Bases de Datos I

Disparadores y procedimientos almacenados en SQL

Lic. Carlos León González
Dra.C. Lucina García Hernández

Facultad de Matemática y Computación
Universidad de La Habana

28 de noviembre de 2023


Sql
Sql
Sql
Sql
Sql
Sql
Sql
Sql
Sql
Sql
Sql

Problema

Supongamos que existe una tabla llamada “Pedido” que almacena información sobre los clientes. Cada vez que se inserta un nuevo pedido en esta tabla, también se desea actualizar automáticamente la tabla “Inventario” para reflejar la disminución de existencias de los productos.

Problema

Supongamos que existe una tabla llamada “Pedido” que almacena información sobre los clientes. Cada vez que se inserta un nuevo pedido en esta tabla, también se desea actualizar automáticamente la tabla “Inventario” para reflejar la disminución de existencias de los productos.

Solución

1. Junto a la instrucción INSERT, mandarle al SGBD una instrucción UPDATE.

Solución

1. Junto a la instrucción INSERT, mandarle al SGBD una instrucción UPDATE.

   Esta propuesta no soluciona el requisito de ser una operación automática

Solución

1. Junto a la instrucción INSERT, mandarle al SGBD una instrucción UPDATE. No son operaciones automáticas, sino manuales.

2. Usar un disparador.

Características de un disparador (*trigger*)

- Es un procedimiento que se activa automáticamente sobre una tabla asociada previamente

Características de un disparador (*trigger*)

- Es un procedimiento que se activa automáticamente sobre una tabla asociada previamente
- Los posibles eventos que activan el *trigger* son aquellas operaciones que modifican el estado de la tabla, o sea:
  - INSERT
  - UPDATE
  - DELETE

Características de un disparador (*trigger*)

- Es un procedimiento que se activa automáticamente sobre una tabla asociada previamente
- Los posibles eventos que activan el *trigger* son aquellas operaciones que modifican el estado de la tabla, o sea:
  - INSERT
  - UPDATE
  - DELETE
- No requiere intervención humana o programática para ejecutarse y no se puede detener una vez activado

Características de un disparador (*trigger*)

- Es un procedimiento que se activa automáticamente sobre una tabla asociada previamente
- Los posibles eventos que activan el *trigger* son aquellas operaciones que modifican el estado de la tabla, o sea:
  - INSERT
  - UPDATE
  - DELETE
- No requiere intervención humana o programática para ejecutarse y no se puede detener una vez activado
- Se utiliza para garantizar el cumplimiento de ciertas reglas del negocio y modificar los valores de los atributos de forma dinámica

Ventajas de los *triggers*

Este recurso brindado por SQL, brinda al programador a:

- Realizar cambios en cascada en la base de datos

Ventajas de los *triggers*

Este recurso brindado por SQL, brinda al programador a:

- Realizar cambios en cascada en la base de datos
- Comprobar restricciones más complejas que las definidas en el CHECK

Ventajas de los *triggers*

Este recurso brindado por SQL, brinda al programador a:

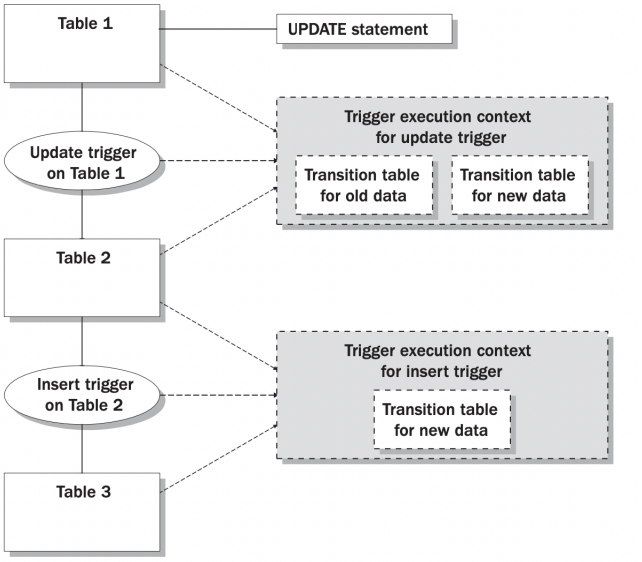
- Realizar cambios en cascada en la base de datos
- Comprobar restricciones más complejas que las definidas en el CHECK
- Evaluar el estado de una tabla antes y después de realizar una modificación de datos y actuar en función de la diferencia utilizando NEW and OLD

