Universidad Nacional del Litoral

Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 4
Persistent Stored Modules - Stored Procedures

Msc. Lic. Hugo Minni
2016

Persistent Stored Modules
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 **SQL/PSM** (*SQL Persistent Stored Modules*) (o simplemente **PSM**[1]), un lenguaje procedural para su uso en procedimientos almacenados.

PSM apareció por primera vez en 1996 como una extensión a SQL-92. Más adelante fue incorporado al estándar SQL:1999 y ha sido actualizado en SQL:2003.

Cada DBMS comercial ofrece su propia adptación de PSM. Para cada característica de PSM veremos la definición estándar y analizaremos las implementaciones T-SQL y PL/pgSQL de las mismas.

En general los stored procedures proporcionan:
- Mayor control sobre los datos.
- Acceso directo a operaciones complejas.
- Mejor performance.

[1] A lo largo de la materia usaremos la teminología ANSI (**PSM - Persistent Stored Module**) o la de los productos comerciales (**stored procedure – procedimiento almacenado- SP**) de manera indistinta.

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).

Batches

PostgreSQL no soporta que escribamos código procedural a la manera de un batch de SQL Server.
Si queremos escribir código procedural tenemos que hacerlo necesariamente dentro de una función.

1. Estructura de procedures y functions PSM

Los elementos principales de un procedimiento son:

```
CREATE PROCEDURE <nombre>
   (<parametros>)
   <local declarations>
   <procedure body>;
```

Donde los parámetros y la declaración de variables locales son elementos opcionales.

Una function es definida de manera muy similar, excepto que utilizan la keyword `FUNCTION`, y deben especificar un return-value. Los elementos de una función son:

```
CREATE FUNCTION <nombre>
   (<parametros>)
   RETURN <type>
   <local declarations>
   <procedure body>;
```

2. Parámetros

Los PSMs pueden recibir y retornar **parámetros**.

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

Los parámetros de un PSM procedure son "ternas" *mode-name-type*.
Es decir, el nombre del parámetro y su tipo de datos son precedidos por un "*modo*", que puede ser `IN`, `OUT` o `INOUT` que indican si el parámetro es **input-only**, **output-only** o ambas cosas: **input y output**. `IN` es el valor por omisión y puede ser omitido.

Los parámetros a funciones pueden ser solamente de modo `IN`. PSM prohíbe que una función modifique valores; la única manera de obtener una salida de una función es a través de su *return-value*.

Si un parámetro de entrada va a ser utilizado para definir –por ejemplo- una condición en una cláusula `WHERE`, el mismo debe coincidir en su tipo de dato con el tipo de dato de la columna a comparar.
En general la performance de los procedimientos es superior si el tipo de dato del parámetro es exactamente el mismo que el de la columna interviniente en la cláusula `WHERE`.

Los parámetros de salida son valores que un procedure retorna a otro procedure externo (llamado *outer procedure*) o a una aplicación cliente invocante.

Ejemplo 1

En el siguiente ejemplo tenemos un procedure PSM que toma el apellido de un autor y un nuevo domicilio como parámetros, y reemplaza el domicilio antiguo por el nuevo para el apellido proporcionado:

```
CREATE PROCEDURE CambiarDomicilio
   (
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40)
   )
   UPDATE authors
      SET address = prmAddress
      WHERE au_lname LIKE prmAu_lname;
```

3. Creación de procedures y functions

Definición del Stored Procedure T-SQL

Un stored procedure T-SQL posee algunas diferencias respecto al estándar ANSI:

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

- Se puede usar la forma abreviada `CREATE PROC` (**A**).

- Los nombres de los parámetros son precedidos por @ (**B**)

- **El *mode* del parámetro se ubica al final, no al principio** (**C**). Pero, además, no existe mode `IN`. Es el modo por omision en T-SQL, pero no puede explicitarse.

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

- Los parámetros de salida se especifican usando la keyword `OUTPUT`:
`(@Nombre-Parametro Tipo-de-dato OUTPUT)`

Definición de function

PostgreSQL solo soporta **funciones** PSM (no procedures), con algunas diferencias también respecto al estándar:

```
CREATE FUNCTION CambiarDomicilio
   (
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40)
   )
   RETURNS void
   LANGUAGE plpgsql
   AS
   $$
   BEGIN
      UPDATE authors
         SET address = prmAddress
         WHERE au_lname LIKE prmAu_lname;
      RETURN;
   END;
   $$;
```

A, B, C, D, E, F

- La function debe especificar un valor de retorno. Si queremos emular un procedure, debemos especificar `RETURNS void` (**A**).

- PostgreSQL soporta varios lenguajes. Especificamos en que lenguaje está escrita la función a través de la keyword `LANGUAGE` (**B**). Nosotros utilizaremos el lenguaje `plpgsql`.

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

- `$$` (**D**) 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` (**E**) 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 (**F**).

