTO employees
  VALUES (NULL, 2, NULL, 'Gabriel', 1000, NULL, '2022-02-02', 0.4);
```

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

Contenido

- Introducción
- Procedimientos
- Funciones
- Disparadores
- **Cursores**

Sintaxis de cursores



mariadb.com/kb/en/library/declare-cursor/

Knowledge Base » Library » MariaDB Server Documentation » Programming & Customizing MariaDB » Programmatic & Compound Statements » Cursors » DECLARE CURSOR

Home
Open Questions
MariaDB Server
MariaDB MaxScale
MariaDB ColumnStore
Connectors

History
Source
Flag as Spam / Inappropriate
Translate

Created
9 years, 4 months ago
Modified
1 year, 4 months ago

DECLARE CURSOR

Syntax

<= MariaDB 10.2

```
DECLARE cursor_name CURSOR FOR select_statement
```

From MariaDB 10.3

```
DECLARE cursor_name [(cursor_formal_parameter[,...])] CURSOR FOR select_statement;

cursor_formal_parameter:
    name type [collate clause]
```

Description

This statement declares a cursor. Multiple cursors may be declared in a stored program, but each cursor in a given block must have a unique name.

`select_statement` is not executed until the OPEN statement is executed. It is important to remember this if the query produces an error, or calls functions which have side effects.

A `SELECT` associated to a cursor can use variables, but the query itself cannot be a variable, and cannot be dynamically

Contents
1. Syntax
2. Description
3. See Also

↑ Cursors ↑
Cursor Overview
DECLARE CURSOR
OPEN
FETCH
CLOSE

Cursor básico

```
DELIMITER //
-- Obtiene la suma de todos los salarios sin usar directamente SUM() en un select
CREATE OR REPLACE FUNCTION sumSalary()
  RETURNS DECIMAL
BEGIN
  DECLARE sum DECIMAL;
  DECLARE employee ROW TYPE OF employees;
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE curEmployees CURSOR FOR SELECT * FROM employees;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done := TRUE;
  SET sum = 0;
  OPEN curEmployees;
readLoop:LOOP
  FETCH curEmployees INTO employee;
  IF done THEN
    LEAVE readLoop;
  END IF;
  SET sum = sum + employee.salary;
  END LOOP;
CLOSE curEmployees;
  RETURN sum;
END //
DELIMITER ;
```

Conclusiones

SQL avanzado proporciona elementos de programación que van más allá de las consultas básicas: procedimientos almacenados, funciones, disparadores y cursores. Estos objetos se ejecutan directamente en el motor del SGBD, mejorando la eficiencia, seguridad y coherencia de los datos.

Los procedimientos almacenados son programas almacenados en la base de datos que implementan lógica de negocio compleja. Se invocan mediante la instrucción CALL y pueden tener parámetros de entrada/salida, permitiendo la reutilización del código y centralización de la lógica.

Las funciones almacenadas son procedimientos que devuelven un valor, permitiendo su uso directamente en consultas SELECT. A diferencia de los procedimientos, están limitadas a operaciones de lectura y ofrecen un mecanismo elegante para encapsular cálculos y transformaciones de datos.

Los disparadores (triggers) son bloques de código asociados a eventos de inserción, actualización o eliminación en una tabla. Se ejecutan automáticamente al ocurrir el evento, permitiendo implementar reglas de negocio complejas, validaciones contextuales y mantener integridad referencial sin dependencias de la aplicación cliente.

Los cursores actúan como iteradores sobre los resultados de una consulta SELECT, permitiendo procesar filas individuales en un bucle. Son útiles para operaciones complejas que requieren procesamiento fila a fila, aunque su uso debe ser cuidadoso debido a consideraciones de rendimiento en grandes volúmenes de datos.

Gracias

Universidad de Sevilla

Departamento de Lenguajes y Sistemas Informáticos