Ventajas de los *triggers*

Este recurso brindado por SQL, brinda al programador a:

- Realizar cambios en cascada en la base de datos
- Comprobar restricciones más complejas que las definidas en el CHECK
- Evaluar el estado de una tabla antes y después de realizar una modificación de datos y actuar en función de la diferencia utilizando NEW and OLD
- Tratar errores de manera más personalizada y compleja

Contexto de ejecución de dos *triggers*



Sintaxis de los *triggers*

```
01 | CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 | ON <table_name>
03 | FOR EACH ROW <trigger_statement>;
```

Sintaxis de los *triggers*

```
01 |  CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 |  ON <table_name>
03 |  FOR EACH ROW <trigger_statement>;
```

El campo `<trigger_order>` define el momento en que se ejecuta/activa un *trigger*:

- BEFORE
  Antes de ejecutarse la cláusula definida
- AFTER
  Posterior a la ejecución de la cláusula definida

Sintaxis de los *triggers*

```
01 | CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 | ON <table_name>
03 | FOR EACH ROW <trigger_statement>;
```

Sintaxis de los *triggers*

```
01 | CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 | ON <table_name>
03 | FOR EACH ROW <trigger_statement>;
```

El campo `<trigger_event>` establece el evento sobre el cual se define el *trigger*. Solo pueden definirse cláusulas que modifican el estado de los registros de la base de datos.

Sintaxis de los *triggers*

```
01 | CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 | ON <table_name>
03 | FOR EACH ROW <trigger_statement>;
```

El campo `<trigger_event>` establece el evento sobre el cual se define el *trigger*. Solo pueden definirse cláusulas que modifican el estado de los registros de la base de datos.

Evento	Acceso a la variable NEW	Acceso a la variable OLD
INSERT	Sí, representa la tupla a insertar	No
DELETE	No	Sí, representa la tupla antes de eliminarla
UPDATE	Sí, representa la tupla después de editarla	Sí, representa la tupla antes de editarla

Sintaxis de los *triggers*

```
01 | CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 | ON <table_name>
03 | FOR EACH ROW <trigger_statement>;
```

Sintaxis de los *triggers*

```
01 |  CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 |  ON <table_name>
03 |  FOR EACH ROW <trigger_statement>;
```

La expresión `<trigger_statement>` permite utilizar instrucciones de la programación imperativa y combinarla con las siguientes instrucciones declarativas:

- INSERT
- UPDATE
- DELETE
- CREATE
- DROP
- ALTER
- SELECT

Sintaxis de los *triggers*

```
01 |  CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 |  ON <table_name>
03 |  FOR EACH ROW <trigger_statement>;
```

La expresión `<trigger_statement>` permite utilizar instrucciones de la programación imperativa y combinarla con las siguientes instrucciones declarativas:

- INSERT
- UPDATE
- DELETE
- CREATE
- DROP
- ALTER
- SELECT

Estructura:

```
01 |  BEGIN
02 |  <body>
03 |  END;
```

Algunas instrucciones imperativas

- Declaración de variables

```
DECLARE <var_name> <var_type>
@<var_name>
```

Algunas instrucciones imperativas

- Declaración de variables
  ```
  DECLARE <var_name> <var_type>
  @<var_name>
  ```