- La especificación de parámetros de salida sigue el estándar:
`(OUT parametro tipo-de-dato)`

3.1. Parámetros opcionales

Los parámetros de entrada de los que hemos estado hablando son *requeridos*. En otras palabras, la ejecución falla si se omite alguno. Es posible hacer que un parámetro de entrada sea opcional si se especifica un valor *default* en el cuerpo del procedure en el caso de su 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)
```

4. Declaración de variables locales

En PSM, declaramos una variable local a través de la sentencia:

```
DECLARE <nombre> <type>;
```

Una variable declarada de esta manera es **local**, y su valor no es preservado por el DBMS luego de la ejecución de la función o procedure.

Las declaraciones deben preceder al cuerpo de la función o procedure.

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 `DECLARE`. Cada declaración es una sentencia que finaliza con punto y coma:

```
CREATE FUNCTION CambiarDomicilio5
   (
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40)
   )
   RETURNS void
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
      apellido VARCHAR(40);
      domicilio VARCHAR(40);
  BEGIN
      UPDATE authors
         SET address = prmAddress
         WHERE au_lname LIKE prmAu_lname;
   END;
   $$;
```

Siguiendo el estándar PSM, la sección `DECLARE` debe ubicarse antes del cuerpo de la función.

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

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

5. 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');
```

5.1. 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**.

5.1.1. Especificación de parámetros por posición (Notación posicional)

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

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

5.1.2. Especificación de parámetros por nombre (Notación por Nombre)

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

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

La especificación de parámetros por nombre es mucho mejor desde el punto de vista de la claridad del código, sobre todo cuando el SP posee gran cantidad de parámetros. Sin embargo, en general la ejecución de los SP es más veloz cuando los parámetros son especificados por posición. Se puede obtener lo mejor de ambos mundos especificando los parámetros por posición y documentando las llamadas complejas. Por ejemplo, en T-SQL:

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

Los parámetros de tipo char o varchar no necesitan comillas salvo que:
- Incluyan signos de puntuación.
- Consistan en una palabra reservada
- Incluyan solo números

Los tipos char y varchar deben especificarse con comillas simples.

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

6. Asignación

Toda variable declarada y sin inicializar posee un valor NULL.

En PSM, le asignamos un valor usando la siguiente sintaxis:

```
SET <variable> = <expresion>
```

Excepto por la introducción de la keyword SET, la asignación en PSM es similar a la asignación en muchos lenguajes de programación. La expresión a la derecha del signo igual es evaluada y el valor obtenido pasa a ser el valor de la variable a la izquierda.
La expresión puede ser un query, siempre y cuando retorne un valor escalar.

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 no soporta la sentencia `SET`. En su lugar, se usa un operador de asignación:

```
CREATE FUNCTION CambiarDomicilio5
   (
    IN prmAu_lname VARCHAR(40),
    IN prmAddress VARCHAR(40)
   )
   RETURNS void
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
      apellido VARCHAR(40);
      domicilio VARCHAR(40);
  BEGIN
      apellido := prmAu_lname;
      domicilio := prmAddress;
      UPDATE authors
         SET address = domicilio
         WHERE au_lname LIKE apellido;

   END;
   $$;
```

Ejercicio 1. Considere la siguiente función PL/pgSQL:

```
CREATE FUNCTION test
   (
   )
   RETURNS ...
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
      apellido VARCHAR(40);
   BEGIN
      ...
   END;
   $$;
```

Complétela de manera tal que retorne un valor booleano indicando si la variable `apellido` ha sido o no inicializada.

Ámbito de vida de las variables en batches

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

Ejercicio 2. Ejecute el siguiente lote de sentencias T-SQL y analice los resultados:

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

SELECT @Mens
GO
```

7. Mensajes informacionales

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

```
PRINT 'Mensaje'
```

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.

8. Estructuras condicionales

La siguiente es la sintaxis para una estructura condicional en PSM:

```
IF <condicion>
   THEN
      <lista-de-sentencias>
   ELSE
      <lista-de-sentencias>
END IF;
```

La condición es cualquier expresión de valor booleano como la que puede aparecer en la cláusula `WHERE` de las sentencias SQL.

PSM no requiere definir `BEGIN` y `END` si poseemos más de una sentencia por bloque de ejecución.

En T-SQL no se especifica ni `THEN` ni `END IF`. Además, a diferencia de como especifica el estándar, la lista de sentencias debe ser encerrada entre las keywords `BEGIN` y `END` (a menos que se trate de una única sentencia):

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

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 sigue al pie de la letra la sintaxis PSM para condicionales:

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

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

Ejercicio 3. Escriba un batch T-SQL que informe al cliente con un mensaje si el precio de la publicación con title_id `'BU1111'` es menor, igual o mayor a $10.

Ejemplo 2

Supongamos que necesitamos escribir un procedimiento que reciba un código de editorial y retorne un valor TRUE si:
- No existen publicaciones para tal editorial.
o si
- Existe más de una publicación de esa editorial para el tipo 'business'
En cualquier otro caso debe retornar FALSE.

En un stored procedure T-SQL solo tenemos una posibilidad para tipo de dato del valor de retorno: INTEGER. Si quisieramos otro tipo de retorno lo tendríamos que implementar a través de parámetros de tipo OUTPUT. En este caso, definimos un valor de retorno 1 equiparable a un TRUE y un valor de retorno 0 equiparable a un FALSE:

```
CREATE PROCEDURE PublicacionesBusiness
   (@pub_id CHAR(4))
   AS
      DECLARE
         @type CHAR(12)
      SET @type = 'business';

