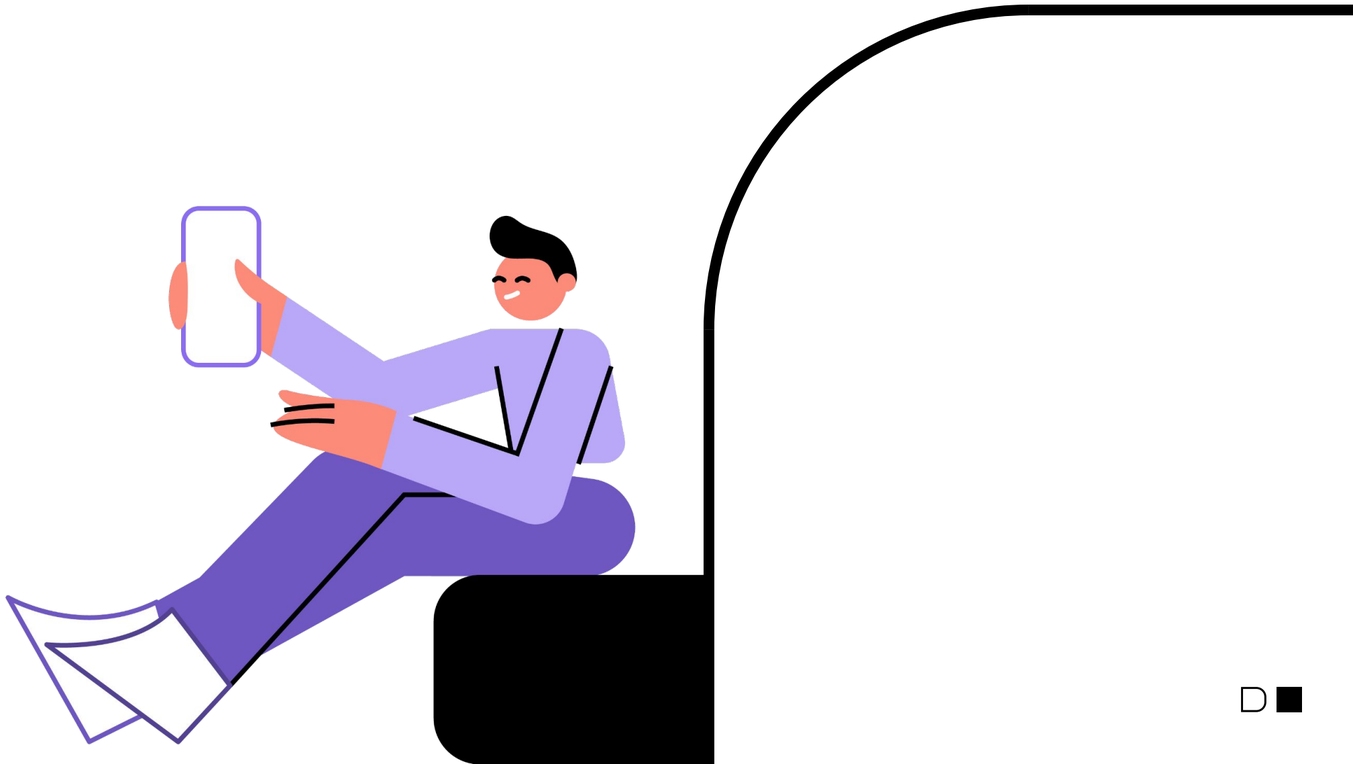
DIGITALHOUSE

Procedimientos almacenados

Certified Tech Developer
The Ultimate Degree

Índice



01

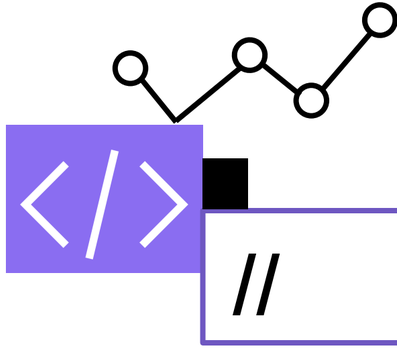
Concepto, estructura y definición

¿Qué es un stored procedure?

Los **SP** (stored procedure) son un conjunto de instrucciones en formato SQL que se almacenan, compilan y ejecutan dentro del servidor de bases de datos.

Pueden incluir parámetros de entrada y salida, devolver resultados tabulares o escalares, mensajes para el cliente e invocar instrucciones DDL y DML.