- Asignación de valor
  ```
  SET <var_name> = <value>
  ```

Algunas instrucciones imperativas

- Declaración de variables

```
DECLARE <var_name> <var_type>
@<var_name>
```

- Asignación de valor

```
SET <var_name> = <value>
```

- Uso de condiciones

```
IF <conditional_expression>
THEN <statements_block>
ELSE <statements_block>
END IF
```

Algunas instrucciones imperativas

- Declaración de variables

```
DECLARE <var_name> <var_type>
@<var_name>
```

- Asignación de valor

```
SET <var_name> = <value>
```

- Uso de condiciones

```
IF <conditional_expression>
THEN <statements_block>
ELSE <statements_block>
END IF
```

- Uso de ciclos (WHILE / DO-WHILE)

```
WHILE <conditional_expression>
DO <statements_block>
END WHILE

REPEAT <statements_block>
UNTIL <conditional_expression>
END REPEAT
```

Algunas instrucciones imperativas

- Declaración de variables

```
DECLARE <var_name> <var_type>
@<var_name>
```

- Asignación de valor

```
SET <var_name> = <value>
```

- Uso de condiciones

```
IF <conditional_expression>
THEN <statements_block>
ELSE <statements_block>
END IF
```

- Uso de ciclos (WHILE / DO-WHILE)

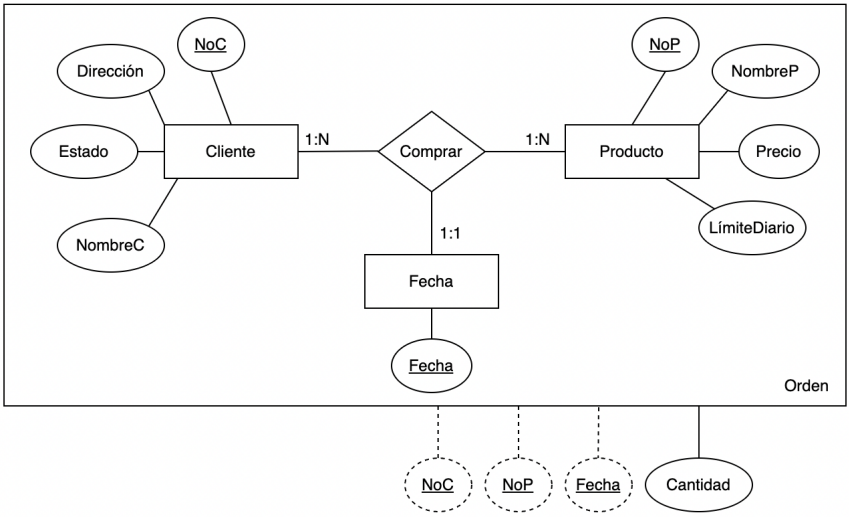
```
WHILE <conditional_expression>
DO <statements_block>
END WHILE

REPEAT <statements_block>
UNTIL <conditional_expression>
END REPEAT
```

- Detener ciclos (BREAK)

```
LEAVE
```

Recordando nuestra base de datos



Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER trg_limit_test <trigger_order> <trigger_event>
02 |  ON <table_name>
03 |  FOR EACH ROW <trigger_statement>;
```

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER trg_limit_test <trigger_order> <trigger_event>
02 |  ON <table_name>
03 |  FOR EACH ROW <trigger_statement>;
```

¿Sobre qué relación trabajará el *trigger*?

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 |  ON Orden
03 |  FOR EACH ROW <trigger_statement>;
```

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER <trigger_name> <trigger_order> <trigger_event>
02 |  ON Orden
03 |  FOR EACH ROW <trigger_statement>;
```

¿Con el uso de qué clásula se activará el *trigger*? ¿Cuándo lo hará?