      IF NOT EXISTS (SELECT *
                        FROM titles
                        WHERE pub_id = @pub_id)
         RETURN 1;
      ELSE
         IF 1 <= (SELECT COUNT(*)
                     FROM titles
                        WHERE pub_id = @pub_id AND type = @type)
            RETURN 1;
         ELSE
            RETURN 0;
         --END IF;
      --END IF;
```

La siguiente sería la solución PL/pgSQL:

```
CREATE OR REPLACE FUNCTION PublicacionesBusiness
   (pub_id1 CHAR(4))

   RETURNS BOOLEAN
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
         type1 CHAR(12);
   BEGIN
      type1 := 'business';
      IF NOT EXISTS (SELECT *
                        FROM titles
                        WHERE pub_id = pub_id1)
         THEN
            RETURN TRUE;
         ELSE
            IF 1 <= (SELECT COUNT(*)
                        FROM titles
                        WHERE pub_id = pub_id1 AND type = type1)
               THEN
                  RETURN TRUE;
               ELSE
                  RETURN FALSE;
            END IF;
      END IF;
   END;
   $$;
```

9. Expresión 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`.

La siguiente es la resolución del Ejercicio 3 reemplazando una sentencia `IF` por una sentencia `CASE`:

```
SELECT @price = price
      FROM titles
      WHERE title_id = 'BU1111';
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;
```

Ejercicio 4. Resuelva el Ejercicio 3 pero esta vez usando una función PL/pgSQL y la sentencia `CASE`.

10. Queries en PSM

Existen varias formas de usar queries *select-from-where* en PSM

10.1. Queries en condiciones

Los queries pueden ser usados en condiciones, o, en general, en cualquier lugar en que un subquery es permitido en SQL. En el Ejemplo 2 vimos dos ejemplos:

```
IF NOT EXISTS (SELECT *
                  FROM titles
                  WHERE pub_id = pub_id1)
   THEN
      ...
END IF


IF 1 <= (SELECT COUNT(*)
            FROM titles
            WHERE pub_id = pub_id1 AND type = type1)
   THEN
      ...
END IF
```

10.2. Queries que retornan un valor escalar

Los queries que retornan un único valor escalar se pueden usar al lado derecho de las sentencias de asignación.

```
SET @price = (SELECT price
                FROM titles
                WHERE title_id = @title_id);
```

```
price1 := (SELECT price
                FROM titles
                WHERE title_id = title_id1);
```

10.3. Sentencias SELECT Single-row

Sentencia SELECT single-row

En el contexto de PSM, denominamos **sentencia SELECT single-row** a una sentencia `SELECT` **que retorna una única tupla** y que posee una cláusula `INTO` que especifica las variables donde deben ser ubicados los componentes de es única tupla. Estas variables pueden ser variables locales o parámetros al procedure PSM.

El siguiente es un ejemplo de sentencia SELECT single-row:

```
DECLARE price1 FLOAT;
SELECT price INTO price1
   FROM titles
```

Los tipos de datos entre los componentes y las variables deben ser compatibles, de otra forma los datos pueden perder precisión o resultar truncados.

En T-SQL las sentencias **SELECT single-row** se escriben anteponiendo una variable y un signo "=" a la columna o expresión:

```
DECLARE
   @price FLOAT,
   @type CHAR(12)
SELECT @price = price, @type = type
   FROM titles
   WHERE title_id = @title_id;
```

PL/pgSQL sigue el estándar PSM:

```
DECLARE
      price1 FLOAT;
      type1 CHAR(12);
   BEGIN
      SELECT price, type INTO price1, type1
         FROM titles
         WHERE title_id = title_id1;
```

Si la sentencia `SELECT` recuperara más de una tupla, las variables asumirían el valor de la primer tupla recuperada.

Si la sentencia `SELECT` no recuperara ninguna tupla, las variables asumirían el valor `NULL`.

10.4. SELECT sobre expresiones

En T-SQL podemos consultar el valor de cualquier variable o expresión usando directamente la sentencia `SELECT`:

```
SELECT @<nombre-variable>
```

Esto es muy útil cuando estamos programando batches.

También, como veremos más adelante, T-SQL permite que la salida de un stored procedure sea una sentencia `SELECT`, así que tenemos la posibilidad de retornar cualquier valor que querramos disparando directamente un `SELECT` de este tipo.

PL/pgSQL no permite ejecutar una sentencia `SELECT` dentro de una función sin que hagamos algo con el resultado.
Tiene sentido, ya que no tenemos la posibilidad de una OUTPUT directa como la que proporciona T-SQL.

Algunas reminiscencias de lo que sería un `SELECT` sin salida lo tenemos en una sentencia llamada `PERFORM`.
`PERFORM` es una especie de `SELECT` que permite evaluar una sentencia (puede ser un query o cualquier expresión) descartando su resultado:

```
PERFORM cantFilas;
```

11. Updates

En T-SQL podemos asignar el valor de una variable en el contexto de una sentencia de actualización `UPDATE`. La variable almacena el valor **previo** a la actualización. La sintaxis es la siguiente:

```
UPDATE tabla
   SET columna = nuevo-valor,
   @variable = columna
   WHERE condicion
```

En PL/pgSQL podemos hacer algo similar usando el modificador `RETURNING`, pero obtenemos el valor **posterior** a la modificación:

```
UPDATE ventas
   SET cant = Cant + 100
   WHERE codvent = 1
   RETURNING cant INTO vCant
```

Ejercicio 5. Ejecute el siguiente batch T-SQL que actualiza la cantidad vendida en una fila de la tabla `Sales`. Consulte el valor de la variable `@Cant`.