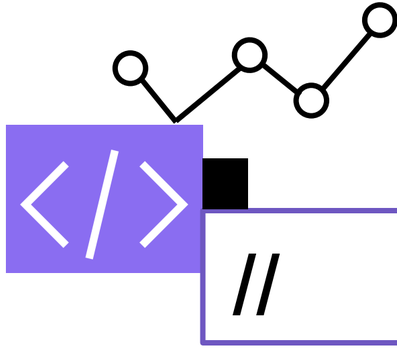
Por lo general, se los utiliza para **definir la lógica del negocio dentro de la base de datos** y reducir la necesidad de codificar dicha lógica en programas clientes.



Estructura de un stored procedure

- **DELIMITER**: se escribe esta cláusula seguida de una combinación de símbolos que no serán utilizados en el interior del SP.
- **CREATE PROCEDURE**: se escribe este comando seguido del nombre que identificará al SP.
- **BEGIN:** Esta cláusula se utiliza para indicar el inicio del código SQL.
- **Bloque de instrucciones SQL.**
- **END:** se escribe esta cláusula seguida de la combinación de símbolos definidos en DELIMITER y se utiliza para indicar el final del código SQL.

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



Definición de un stored procedure

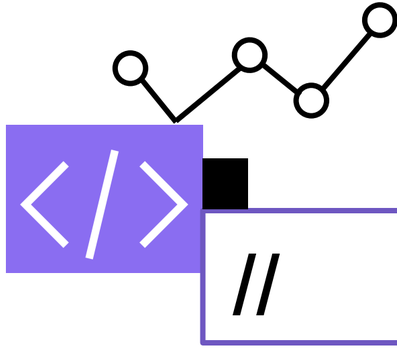
- **CREATE PROCEDURE:** crea un procedimiento almacenado.

```SQL
CREATE PROCEDURE sp_nombre_procedimiento()
```

- **DROP PROCEDURE:** elimina un procedimiento almacenado. Se requiere del privilegio de ALTER ROUTINE.

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

02 Variables



Declaración de variables

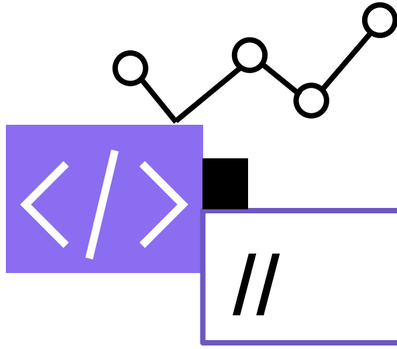
- Dentro de un **SP** se permite declarar variables que son elementos que almacenan datos que pueden ir cambiando a lo largo de la ejecución.
- La declaración de variables se coloca después de la cláusula BEGIN y antes del bloque de instrucciones SQL.
- Opcionalmente, se puede definir un valor inicial mediante la cláusula DEFAULT.

Sintaxis:

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

Ejemplo:

```SQL
DECLARE salario FLOAT DEFAULT 1000.00;
```



Asignación de valores a variables

- Para asignar un valor a una variable se utiliza la cláusula SET.
- Las variables solo pueden contener valores escalares. Es decir, un solo valor.

Sintaxis:

```sql
SET nombre_variable = expresión;
```

Ejemplo:

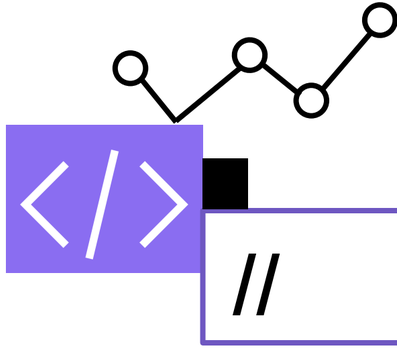
```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento()
BEGIN
     DECLARE salario FLOAT DEFAULT 1000.00;
     SET salario = 25700.50;
END $$
```

03 Parámetros

¿Qué es un parámetro?

- Los parámetros son variables por donde se envían y reciben datos de programas clientes.
- Se definen dentro de la cláusula CREATE.
- Los SP pueden tener uno, varios o ningún parámetro de entrada y asimismo, pueden tener uno, varios o ningún parámetro de salida.
- Existen tres tipos de parámetros:

Parámetro	Tipo	Función
IN	Entrada	Recibe datos
OUT	Salida	Devuelve datos
INOUT	Entrada-Salida	Recibe y devuelve datos



Declaración del parámetro IN

Es un parámetro de entrada de datos y se utiliza para recibir valores. Este parámetro viene definido por defecto cuando no se especifica su tipo.

