



Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 4
Stored Procedures: Fundamentos

Stored Procedures

Los DBMs comerciales incluyen la posibilidad de crear procedimientos almacenados, que, como su nombre lo indica, son almacenados en la base de datos, **como parte del schema**.

Estos procedimientos pueden ser utilizados en queries SQL y otras sentencias a fin de realizar operaciones que no podríamos realizar solamente con SQL.


El estándar ANSI SQL define una especificación para **SQL/PSM** (*SQL Persistent Stored Modules*). En realidad, cada DBMS comercial ofrece su propia adaptación de PSM, y no siguen al pie de la letra el estándar.



Batches

SQL Server nos permite escribir –y ejecutar– código T-SQL directamente, sin que tengamos que crear necesariamente un stored procedure o función.

T-SQL define como batch a **un grupo de sentencias SQL enviadas al servidor a fin de que sean ejecutadas como un grupo**.

Cuando trabajamos con el Analizador de Consultas SQL, ejecutamos un batch cada vez que hacemos click en .

Dentro de un bloque de sentencias SQL se puede incluir la cláusula `GO` para indicar al programa cliente que envíe a procesar todas las sentencias anteriores al `GO` y continúe con el resto de las sentencias SQL luego de obtener los resultados del primer lote de sentencias.

`GO` no es un comando T-SQL. Es una keyword utilizada por aplicaciones cliente para separar batches.

La mayoría de los elementos de programación que veremos a continuación se pueden utilizar también desde batches T-SQL. (por supuesto, cuando hablamos de batches no tendremos ni parámetros ni valores de retorno).



PostgreSQL permite que escribamos código procedural fuera de una función a la manera de un batch de SQL Server. Estos bloques de código se denominan Anonymous Code Blocks. Los veremos más adelante. A lo largo de estas guías de estudio escribiremos código procedural usando funciones.

1. Parámetros

Los procedimientos pueden recibir parámetros.

Los **parámetros** –como en cualquier lenguaje de programación– nos permiten escribir procedimientos más generales. Los parámetros de salida generalmente retornan valores a otro procedure invocante o programa de aplicación cliente.

Un parámetro tiene un nombre, un tipo de dato y un **modo**. Este modo por lo general es **IN** o **OUT**.

2. Creación de procedures



En T-SQL

```
A CREATE PROC[EDURE] CambiarDomicilio
(
C @prmAu_lname VARCHAR(40), B
  @prmAddress VARCHAR(40) B
)
D AS
  UPDATE authors
    SET address = @prmAddress
    WHERE au_lname LIKE @prmAu_lname;
  RETURN
```

- Creamos un procedure con la sentencia `CREATE PROCEDURE`. Se puede usar la forma abreviada `CREATE PROC` (**A**).

- Si el procedure posee parámetros, los mismos se definen entre paréntesis (**B**). Los nombres de los parámetros son precedidos por @ (**C**) (al igual que todas las variables en T-SQL)

- El modo por omisión de un parámetro en T-SQL es IN, pero no lo explicitamos.

Si un parámetro posee modo OUT, lo indicamos con la keyword OUTPUT al final de la definición del parámetro. Por ejemplo:

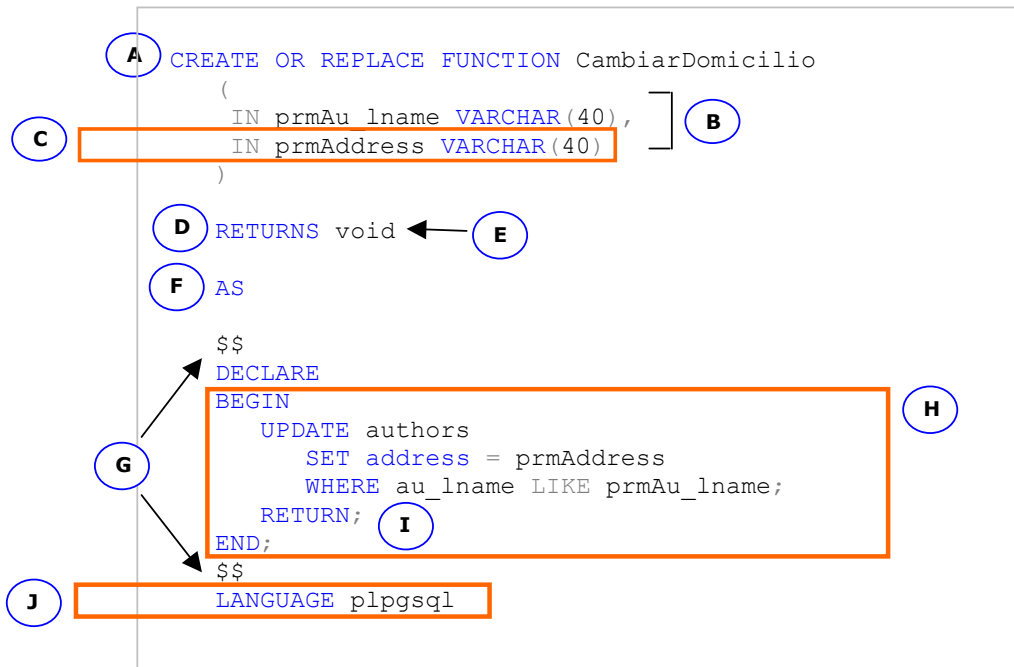
(@Nombre-Parametro Tipo-de-dato OUTPUT)

T-SQL no posee modo INOUT.

- Antes del cuerpo del procedimiento hay que agregar la keyword `AS` (**D**).



En PL/pgSQL definimos functions, no procedures.



- Creamos una function con la sentencia **CREATE FUNCTION (A)**. Cuando usamos la cláusula **OR REPLACE** PostgreSQL trata de crear la function y si la misma ya existe, automáticamente la sobrescribe con nuestro código.
- Si el procedure posee parámetros, los mismos se definen entre paréntesis **(B)**.
- El modo de los parámetros se indica al principio. Luego el nombre y por último el tipo **(C)**. Por omisión los parámetros son **IN**.
- La function debe especificar un valor de retorno. El tipo de ese valor de retorno se indica en la cláusula **RETURNS (D)**.
- Si necesitamos que la function se comporte como un procedure (como en este caso, que simplemente debe ejecutar una sentencia DML), debemos especificar **RETURNS void (E)**.

- Antes del cuerpo del procedimiento hay que agregar la keyword **AS** (F).
- \$\$ (G) son **dollar quotes** y permiten que dentro del bloque de la función se puedan usar comillas simples sin que haya necesidad de "escaparlas".
- Las keywords **BEGIN** y **END** (H) demarcan el cuerpo de la función.
- Si la function está definida con un a cláusula **RETURNS** void, podemos omitir la sentencia **RETURN**. También podemos consignar **RETURN**, como en este caso, sin un valor de retorno (I).
- Si necesitamos especificar un parámetro de salida especificamos:
(OUT parametro tipo-de-dato)
- PostgreSQL soporta varios lenguajes. Especificamos en que lenguaje está escrita la función a través de la keyword **LANGUAGE** (J). Nosotros utilizaremos el lenguaje `plpgsql`.

Nota

En versiones previas de PostgreSQL la cláusula **LANGUAGE** debe ubicarse luego de **RETURN** y antes de **AS**.

2.1. Parámetros opcionales

Los parámetros de entrada de los que hemos estado hablando son *requeridos*. En otras palabras, la ejecución falla si se omite alguno.

Para hacer que un parámetro de entrada sea opcional debemos especificar un valor *default* en el cuerpo del procedure, para que el mismo sea asumido en caso de omisión.