```
DECLARE @Cant smallint
UPDATE sales
   SET qty = qty + 1,
   @cant = qty
   WHERE stor_id = '7067' AND
         title_id = 'PS2091'
```

12. Finalizar la ejecución de un batch

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

13. Labels

PL/pgSQL permite definir **labels**.
Una label permite otorgar un nombre a una parte particular de nuestro programa.
Las labels se definen encerradas entre << y >>
Las labels son muy útiles para dirigir el control dentro de bucles, como veremos más adelante:

```
<<loopContador>>
LOOP
   EXIT loopContador WHEN cont = 5;
   cont := cont + 1;
END LOOP;
```

14. Loops

14.1. Sentencia LOOP

La construcción básica de loop en PSM es:

```
LOOP
   <lista-de-sentencias>
END LOOP;
```

Frecuentemente esta sentencia se le agrega una etiqueta (label):

```
titlesLoop: LOOP
    <lista-de-sentencias>
END LOOP;
```

De esta manera es posible abandonar (break) el loop usando la sentencia:

```
LEAVE <etiqueta-de-loop>
```

T-SQL no posee sentencia LOOP.

En PL/pgSQL definimos el LOOP de la siguiente manera:

```
cont:=0;

<<loopContador>>          A
LOOP
   EXIT loopContador WHEN cont = 5;    B
   cont := cont + 1;
END LOOP;
```

A. Las labels se definen encerradas entre << y >>.
B. Se quiebra el bucle usando la consrucción EXIT..WHEN. PL/pgSQL no define sentencia LEAVE.

14.2. Sentencia WHILE

PSM también define loops `WHILE`:

```
WHILE <condicion> DO
   <lista-de-sentencias>
END 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.

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

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

Ejercicio 6. En un batch T-SQL defina una tabla `t1` con dos columnas: `ID` de tipo autoincremental, `FechaHora` de tipo `datetime` no nulo con un valor por omisión ajustado a la fecha y hora actual.
Implemente un bucle que inserte 100 filas a la tabla almacenando en la columna `FechaHora` la fecha y hora actual. Verifique los datos insertados.

14.3. Numeric FOR Loop

PL/pgSQL también posee un numeric `FOR` loop con la siguiente sintaxis:

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

Este tipo de bucles se puede extender para recorrer resultados de query como veremos más adelante.

15. Anchored declarations

Anchored declarations

Son declaraciones de variables en las que se “asocia” o “ancla” la declaración de una variable a un objeto de la base de datos.

Cuando asociamos de esta manera un tipo de dato estamos indicando que queremos setear el tipo de dato de nuestra variable al tipo de dato de una estructura de datos previamente definida, que puede ser una tabla de la base de datos, una columna de una tabla, o un TYPE predefinido.

15.1. Scalar anchoring

Scalar anchoring

Definen una variable en base a una columna de una tabla.

Se especifican a través del atributo %TYPE.

```
DECLARE
   price1 titles.price%TYPE;
```

15.2. Record anchoring
(tipo de dato para la tupla completa)

Record anchoring

Definen una estructura de record en base al formato de tupla de una tabla de la base de datos.

Se especifican anexando al nombre de la tabla el atributo `%ROWTYPE`:

```
DECLARE
   title titles%ROWTYPE;
```

T-SQL no soporta anchored declarations.

16. Salida de un stored procedure

En T-SQL, tenemos tres posibilidades para obtener una salida de un stored procedure:

Un return value
También llamado "status de retorno". El return value es un valor entero, y está pensado para proporcionar un "código de status" al programa invocante. Por omisión, la ejecución exitosa de un stored procedure T-SQL retorna 0. Analizaremos los return values en la Sección 18.

Un parámetro OUTPUT
T-SQL permite definir parámetros OUTPUT de cualquiera de los tipos de datos T-SQL.

La salida de una sentencia select-from-where
T-SQL permite retornar directamente la salida de una sentencia *select-from-where*

Por ejemplo:

```
CREATE PROCEDURE ListarTitles
AS
   SELECT * FROM titles


EXECUTE ListarTitles
```

	title_id	title	type	pub_id	price	advance	royalty	ytd_sales	notes
1	BU1032	The Busy Executive's Database Guide	business	1389	19.99	5000.00	10	4095	An overview of available database systems with e...
2	BU1111	Cooking with Computers: Surreptitious Balance Sh...	business	1389	11.95	5000.00	10	3876	Helpful hints on how to use your electronic resourc...
3	BU2075	You Can Combat Computer Stress!	business	0736	2.99	10125.00	24	18722	The latest medical and psychological techniques f...
4	BU7832	Straight Talk About Computers	business	1389	19.99	5000.00	10	4095	Annotated analysis of what computers can do for y...
5	MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19.99	0.00	12	2032	Favorite recipes for quick, easy, and elegant meals.
6	MC3021	The Gourmet Microwave	mod_cook	0877	2.99	15000.00	24	22246	Traditional French gourmet recipes adapted for mo...