Sintaxis:

```SQL
CREATE PROCEDURE sp_nombre_procedimiento(IN param1 TIPO_DE_DATO, IN param2 TIPO_DE_DATO);
```

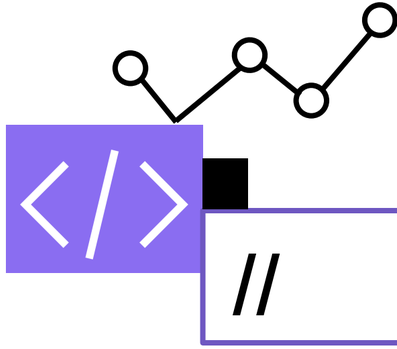
Ejemplo:

```SQL
DELIMITER $$

CREATE PROCEDURE sp_nombre_procedimiento(IN id_usuario INT)
BEGIN
     -- Bloque de instrucciones SQL;
END $$
```

Ejecución:

```SQL
CALL sp_nombre_procedimiento(11);
```



Declaración del parámetro OUT

Es un parámetro de salida de datos y se utiliza para devolver valores.

Sintaxis:

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

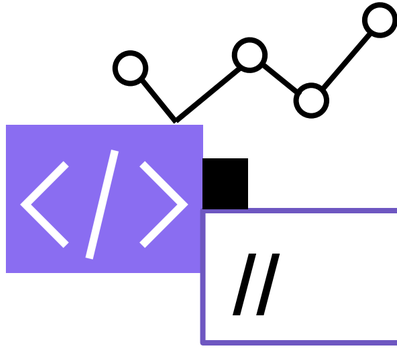
Ejemplo:

```sql
DELIMITER $$

CREATE PROCEDURE sp_nombre_procedimiento(OUT salario FLOAT)
BEGIN
    SET salario = 25700.50;
END $$
```

Ejecución:

```sql
CALL sp_nombre_procedimiento(@salario);
SELECT @salario; -- Bloque de instrucciones SQL
```



Declaración del parámetro INOUT

Es un mismo parámetro que se utiliza para la entrada y salida de datos. Puede recibir valores y devolver los resultados en la misma variable.

Sintaxis:

```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 (20.000)
SELECT @salario; -- Muestra el resultado
```

04 Bloque de sentencias

Bloque de sentencias

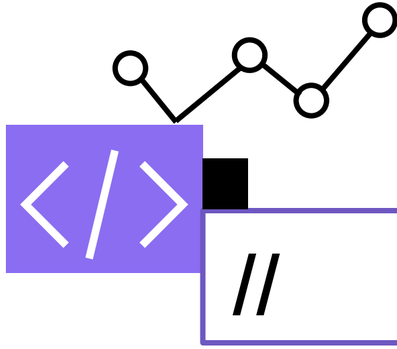
Dentro del bloque BEGIN...END, además de la definición de variables se pueden escribir distintas sentencias para nuestros procedimientos:

- Sentencias de control de flujo
- Cursores
- Manejo de errores

Sentencias de control de flujo

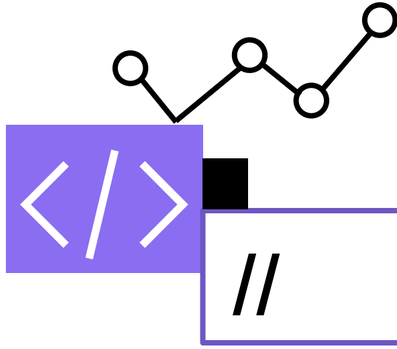
Cuando necesitamos aplicar condiciones y/o bucles en nuestro programa, MySQL nos brinda las siguientes sentencias:

1. **CASE**: se utiliza para condicionales complejas.
2. **IF:** se utiliza para condicionales simples.
3. **ITERATE:** vuelve a empezar una iteración. Solo se utiliza en Loop, Repeat y While.
4. **LEAVE**: se utiliza para salir de iteraciones.
5. **LOOP:** bucle que se ejecuta x cantidad de veces.
6. **REPEAT:** bucle que se ejecuta hasta que se cumple una condición.
7. **RETURN:** se utiliza para retornar un valor en una función o procedimiento.
8. **WHILE**: bucle que se ejecuta mientras se cumpla una condición.



Ejemplo 1: sentencias de control de flujo

```SQL
DELIMITER $$
CREATE FUNCTION sp_nombre_funcion(IN id_usuario INT) RETURNS INT
BEGIN
DECLARE v INT;
SET v=id_usuario;
CASE v
WHEN v=1 THEN SET v=2;
WHEN v=3 SET v=4;
END CASE;
nombreLoop1: LOOP
 SET v=v+1;
 IF v<=3 THEN ITERATE nombreLoop1;
 ELSEIF v>3 LEAVE nombreLoop1;
 END IF;
END LOOP nombreLoop1;
RETURN v;
END $$
```



Ejemplo 2: sentencias de control de flujo

```SQL
CREATE FUNCTION sp_nombre_funcion() RETURNS INT
BEGIN
DECLARE v INT;
SET v=1;
REPEAT
 SET v=v+1;
UNTIL v= 1000 END REPEAT;

WHILE v > 0 DO
 SET v= v-1;
END WHILE;

RETURN v;
END $$
```

Cursor

En el resultado de una consulta, ¿cómo podríamos realizar operaciones por cada uno de los registros del resultado?