Los parámetros opcionales se declaran de la siguiente manera:

(@Nombre-Parametro Tipo-de-dato = Valor-por-omisión)



Especificamos un parámetro opcional de la siguiente manera:

(nombre-parametro tipo-de-dato DEFAULT valor-por-omisión)

En nuestro ejemplo:

```
CREATE OR REPLACE FUNCTION CambiarDomicilio
(
  IN prmAu_lname VARCHAR(40),
  IN prmAddress VARCHAR(40) DEFAULT 'NO ESPECIFICADO'
)
RETURNS void
...
```

3. Declaración de variables locales



En T-SQL las variables se pueden declarar en cualquier parte del cuerpo del procedimiento, luego de la keyword **AS**. Al igual que los nombres de parámetros, los nombres de variables son precedidos por @:

```
CREATE PROCEDURE CambiarDomicilio3
(
    @prmAu_lname VARCHAR(40),
    @prmAddress VARCHAR(40)
)
AS
    UPDATE authors
        SET address = @prmAddress
        WHERE au_lname LIKE @prmAu_lname;
    DECLARE @apellido VARCHAR(40)
    DECLARE @domicilio VARCHAR(40)
```

También se pueden agrupar varias declaraciones separadas por coma:

```
DECLARE
    @apellido VARCHAR(40),
    @domicilio VARCHAR(40)
```




En PostgreSQL las variables locales se declaran en una sección especial **DECLARE**. Cada declaración es una sentencia que finaliza con punto y coma:

```
CREATE OR REPLACE FUNCTION CambiarDomicilio3
(
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40) DEFAULT 'NO ESPECIFICADO'
)
RETURNS void

AS
$$

    DECLARE
        apellido VARCHAR(40);
        domicilio VARCHAR(40);

    BEGIN
        UPDATE authors
            SET address = prmAddress
            WHERE au_lname LIKE prmAu_lname;
        RETURN;
    END;
$$
LANGUAGE plpgsql
```

La sección **DECLARE** debe ubicarse antes del cuerpo de la función.

También podemos asignar un valor por omisión a estas variables (veremos la asignación más adelante en la Sección 6):

```
DECLARE
    apellido VARCHAR(40) := 'LOPEZ';
```

4. Ejecución



La sintaxis para ejecutar SPs es la siguiente:

```
[EXEC[UTE]] Nombre-SP [@Nombre-Parametro =] Valor-del-parametro [...]
```

La cláusula `EXECUTE` es opcional cuando la sentencia de ejecución del SP es la primer sentencia de un batch.



Ejecutamos una stored function invocándola como salida de una sentencia `SELECT`:

```
SELECT CambiarDomicilio ('Ringer', 'Colon 444');
```

4.1. Especificación de parámetros

Si el SP maneja más de un parámetro, estos pueden ser transferidos al mismo bajo dos modalidades **por posición** o **por nombre**.

Especificación por posición



```
EXECUTE <Nombre-SP>  
valor-de-parametro1,  
valor-de-parametro2,  
valor-de-parametro3,...
```



```
SELECT nombre-funcion (valor-de-parametro1, valor-de-parametro2,  
valor-de-parametro3,...);
```

En nuestro ejemplo:

```
SELECT CambiarDomicilio4 (  
    'Yokomoto',  
    '3 Silver Ct.'  
);
```

Especificación por nombre



Parámetros por nombre

```
EXECUTE <Nombre-SP>  
    @nombre-parametro = valor-de-parametro,  
    @nombre-parametro = valor-de-parametro, ...
```



Parámetros por nombre

```
SELECT nombre-funcion (nombre-Parametro1 := valor-de-parametro1,  
    nombre-Parametro2 := valor-de-parametro2);
```

En nuestro ejemplo:

```
SELECT CambiarDomicilio4 (  
    prmAu_lname := 'Yokomoto',  
    prmAddress := '3 r Ct.'  
);
```

Podemos especificar los parámetros por posición y documentar los mismos en los procedures con muchos parámetros. Por ejemplo:

T-SQL

```
EXECUTE <Nombre-SP>  
    Valor-de-Parametro1,    -- @Nombre-Parametro  
    Valor-de-Parametro2,    -- @Nombre-Parametro  
    Valor-de-Parametro3     -- @Nombre-Parametro
```

pl/pgSQL

```
SELECT CambiarDomicilio4 (  
    'Yokomoto',    --prmAu_lname  
    '3 Silver Ct.' --prmAddress  
);
```



Los parámetros de tipo `char` o `varchar` no necesitan comillas salvo que:

- Incluyan signos de puntuación.
- Consistan en una palabra reservada
- Incluyan solo números

PostgreSQL



Los tipos `char` y `varchar` deben especificarse con comillas simples.

Las fechas deben especificarse como strings entre comillas ya que incluyen el símbolo `" / "`.

5. Asignación



En T-SQL podemos asignar valores a variables usando `SET` o `SELECT` de manera indistinta:

```
CREATE PROCEDURE CambiarDomicilio5
(
    @prmAu_lname VARCHAR(40),
    @prmAddress VARCHAR(40)
)
AS
DECLARE
    @apellido VARCHAR(40),
    @domicilio VARCHAR(40)
SET @apellido = @prmAu_lname;
SELECT @domicilio = @prmAddress;
UPDATE authors
SET address = @domicilio
WHERE au_lname LIKE @apellido;
```

PostgreSQL



PostgreSQL usa el operador `:=` para la asignación:

```
CREATE OR REPLACE FUNCTION CambiarDomicilio5
(
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40) DEFAULT 'NO ESPECIFICADO'
)
RETURNS void
AS
$$
DECLARE
    apellido VARCHAR(40);
    domicilio VARCHAR(40);
BEGIN
    apellido := prmAu_lname;
    domicilio := prmAddress;

    UPDATE authors
    SET address = domicilio
    WHERE au_lname LIKE apellido;
    RETURN;
END;
$$
LANGUAGE plpgsql
```



Ámbito de vida de las variables en batches

En un batch T-SQL, las variables definidas **existen mientras existe el batch donde fueron definidas**.

Por ejemplo, si tenemos:

```
DECLARE @Mens varchar(40)
SET @Mens = 'Just testing...'
SELECT @Mens
GO
```

```
SELECT @Mens
GO
```

Estos son claramente dos batches.

La variable @Mens deja de existir luego de ejecutado el primer batch.

Por lo tanto, cuando se ejecuta el segundo batch está indefinida y obtenemos un error.

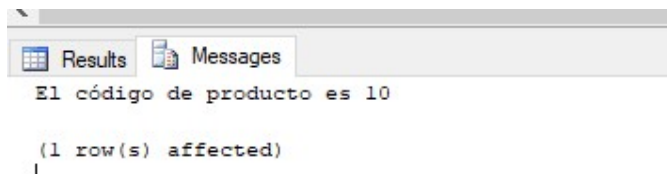
6. Mensajes informativos



Podemos retornar mensajes informativos al cliente a través de la sentencia `PRINT`. Su sintaxis es:

```
PRINT 'Mensaje'
```

```
ALTER PROCEDURE buscarPrecio
(
    @CodProd int,           -- Parametro de entrada
    @PrecUnit float OUTPUT  -- Parametro de salida
)
AS
    PRINT 'El código de producto es ' + CONVERT(VARCHAR, @CodProd)
    ...
    RETURN
```





Una sentencia PL/pgSQL similar a `PRINT` es `RAISE NOTICE`. Su sintaxis es la siguiente:

```
RAISE NOTICE 'No se encontro la publicación %', vTitle_id;
```

Como en el ejemplo, dentro de la string de salida se pueden especificar placeholders (símbolos `%`), cuyos valores deben ser especificados a continuación separados por coma.