Una función PL/pgSQL posee por supuesto -como todo función- un valor de retorno. Sin embargo, su semántica no es necesariamente equiparable a un "satus de retorno" T-SQL. (debería ser de tipo INTEGER y deberíamos darle ese significado).

PL/pgSQL tampoco permite el OUTPUT directo de una sentencia *select-from-where*. Para paliar la necesidad de retornar conjuntos de datos, PL/pgSQL permite a una función retornar diferentes tipos de **sets**. Los analizaremos a continuación.

16.1. Retornando sets de valores

PL/pgSQL permite definir el valor de retorno de una función como `setof <tipo-de-dato>`. Esto permite que una función retorne una especie de "lista" de valores.
Por un lado definimos el tipo de retorno de la función. Por ejemplo:

```
RETURNS setof FLOAT
```

Y en el cuerpo del procedimiento debemos utilizar una cláusula `RETURN` especial: `RETURN QUERY`. Por ejemplo:

```
CREATE FUNCTION test
   ()
   RETURNS setof date
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
      RETURN QUERY SELECT ord_date FROM Sales

   END
```

16.2. Retornando sets de tuplas completas

Si especificamos un valor de retorno como setof <nombre-de-tabla> la función podrá retornar cualquier conjunto de tuplas de la tabla especificada. Por ejemplo:

Por un lado definimos el tipo de retorno de la función:

```
RETURNS setof titles
```

Y en el cuerpo del procedimiento utillzamos RETURN QUERY. Por ejemplo:

```
CREATE FUNCTION test503()
   RETURNS setof titles
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
     RETURN QUERY SELECT * FROM titles;
   END
   $$;
```

```
SELECT *
   FROM test503()
   WHERE pub_id = '1389' ;
```

	title_id character varying(6)	title character varying(80)	type character(12)	pub_id character(4)	price numeric(10,2)	advance numeric(10,2)
1	PC9999	Net Etiquette	popular comp	1389		
2	BU1111	Cooking with Compu	business	1389	8.71	5000.00
3	BU7832	Straight Talk Abou	business	1389	14.57	5000.00
4	PC1035	But Is It User Fri	popular comp	1389	16.73	7000.00
5	BU1032	The Busy Executive	business	1389	14.57	5000.00
6	PC8888	Secrets of Silicon	popular cop	1389	16.00	8000.00
7	PC7777	Java Ingeniería We	popular comp	1389	10.00	

16.3. Retornando sets de proyecciones de tuplas

Si se necesita obtener un set de algunas columnas de un set de tuplas podemos definir un **composite-type** con la estructura deseada. Por ejemplo:

```
CREATE TYPE publisherCT
   AS (
       pub_id CHAR(4),
       totalPrice numeric
      );
```

Luego nuestra function es definida para retornar este composite-type. En el cuerpo del procedimiento también utillzamos `RETURN QUERY`:

```
CREATE OR REPLACE FUNCTION test502()
   RETURNS setof publisherCT
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN

     RETURN QUERY
        SELECT pub_id, SUM(price) AS totalPrice
           FROM titles
           WHERE price IS NOT NULL
           GROUP BY pub_id;

   END
   $$;
```

```
SELECT  *
   FROM test502();
```

	pub_id character(4)	totalprice numeric
1	1389	80.58
2	0736	36.16
3	0877	51.73

16.4. Retornar composite types

También podemos retornar un composite type pero sin que el mismo sea necesariamente resultado de un query.

Definimos un **composite-type** con la estructura deseada. Por ejemplo:

```
CREATE TYPE publisherCT
   AS (
       pub_id CHAR(4),
       totalPrice numeric
      );
```

Y luego:

```
CREATE OR REPLACE FUNCTION test520()
   RETURNS setof publisherCT
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
      fila publisherCT%rowtype;
   BEGIN
      fila.pub_id:='0736';
      fila.totalPrice := 240.00;
      RETURN NEXT fila;
   END
   $$;
```

A

B

C

Nuestra function es definida para retornar este composite-type (**A**).

Definimos una variable especial de tipo <composite-type>%rowtype (**B**).

En el cuerpo del procedimiento utilizamos una cláusula `RETURN NEXT` <tupla-del-composite-type> (**C**).

17. Trabajar con stored procedures

17.1. 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()
   ....
```

17.2. Eliminar stored procedures

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

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

Para eliminar un stored procedure usamos la sentencia `DROP FUNCTION`. Si la función posee parámetros debemos especificar el tipo de los mismos. Por ejemplo:

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

18. 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.

18.1. 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 usado `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;
```

La variable FOUND

PostgreSQL proporciona una variable especial llamada `FOUND` que nos indica si un sentencia SQL encontró o no tuplas como resultado de su ejecución. Por ejemplo:

```
SELECT price
   INTO vPrice
   FROM titles
   WHERE title_id = 'PC6565';

   IF NOT FOUND THEN
      RAISE NOTICE 'Publicación no encontrada';
   END IF;
```

18.2. Columnas autoincrementales

Si estamos utilizando columnas de tipo autoincremental, la variable del sistema `@@identity` nos proporciona - luego de realizada una operación `INSERT`- el valor insertado automáticamente por el DBMS para la misma.