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER <trigger_name> BEFORE INSERT
02 |  ON orden
03 |  FOR EACH ROW <trigger_statement>;
```

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER <trigger_name> BEFORE INSERT
02 |  ON orden
03 |  FOR EACH ROW <trigger_statement>;
```

¿Qué debe de hacer el *trigger*?

Ejemplo

Problema:
Construya un *trigger* para limitar la cantidad de unidades por producto, en una orden de compra. Cada producto tiene definido el máximo de compras por día.

Solución:

```
01 |  CREATE TRIGGER <trigger_name> BEFORE INSERT
02 |  ON Orden
03 |  FOR EACH ROW BEGIN
04 |
05 |    -- (1) Dado el cliente (NEW.NoC), obtener la cantidad de productos igual al
          comprado (NEW.NoP), adquiridos en la fecha actual (NEW.Fecha)
06 |
07 |    -- (2) Obtener el límite de productos del tipo NEW.NoP, que se puede
          adquirir en un mismo día
08 |
09 |    -- (3) Verificar si la cantidad de productos comprados, más la almacenada,
          supera la cantidad límite
10 |
11 |  END;
```

Ejemplo - Solución

```
01 | CREATE TRIGGER trg_limit_test BEFORE INSERT
02 | ON Orden
03 | FOR EACH ROW BEGIN
04 |   DECLARE total INT;
05 |   DECLARE limiteDiario INT;
06 |   -- (1)
07 |   SET total = (
08 |     SELECT SUM(Cantidad)
09 |     FROM orden
10 |     WHERE NoC = NEW.NoC AND NoP = NEW.NoP AND DATE(Fecha) = DATE(NEW.Fecha)
11 |   );
12 |   -- (2)
13 |   SET limiteDiario = (
14 |     SELECT MIN(producto.LimiteDiario)
15 |     FROM producto
16 |     WHERE producto.NoP = NEW.NoP
17 |   );
18 |   -- (3)
19 |   IF total + NEW.Cantidad > limiteDiario THEN
20 |     SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Acaparador!!!';
21 |   END IF;
22 | END;
```

Algunas restricciones de los *triggers*

- No recibe parámetros de entrada o salida

Algunas restricciones de los *triggers*

- No recibe parámetros de entrada o salida
- La única forma de trabajar con la fila “modificada” es a través de las pseudovariables (NEW y OLD)

Algunas restricciones de los *triggers*

- No recibe parámetros de entrada o salida
- La única forma de trabajar con la fila “modificada” es a través de las pseudovariables (NEW y OLD)
- No se puede ejecutar una operación de modificación sobre la misma tabla donde el *trigger* se define

Algunas restricciones de los *triggers*

- No recibe parámetros de entrada o salida
- La única forma de trabajar con la fila “modificada” es a través de las pseudovariables (NEW y OLD)
- No se puede ejecutar una operación de modificación sobre la misma tabla donde el *trigger* se define
- No se puede ejecutar una tarea sobre otra tabla, si la segunda tiene un *trigger* que afecte a la tabla del primer *trigger* en ejecución, o sea, no se acepta recursividad

Algunas restricciones de los *triggers*

- No recibe parámetros de entrada o salida
- La única forma de trabajar con la fila “modificada” es a través de las pseudovariables (NEW y OLD)
- No se puede ejecutar una operación de modificación sobre la misma tabla donde el *trigger* se define
- No se puede ejecutar una tarea sobre otra tabla, si la segunda tiene un *trigger* que afecte a la tabla del primer *trigger* en ejecución, o sea, no se acepta recursividad
- No se puede invocar procedimientos desde un *trigger*

Algo más que un *trigger* ...

¿Existe “algo” más general que un *trigger*?

Algo más que un *trigger* ...

¿Existe “algo” más general que un *trigger*?

Sí, se llama **procedimiento almacenado**.