Ejemplo

```
CREATE OR REPLACE FUNCTION PublicacionesPorType1
(
    IN prmType VARCHAR(40)
)
RETURNS NUMERIC
AS
$$
DECLARE
    cantidad NUMERIC;
    pattern VARCHAR(40);
BEGIN
    RAISE NOTICE 'Se recibió el tipo %', prmType;
    pattern := '%' || prmType || '%';
    RAISE NOTICE 'El patrón a buscar es %', pattern;

    SELECT COUNT(*) INTO cantidad
    FROM Titles
    WHERE type LIKE pattern;
    RETURN cantidad;
END;
$$
LANGUAGE plpgsql
```

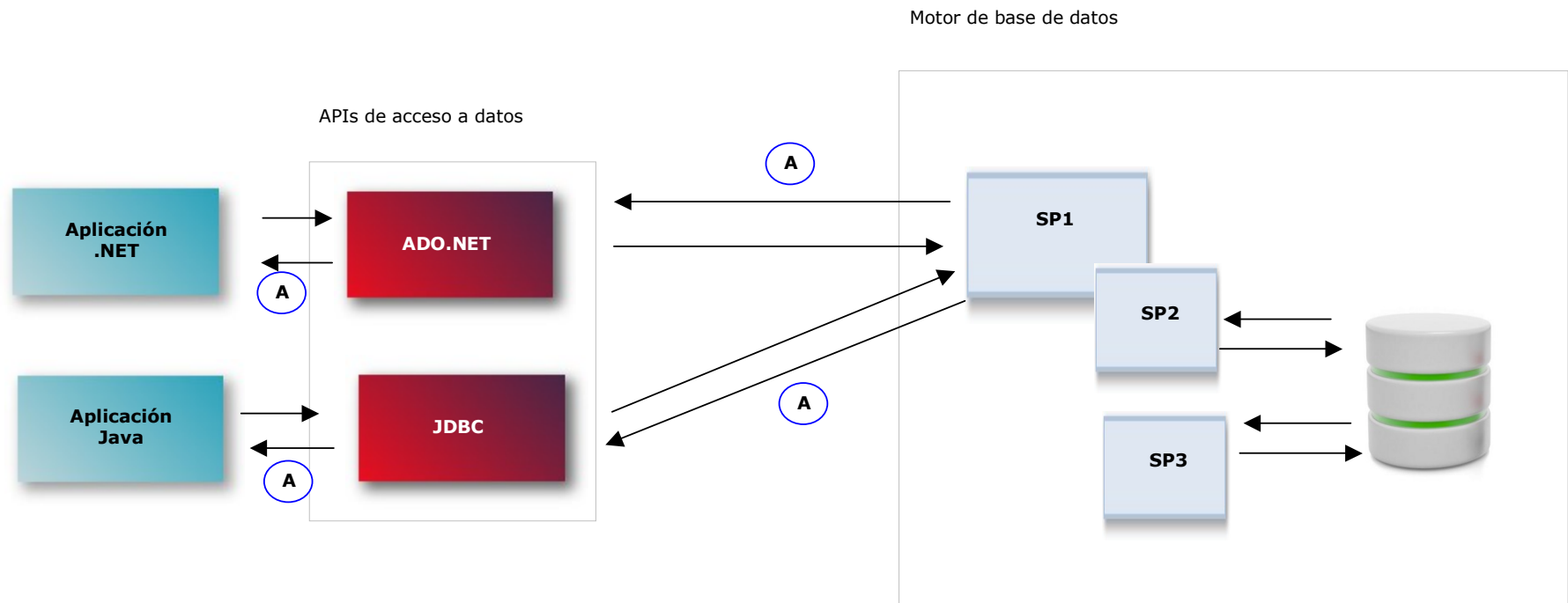
En la ventana Data Output:

Data Output	
publicacionesportype1	numeric
1	3

En la ventana Messages:

```
NOTICE: Se recibió el tipo popular_comp
NOTICE: El patrón a buscar es %popular_comp%

Successfully run. Total query runtime: 96 msec.
1 rows affected.
```



(A) puede contener:

- Recordset (resultset) (relaciones)
- Output parameters (incluye Output parameter)
- Errores

**Los mensajes no llegan al cliente.
Están disponibles en la consola de la Base de Datos**

7. Estructuras condicionales



En T-SQL los condicionales poseen la siguiente sintaxis:

```
IF <condicion>  
  BEGIN  
    ...  
  END  
[ELSE]  
  BEGIN  
    ...  
  END
```

A diagram shows a circle labeled 'A' with two arrows pointing to the `BEGIN` and `END` keywords of the first code block, indicating that the block is enclosed between them.