Ya vimos el modificador `RETURNING` cuando obteníamos el valor posterior a un `UPDATE`. De manera similar, podemos obtener el último valor insertado correspondiente a una columna autoincremental usando `RETURNING` en un `INSERT`.

Por ejemplo, dada la tabla proveed:

```
CREATE TABLE proveed
  (
   codProv    SERIAL,
   razonSoc  varchar(30)   NOT NULL,
   dir        varchar(30)   NOT NULL
  )
```

Podemos insertar una fila y obtener el ID insertado:

```
INSERT
   INTO proveed
      (razonsoc, dir)
      VALUES ('TCM Motors', 'Pinar de Rocha 8844')
      RETURNING codprov INTO vUltimoId;
```

19. Status de retorno de un stored procedure

Como vimos en la Sección 14, el *status de retorno* en T-SQL se usa para descrbir el *estado* del procedimiento al finalizar el mismo. En un entorno cliente/servidor sirve para indicar al programa invocante el motivo por el cual el procedimiento finalizó su ejecución.
Bajo condiciones normales, un SP finaliza su ejecución cuando alcanza el final del código del mismo o cuando ejecuta una sentencia `RETURN`. Esta finalización normal retorna un status de `0`.

Microsoft reserva un bloque de números de `-1` a `-99` para identificar status de error. Solo los primeros 14 valores poseen significado:

Valor	Significado
-1	Falta objeto
-2	Error de tipo de dato
-3	El proceso fue elegido como víctima de deadlock.
-4	Error de permisos
-5	Error de sintaxis
-6	Error de usuario de miscelánea
-7	Error de recursos (sin espacio, por ejemplo)
-8	Problema interno no fatal.
-9	El sistema ha alcanzado su límite.
-10	Inconsistencia interna fatal.
-11	Inconsistencia interna fatal.
-12	Tabla o índice corrupto.
-13	Base de datos corrupta.
-14	Error de hardware.

El valor del status de retorno de un SP puede capturarse mediante la siguiente sintaxis:

```
EXECUTE <@Variable-local-de-tipo-Int> = <Nombre-SP>
   Valor-de-Parametro1,
   Valor-de-Parametro2,
   Valor-de-Parametro3
```

Como vimos en la Sección 14, en PL/pgSQL no existe el concepto de status de retorno.

Valores de retorno personalizados

Si un SP interno -invocado por otro SP- genera una determinada condición que consideramos un error para nuestra aplicación (un no cumplimiento de una regla de negocio, por ejemplo) el desarrollador puede notificar de esta situación al SP invocante retornando un código de error personalizado que el SP invocante pueda interpretar.

En T-SQL, estos códigos de error pueden ser cualquier valor de tipo `Int` fuera de los reservados (`-99` a `-1`) y se especifican a continuación de la(s) cláusula(s) `RETURN` en el SP invocado.

20. Procedimientos almacenados e INSERT

En la Guía de Ejercicios Nro. 2 se revisó un tipo de sentencia INSERT con la siguiente sintaxis:

```
INSERT <tabla-destino>
   SELECT *
      FROM <tabla-origen>
      WHERE <condicion>
```

en donde el lote de datos a insertar era obtenido de un *result set* resultado de un acceso a datos. Podemos hacer que este lote de datos pueda ser proporcionado también por un procedimiento:

```
INSERT <tabla-destino>
   EXECUTE Nombre-SP
```

```
INSERT INTO <tabla-destino>
   SELECT * FROM Nombre-Funcion();
```

21. SQL dinámico

Muchos motores de bases de datos permiten definir sentencias SQL a partir de una cadena de caracteres construida a partir de elementos variables, tales como variables locales o valores de retorno de procedimientos almacenados.

Ejecutamos una cadena de caracteres como una sentencia SQL a través del comando EXEC (o EXECUTE):

```
EXEC (@Nombre-de-variable)
```

En el siguiente ejemplo disparamos un SELECT sobre una tabla y columna de salida cuyos nombres conocemos en tiempo de ejecución:

```
DECLARE @Cad Varchar(100)
       DECLARE @nomTabla Varchar(30) = 'titles'
       DECLARE @nomColumna VARCHAR(30) = 'title_id'
SET @Cad = 'SELECT ' + @nomColumna;
SET @Cad = @Cad + ' FROM ' + @nomTabla;
EXEC (@Cad)
```

Ejecutamos la cadena de caracteres como una sentencia SQL a través del comando EXECUTE:

```
EXECUTE Nombre-de-variable
```

El siguiente es el mismo ejemplo de SQL dinámico en una función PL/pgSQL:

```
  ...
 DECLARE
     cad Varchar(100);
     nomTabla Varchar(30) := 'titles';
     nomColumna VARCHAR(30) := 'title_id';
  BEGIN

     cad := 'SELECT ' || nomColumna;
     cad = cad || ' FROM ' || nomTabla;
     RETURN QUERY EXECUTE cad;
  END
  ...
```

22. Demarcación de transacciones en SQL

Recordemos que una transacción es un grupo de sentencias que deben ejecutarse de manera exitosa como una unidad o fracasar también de manera completa.

En SQL podemos demarcar un conjunto de sentencias para indicar al DBMS que las mismas deben ejecutarse como una unidad.

La sentencia SQL estándar para marcar el inicio de una transacción es START TRANSACTION.