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos
- Puede recibir y devolver parámetros

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos
- Puede recibir y devolver parámetros
- Utilizado, fundamentalmente, en la arquitectura cliente/servidor para el envío de parámetros para ejecutar funciones directamente sobre la base de datos

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos
- Puede recibir y devolver parámetros
- Utilizado, fundamentalmente, en la arquitectura cliente/servidor para el envío de parámetros para ejecutar funciones directamente sobre la base de datos
- No está asociada a una tabla en específico; puede manejar cualquier tabla, realizar operaciones sobre ella y realizar iteraciones de lectura/escritura

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos
- Puede recibir y devolver parámetros
- Utilizado, fundamentalmente, en la arquitectura cliente/servidor para el envío de parámetros para ejecutar funciones directamente sobre la base de datos
- No está asociada a una tabla en específico; puede manejar cualquier tabla, realizar operaciones sobre ella y realizar iteraciones de lectura/escritura
- Permite la recursividad, aunque no se recomienda

Más detalles de un procedimiento almacenado

- Es una función alojada en la base de datos
- Puede recibir y devolver parámetros
- Utilizado, fundamentalmente, en la arquitectura cliente/servidor para el envío de parámetros para ejecutar funciones directamente sobre la base de datos
- No está asociada a una tabla en específico; puede manejar cualquier tabla, realizar operaciones sobre ella y realizar iteraciones de lectura/escritura
- Permite la recursividad, aunque no se recomienda
- Se puede autorizar a un usuario para ejecutar procedimientos almacenados, aunque no tenga permiso de acceso directo a las tablas de la base de datos con la que trabaja el procedimiento

Sintaxis de un procedimiento almacenado

```
01 |  CREATE PROCEDURE <procedure_name> (<params>)
02 |  BEGIN
03 |    <instructions>
04 |  END
```

Sintaxis de un procedimiento almacenado

```
01 |  CREATE PROCEDURE <procedure_name> (<params>)
02 |  BEGIN
03 |    <instructions>
04 |  END
```

Los parámetros se especifican de la siguiente manera

```
01 |  IN <param_name> <param_type>
02 |  OUT <param_name> <param_type>
```

Sintaxis de un procedimiento almacenado

```
01 |  CREATE PROCEDURE <procedure_name> (<params>)
02 |  BEGIN
03 |    <instructions>
04 |  END
```

Los parámetros se especifican de la siguiente manera

```
01 |  IN <param_name> <param_type>
02 |  OUT <param_name> <param_type>
```

Los procedimientos se ejecutan de la forma

```
01 |  CALL <procedure_name> (<argument_list>)
```

Ejemplo

Construya un procedimiento para obtener el cliente con más compras efectuadas en de un intervalo de tiempo.

Ejemplo

Construya un procedimiento para obtener el cliente con más compras efectuadas en de un intervalo de tiempo.

```
01 | CREATE PROCEDURE ConsultarMayorCompradorEntreFechas(IN fechaInicio DATE, IN
       fechaFinal DATE, OUT resultado Char(50))
02 | BEGIN
03 |   SELECT NombreC INTO resultado
04 |   FROM orden
05 |     JOIN cliente ON cliente.NoC = orden.NoC
06 |     JOIN producto ON producto.NoP = orden.NoP
07 |   WHERE DATE(fecha) >= fechaInicio AND fecha <= fechaFinal
08 |   GROUP BY NombreC
09 |   ORDER BY SUM(Cantidad) DESC
10 |   LIMIT 1;
11 | END;
12 |
13 | CALL ConsultarMayorCompradorEntreFechas('2019-01-01 00:00:00', '2022-12-31
       23:59:59', @cliente);
14 | SELECT @cliente AS `Cliente`;
```

Ejercicio propuesto

Construya un procedimiento almacenado para computar el *n*-ésimo número de Fibonacci.

Ejercicio propuesto

Construya un procedimiento almacenado para computar el *n*-ésimo número de Fibonacci.

Tipos de posibles soluciones:

1. Iterativa
2. Recursiva

¿Dudas, comentarios, sugerencias?