Cada lista de sentencias debe ser encerrada entre las keywords `BEGIN` y `END` (a menos que se trate de una única sentencia).



Bloques de código

Las keywords `BEGIN` y `END` delimitan en T-SQL *bloques de código*. Si no las especificamos, T-SQL ejecuta sólo la primer sentencia del grupo.

PostgreSQL En PL/SQL los condicionales poseen la siguiente forma:



```
IF vPrice < 100 THEN
    RETURN 'El precio es menor que 100';
ELSE
    RETURN 'El precio es mayor que 100';
END IF;
```

Como vemos, se especifica la cláusula **THEN** y el condicional finaliza con **END IF**.











No necesitamos definir **BEGIN** y **END** si poseemos más de una sentencia por bloque de ejecución.

Ejemplo

```
CREATE OR REPLACE FUNCTION MensajePrecio2
(
    IN prmtitle_id VARCHAR(6),
    IN valor NUMERIC
)
RETURNS text

AS
$$
DECLARE
    precio NUMERIC;
BEGIN
    precio := (SELECT price FROM titles WHERE title_id = prmtitle_id);
    IF precio < valor THEN
        RETURN 'El precio es menor que ' || valor::VARCHAR;
    ELSE
        RETURN 'El precio es mayor que ' || valor::VARCHAR;
    END IF;
END;
$$
LANGUAGE plpgsql

SELECT MensajePrecio2 ('BU1032', 100);
```

Data Output		Messages			Notifications		
     		  					
	mensajeprecio2						
	text						
1	El precio es menor que 100						

8. CASE

La sentencia `CASE` se puede incluir en cualquier ocasión en que necesitemos evaluar una condición, sin que necesariamente se trate de la salida de una sentencia `SELECT`.

Por ejemplo, en T-SQL:

```
SET @cad = 'El precio es ' + CASE
    WHEN @price < 10 THEN 'menor que 10 '
    WHEN @price = 10 THEN 'igual a 10'
    ELSE 'mayor a 10'
END;

SELECT @cad;
```

El mismo ejemplo en PostgreSQL

```
CREATE OR REPLACE FUNCTION MensajePrecio3
(
    IN prmttitle_id VARCHAR(6),
    IN valor NUMERIC
)
RETURNS text
AS
$$
DECLARE
    precio NUMERIC;
    cad text;
BEGIN
    precio := (SELECT price FROM titles WHERE title_id = prmttitle_id);
    cad := 'El precio es ' || CASE
        WHEN precio < valor THEN 'menor que ' ||
            valor::VARCHAR
        WHEN precio = valor THEN 'igual a ' || valor::VARCHAR
        ELSE 'mayor a ' || valor::VARCHAR
    END;

    RETURN cad;
END;
$$
LANGUAGE plpgsql

SELECT MensajePrecio3 ('BU1032', 100);
```

9. Estructuras repetitivas

9.1. Loop

PostgreSQL En PL/pgSQL definimos un `LOOP` de la siguiente manera:



```
cont:=0;
```

```
<<loopContador>>
```

A

```
LOOP
```

```
  EXIT loopContador WHEN cont = 5;
```

B

```
  cont := cont + 1;
```

```
END LOOP;
```

A. Podemos especificar labels, que se definen encerradas entre << y >>.

B. Se quiebra el bucle usando la construcción `EXIT . WHEN`.



T-SQL no posee sentencia `LOOP`.