En T-SQL la sentencia es BEGIN TRANSACTION o su forma abreviada BEGIN TRAN

PostgreSQL soporta las sintaxis: START TRANSACTION, BEGIN TRANSACTION, BEGIN WORK o simplemente BEGIN.

Hay dos maneras de finalizar una transacción:

A. La sentencia SQL COMMIT provoca que la transacción finalice de manera satisfactoria. Cualquier cambio a la base de datos provocado por la o las sentencias SQL desde el inicio de la transacción es "hecho permanente" en la base de datos. Antes de que la sentencia COMMIT sea ejecutada, los cambios son **tentativos** y **pueden o no ser visibles a otras transacciones**.

En T-SQL también podemos escribir COMMIT TRANSACTION o su forma abreviada COMMIT TRAN

PostgreSQL también soporta las sintaxis `COMMIT TRANSACTION` y `COMMIT WORK`.

B. La sentencia SQL `ROLLBACK` provoca que la transacción sea abortada o terminada de manera no satisfactoria. Cualquier cambio llevado a cabo por las sentencias SQL involucradas en la transacción es deshecho y no se verá como permanente en la base de datos.

En T-SQL también podemos escribir `ROLLBACK TRANSACTION` o su forma abreviada `ROLLBACK TRAN`

PostgreSQL también soporta las sintaxis `ROLLBACK TRANSACTION` y `ROLLBACK WORK`.

22.1. Implicancias de las transacciones

- Toda sentencia SQL aislada disparada contra el DBMS desde una aplicación cliente representa en sí lo que se denomina una *transacción implícita*. Es decir, el inicio (`BEGIN`) y confirmación (`COMMIT`) de la transacción están implícitamente definidos en toda sentencia SQL.

- La solicitud de deshacer (`ROLLBACK`) o confirmar (`COMMIT`) una transacción no interfiere en absoluto con el flujo de control del batch o procedimiento. Por ejemplo, después de una sentencia `ROLLBACK` el procesamiento continúa normalmente con la próxima sentencia del batch o procedimiento.

- Las transacciones persisten **abiertas** hasta que son confirmadas o deshechas. Debemos tener en cuenta que, cuando una transacción está pendiente de confirmación, las *páginas de datos* correspondientes a las filas afectadas por la transacción resultan bloqueadas, impidiendo a otras conexiones acceder a esos datos. Es por ello que las transacciones deben agrupar solo las sentencias relacionadas lógicamente y –como regla general- **deben involucrar la menor cantidad posible de operaciones**. Debemos tener siempre presente que nuestra transacción, desde principio a fin, debe durar activa el menor tiempo posible.

Ejercicio 7. Cree un SP T-SQL (sp_ObtenerPrecio) sin parámetros de entrada que liste el precio de la publicación con código "PS2091". Ejecútelo.

Ejercicio 8. Reescriba en T-SQL el SP del **Ejercicio 7** a fin de que proporcione el precio de cualquier publicación para la cual se proporcione un código. (sp_ObtenerPrec2). Ejecútelo a fin de consultar el precio de la publicación PS1372.

Ejercicio 9. Cree una función PL/pgSQL (sp_VerVenta) que obtenga la fecha de venta para aquellas ventas para las que se especifique el código de almacén (stor_id), número de orden y cantidad. Ejecútela para los siguientes parámetros: código de almacén 7067, número de orden P2121 y cantidad 40, primero especificados por *posición* (documentados) y luego *por nombre*.

Ejercicio 10. Reescriba en T-SQL el SP del **Ejercicio 8** a fin de que la especificación del código de publicación sea opcional y establecido a NULL por omisión. (sp_ObtenerPrec3)
Ejecútelo proporcionando el código de publicación PS1372 y sin proporcionar parámetros.

Ejercicio 11. Reescriba en T-SQL el SP del **Ejercicio 10** de manera tal que –si el usuario omite el parámetro- se le notifique esta situación por medio de un mensaje informativo con la forma *"El SP sp_ObtenerPrec4 requiere del parámetro title_id"* y se finalice la ejecución del mismo.
Ejecútelo con y sin parámetros.

Ejercicio 12.

En SQL Server, considere las tablas creadas en la Guía de ejercicios Nro. 2:

productos

```
codprod      int          not null,
descr        varchar(30)  not null,
precUnit     float        not null,
Stock        smallint     not null
```

Detalle

```
CodDetalle   Int          not null,
NumPed       Int          not null,
CodProd      Int          Not Null
Cant         Int          Not Null,
PrecioTot    float        Null
```

Cargue el siguiente lote de prueba en la tabla de Productos:

```
(10, "Articulo 1", 50, 20)
(20, "Articulo 2", 70, 40)
```

El precio total de la tabla Detalle se calcula en función de la cantidad pedida de un producto y su precio unitario.
Si se necesita insertar un nuevo detalle de pedido en la tabla detalle, el precio total puede calcularse en función del precio unitario en la tabla Productos. Un SP (sp_BuscaPrecio) podría obtener el precio unitario y proveerlo como parámetro de salida a un SP invocante (sp_InsertaDetalle) a fin de que implemente finalmente el alta en la tabla detalle.