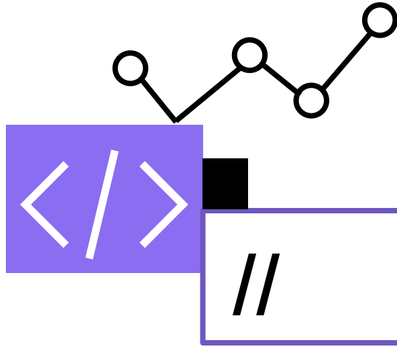
Para ello existen los cursores, en MySQL tenemos las siguientes sentencias para su definición y uso.

DECLARE: se utiliza para definir el cursor

OPEN: se utiliza para abrir el cursor.

FETCH: se utiliza para asignar el próximo valor del cursor a una variable.

CLOSE: se utiliza para cerrar un cursor.



Ejemplo de cursor

SQL

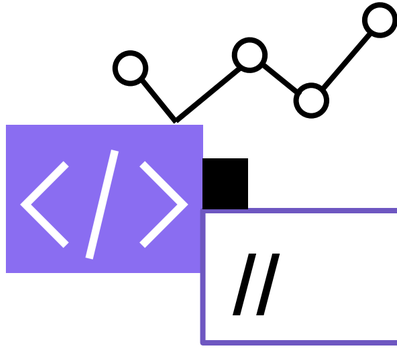
```sql
DELIMITER $$
CREATE PROCEDURE sp_nombre_procedimiento(IN id_usuario INT)
BEGIN
   DECLARE a INT, b INT
   DECLARE cur1 CURSOR FOR SELECT id FROM test.t1;
   OPEN cur1;
   sumarIds: LOOP
   FETCH cur1 INTO a;
      IF a > 10 THEN
         SET b = a + b;
      END IF;
      IF b > 100 THEN
         LEAVE sumarIds;
      END IF;
   END LOOP;
   CLOSE cur1;
END $$
```

Manejo de errores

Imaginemos que tenemos un error en uno de nuestro procedimiento. ¿Qué realizamos con este error? ¿Cómo lo vamos a tratar?

Para esto, MySQL nos permite crear Handlers y Conditions así podemos aplicar en nuestros programas un correcto manejo de errores.

- **Conditions:** son nombres que podemos agregar a los errores para dejar más legible nuestro código.
- **Handlers:** Es el manejador de errores. Por cada error que se ejecute, y si esa condición está definida en un handler, el código que definimos será ejecutado.



Ejemplo de manejo de errores

Conditions:

Si una condition no está definida, al leer nuestro código tenemos que saber de antemano que es el “error”.

```SQL
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
     -- Bloque de instrucciones SQL;
END
```

Si definimos la condition, al momento de leer el código nos hacemos una idea sobre de qué trata el error.
Luego sigue el código que definimos para Handler.

```SQL
DECLARE no_existe_tabla CONDITION FOR 1051
DECLARE CONTINUE HANDLER FOR no_existe_tabla
BEGIN
     -- Bloque de instrucciones SQL;
END
```

05

Ventajas y desventajas

Ventajas

del stored procedure

- **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 directa por parte de las aplicaciones clientes.
- **Independencia**: todo el código está dentro de la base de datos y no depende de archivos externos.
- **Reutilización del código**: se elimina la necesidad de escribir nuevamente un conjunto de instrucciones.
- **Mantenimiento más sencillo**: disminuye el costo de modificación cuando cambian las reglas de negocio.

Desventajas

del stored procedure

- **Difícil modificación:** si se requiere modificarlo, su definición tiene que ser reemplazada totalmente. En bases de datos muy complejas, la modificación puede afectar a las demás piezas de software que directa o indirectamente se refieran a este.
- **Aumentan el uso de la memoria:** Si usamos muchos procedimientos almacenados, el uso de la memoria de cada conexión que utiliza esos procedimientos se incrementará sustancialmente.
- **Restringidos para una lógica de negocios compleja:** en realidad, las construcciones de procedimientos almacenados no están diseñadas para desarrollar una lógica de negocios compleja y flexible.

DIGITALHOUSE

¡Muchas gracias!