9.2. Sentencia WHILE



En T-SQL la sintaxis de **WHILE** es:

```
WHILE <condicion>  
  BEGIN  
    ...  
  END
```

La sentencia **WHILE** posee dos cláusulas adicionales: **CONTINUE** salta el control al principio del bucle y vuelve a evaluar la condición del **WHILE**. **BREAK** efectúa una salida incondicional del bucle.

PostgreSQL



En PL/pgSQL definimos un **WHILE** de la siguiente manera:

```
cont:=0;  
WHILE cont < 5 LOOP  
  cont := cont + 1;  
END LOOP;
```

9.3. FOR

PostgreSQL



PL/pgSQL también posee una construcción **FOR** con la siguiente sintaxis:

```
suma:=0;  
FOR i IN 1..10 LOOP  
    suma := suma + i;  
END LOOP;
```

Esta sintaxis se puede extender para recorrer resultados de query como veremos más adelante.

10. Finalizar la ejecución de un batch



En T-SQL, La sentencia **RETURN** permite abandonar el batch de manera inmediata.

11. Trabajar con stored procedures

11.1. Modificar stored procedures



Para modificar un stored procedure podemos reemplazar la keyword `CREATE` por la keyword `ALTER`. Por ejemplo:

```
ALTER PROCEDURE CambiarDomicilio
(
    @prmAu_lname VARCHAR(40),
    @prmAddress VARCHAR(40)
)
AS
...
```



Cuando creamos una función tenemos la opción de agregar el modificador `OR REPLACE`:

```
CREATE OR REPLACE FUNCTION test()
....
```

11.2. Eliminar stored procedures



Para eliminar un stored procedure usamos la sentencia `DROP PROCEDURE`:

```
DROP PROCEDURE [EDURE] <Nombre-SP>
```



Para eliminar una function usamos la sentencia `DROP FUNCTION`.

```
DROP FUNCTION CambiarDomicilio
```

En versiones previas de PostgreSQL, si la función posee parámetros debemos especificar el tipo de los mismos. Por ejemplo:

```
DROP FUNCTION CambiarDomicilio (VARCHAR(40), VARCHAR(40));
```

12. Recursos del sistema



Variables del sistema

T-SQL posee una serie de variables del sistema, que tienen la característica de ser globales (también se las denomina variables globales T-SQL). Sus valores son establecidos automáticamente por el DBMS y son de solo lectura.

Las variables del sistema se diferencian de las locales en que su nombre es precedido por dos símbolos @.



Result Status

PostgreSQL posee un gran manejo de lo que llama **Result Status**. El comando `GET DIAGNOSTICS` permite recuperar en una variable determinado aspecto del resultado de una operación. El "aspecto" a evaluar se especifica a través de una keyword.

12.1. Cantidad de filas afectadas por una sentencia



La variable del sistema @@rowcount proporciona la cantidad de filas afectadas por la última sentencia ejecutada.

Las sentencias de control de flujo T-SQL como IF o WHILE restablecen esta variable a 0.



Podemos obtener la cantidad de filas afectadas por una operación usando GET DIAGNOSTICS con la keyword ROW_COUNT. En el siguiente ejemplo recuperamos en la variable vCantFilas la cantidad de filas recuperadas por un query.

```
PERFORM *  
  FROM titles;  
GET DIAGNOSTICS vCantFilas = ROW_COUNT;
```

Ejemplo:

```
CREATE OR REPLACE FUNCTION CantidadFilasAfectadas  
(  
  IN prmtype VARCHAR(30)  
)  
RETURNS INTEGER  
  
AS  
$$  
DECLARE  
  cantFilas INTEGER;  
BEGIN  
  PERFORM *  
    FROM titles  
    WHERE type = prmtype;  
  GET DIAGNOSTICS cantFilas = ROW_COUNT;  
  
  RETURN cantFilas;  
END;  
$$  
LANGUAGE plpgsql  
  
SELECT CantidadFilasAfectadas ('popular_comp');
```