El siguiente código -por ejemplo- implementa el procedimiento que busca el precio del artículo pedido:

```
CREATE PROCEDURE sp_BuscarPrecio
   (@CodProd int,                  -- Parametro de entrada
    @PrecUnit money  OUTPUT)         -- Parametro de salida
   AS
      SELECT @PrecUnit = PrecUnit
         FROM Productos
         WHERE CodProd = @Codprod
      RETURN
```

A fin de ejecutar este SP interactivamente hace falta definir una variable que reciba el parámetro de salida retornado por sp_BuscarPrecio:

```
DECLARE  @PrecioObtenido MONEY
EXECUTE sp_BuscarPrecio 10, @PrecioObtenido OUTPUT
SELECT @PrecioObtenido "Par. de salida"
```

El procedimiento invocante (*outer procedure*) completo se lista a continuación:

```
CREATE PROCEDURE sp_InsertaDetalle
   (@CodDetalle Int,     -- Parametro de entrada a sp_InsertaDetalle
    @NumPed Int,         -- Parametro de entrada a sp_InsertaDetalle
    @CodProd int,        -- Parametro de entrada a sp_InsertaDetalle y al inner proc
    @Cant  Int)          -- Parametro de entrada a sp_InsertaDetalle
    AS
      DECLARE  @PrecioObtenido MONEY   --Parametro de salida del inner procedure
      EXECUTE sp_BuscarPrecio @CodProd, @PrecioObtenido OUTPUT
      INSERT Detalle Values(@CodDetalle, @NumPed, @CodProd, @Cant,
                            @Cant * @PrecioObtenido)
      If @@RowCount = 1
         PRINT 'Se inserto una fila'
      RETURN
```

Cree ambos procedimientos y ejecute sp_InsertaDetalle a fin de insertar el siguiente detalle:

```
CodDetalle    1540
NumPed        120
CodProd       10
Cant          2
```

Ejercicio 13. Ejecute nuevamente sp_InsertaDetalle con los datos del **Ejercicio 12** pero omitiendo el valor de Cantidad. Capture y muestre el status de retorno del SP.

Ejercicio 14.

Redefina en T-SQL el SP de inserción de detalles de pedidos a fin de que contemple la posibilidad de que no se encuentre el producto a pedir o que su precio sea NULL (aunque en este caso la definición de la tabla no lo permite).

```
CREATE PROCEDURE sp_BuscarPrecio2
   (@CodProd int,                  -- Parametro de entrada
    @PrecUnit money  OUTPUT)         -- Parametro de salida
   AS
      SELECT @PrecUnit = PrecUnit
         FROM Productos
         WHERE CodProd = @Codprod
      IF @@RowCount = 0
         RETURN 70            -- No se encontro el producto
       IF @PrecUnit IS NULL
         RETURN 71            -- El producto existe pero su precio es NULL
      RETURN 0                -- El producto existe y su precio no es NULL
```

```
CREATE PROCEDURE sp_InsertaDetalle2
   (@CodDetalle Int,     -- Parametro de entrada a sp_InsertaDetalle2
    @NumPed Int,         -- Parametro de entrada a sp_InsertaDetalle2
    @CodProd int,        -- Parametro de entrada a sp_InsertaDetalle2 y al inner proc
    @Cant  Int)          -- Parametro de entrada a sp_InsertaDetalle2
    AS
      DECLARE  @PrecioObtenido MONEY   --Parametro de salida del inner procedure
      DECLARE @StatusRetorno Int
      EXECUTE @StatusRetorno = sp_BuscarPrecio2 @CodProd, @PrecioObtenido OUTPUT
      IF @StatusRetorno != 0
         BEGIN
            IF @StatusRetorno = 70
               PRINT 'Codigo de producto inexistente'
            ELSE
               IF @StatusRetorno = 71
                  PRINT 'Codigo de producto inexistente'
               ELSE
                  PRINT 'Error en el SP sp_BuscarPrecio2'
                  RETURN 99
         END
      INSERT Detalle Values(@CodDetalle, @NumPed, @CodProd, @Cant,
                            @Cant * @PrecioObtenido)
      IF @@Error != 0
         RETURN 77
      If @@RowCount = 1
         PRINT 'Se insertó una fila'
      RETURN 0
```

Ejecute sp_InsertaDetalle2 a fin de insertar el siguiente detalle, correspondiente a un producto inexistente:

```
CodDetalle    1540
NumPed        120
CodProd       99
Cant          2
```

Ejercicio 15.

En PostgreSQL, cree en pubs una tabla -llamada Editoriales- análoga a la tabla Publishers en Pubs, pero únicamente con las columnas pub_id y pub_name y sin datos. Para la creación de la estructura puede utilizar un Select pub_id, pub_name into Editoriales para una condición que jamás sea verdadera.

Ejercicio 16. En PostgreSQL también, inserte en la tabla Editoriales –usando una función PL/pgSQL- todas las filas de la tabla Publishers en Pubs. Implemente la inserción a través del enfoque visto en la sección *18*.

Ejercicio 17. En T-SQL, y en base a las tablas Productos y Detalle creadas en la Guía de Trabajo Nro. 2, registre el pedido de cinco unidades del artículo con código 100. Para ello debe disminuir el stock en la tabla Productos e insertar los datos del pedido en la tabla Detalle (Código de detalle 1200, Número de pedido 1108). Tansaccione las operaciones de manera tal que tengan éxito o fracasen como una unidad.