Chương 5

THỦ TỤC LƯU TRỮ, HÀM VÀ TRIGGER

5.1 Thủ tục lưu trữ (stored procedure)

5.1.1 Các khái niệm

Như đã đề cập ở các chương trước, SQL được thiết kế và cài đặt như là một ngôn ngữ để thực hiện các thao tác trên cơ sở dữ liệu như tạo lập các cấu trúc trong cơ sở dữ liệu, bổ sung, cập nhật, xoá và truy vấn dữ liệu trong cơ sở dữ liệu. Các câu lệnh SQL được người sử dụng viết và yêu cầu hệ quản trị cơ sở dữ liệu thực hiện theo chếđộ tương tác.

Các câu lệnh SQL có thể được nhúng vào trong các ngôn ngữ lập trình, thông qua đó chuỗi các thao tác trên cơ sở dữ liệu được xác định và thực thi nhờ vào các câu lệnh, các cấu trúc điều khiển của bản thân ngôn ngữ lập trình được sử dụng.

Với thủ tục lưu trữ, một phần nào đó khả năng của ngôn ngữ lập trình được đưa vào trong ngôn ngữ SQL. Một thủ tục là một đối tượng trong cơ sở dữ liệu bao gồm một tập nhiều câu lệnh SQL được nhóm lại với nhau thành một nhóm với những khả năng sau:

- Các cấu trúc điều khiển (IF, WHILE, FOR) có thể được sử dụng trong thủ tục.
- Bên trong thủ tục lưu trữ có thể sử dụng các biến như trong ngôn ngữ lập trình nhằm lưu giữ các giá trị tính toán được, các giá trị được truy xuất được từ cơ sở dữ liệu.
- Một tập các câu lệnh SQL được kết hợp lại với nhau thành một khối lệnh bên trong một thủ tục. Một thủ tục có thể nhận các tham số truyền vào cũng như có thể trả về các giá trị thông qua các tham số (như trong các ngôn ngữ lập trình). Khi một thủ tục lưu trữ đã được định nghĩa, nó có thể được gọi thông qua tên thủ tục, nhận các tham số truyền vào, thực thi các câu lệnh SQL bên trong thủ tục và có thể trả về các giá trị sau khi thực hiện xong.

Sử dụng các thủ tục lưu trữ trong cơ sở dữ liệu sẽ giúp tăng hiệu năng của cơ sở dữ liệu, mang lại các lợi ích sau:

- Đơn giản hoá các thao tác trên cơ sở dữ liệu nhờ vào khả năng module hoá các thao tác này.

- Thủ tục lưu trữ được phân tích, tối ưu khi tạo ra nên việc thực thi chúng nhanh hơn nhiều so với việc phải thực hiện một tập rời rạc các câu lệnh SQL tương đương theo cách thông thường.
- Thủ tục lưu trữ cho phép chúng ta thực hiện cùng một yêu cầu bằng một câu lệnh đơn giản thay vì phải sử dụng nhiều dòng lệnh SQL. Điều này sẽ làm giảm thiểu sự lưu thông trên mạng.
- Thay vì cấp phát quyền trực tiếp cho người sử dụng trên các câu lệnh SQL và trên các đối tượng cơ sở dữ liệu, ta có thể cấp phát quyền cho người sử dụng thông qua các thủ tục lưu trữ, nhờ đó tăng khả năng bảo mật đối với hệ thống.

5.1.2 Tạo thủ tục lưu trữ

Thủ tục lưu trữ được tạo bởi câu lệnh CREATE PROCEDURE với cú pháp như sau:

```
CREATE PROCEDURE tên_thủ_tục [(danh_sách_tham_số)]
[WITH RECOMPILE|ENCRYPTION|RECOMPILE,ENCRYPTION]
AS
     Các_câu_lệnh_của_thủ_tục
```

Trong đó:

`tên_thủ_tục`	Tên của thủ tục cần tạo. Tên phải tuân theo qui tắc định danh và không được vượt quá 128 ký tự.
`danh_sách_tham_số`	Các tham số của thủ tục được khai báo ngay sau tên thủ tục và nếu thủ tục có nhiều tham số thì các khai báo phân cách nhau bởi dấu phẩy. Khai báo của mỗi một tham số tối thiểu phải bao gồm hai phần: • tên tham số được bắt đầu bởi dấu @. • kiểu dữ liệu của tham số **Ví dụ:** `@mamonhoc  nvarchar(10)`
`RECOMPILE`	Thông thường, thủ tục sẽ được phân tích, tối ưu và dịch sẵn ở lần gọi đầu tiên. Nếu tuỳ chọn WITH RECOMPILE được chỉ định, thủ tục sẽ được dịch lại mỗi khi được gọi.

ENCRYPTION	Thủ tục sẽ được mã hoá nếu tuỳ chọn WITH ENCRYPTION được chỉ định. Nếu thủ tục đã được mã hoá, ta không thể xem được nội dung của thủ tục.
các_câu_lệnh_của_thủ_tục	Tập hợp các câu lệnh sử dụng trong nội dung thủ tục. Các câu lệnh này có thể đặt trong cặp từ khóa BEGIN...END hoặc có thể không.

Ví dụ 5.1: Giả sử ta cần thực hiện một chuỗi các thao tác như sau trên cơ sở dữ liệu

1. Bổ sung thêm môn học *cơ sở dữ liệu* có mã *TI-005* và số đơn vị học trình là *5* vào bảng MONHOC
2. Lên danh sách nhập điểm thi môn *cơ sở dữ liệu* cho các sinh viên học lớp có mã *C24102* (tức là bổ sung thêm vào bảng DIEMTHI các bản ghi với cột MAMONHOC nhận giá trị *TI-005,* cột MASV nhận giá trị lần lượt là mã các sinh viên học lớp có mã *C24105* và các cột điểm là NULL).

Nếu thực hiện yêu cầu trên thông qua các câu lệnh SQL như thông thường, ta phải thực thi hai câu lệnh như sau:

```
INSERT INTO MONHOC
VALUES('TI-005','Cơ sở dữ liệu',5)

INSERT INTO DIEMTHI(MAMONHOC,MASV)
       SELECT 'TI-005',MASV
       FROM SINHVIEN
       WHERE MALOP='C24102'
```

Thay vì phải sử dụng hai câu lệnh như trên, ta có thể định nghĩa môt thủ tục lưu trữ với các tham số vào là *@mamonhoc*, *@tenmonhoc*, *@sodvht* và *@malop* như sau:

```
CREATE PROC sp_LenDanhSachDiem(
                          @mamonhoc      NVARCHAR(10),
                          @tenmonhoc     NVARCHAR(50),
                          @sodvht        SMALLINT,
                          @malop         NVARCHAR(10))
AS
     BEGIN
          INSERT INTO monhoc
          VALUES(@mamonhoc,@tenmonhoc,@sodvht)

          INSERT INTO diemthi(mamonhoc,masv)
```

```
                SELECT @mamonhoc,masv
                FROM sinhvien
                WHERE malop=@malop
        END
```

Khi thủ tục trên đã được tạo ra, ta có thể thực hiện được hai yêu cầu đặt ra ở trên một cách đơn giản thông qua lời gọi thủ tục:

```
sp_LenDanhSachDiem 'TI-005','Cơ sở dữ liệu',5,'C24102'
```

5.1.3 Lời gọi thủ tục lưu trữ

Như đã thấy ở ví dụ ở trên, khi một thủ tục lưu trữ đã được tạo ra, ta có thể yêu cầu hệ quản trị cơ sở dữ liệu thực thi thủ tục bằng lời gọi thủ tục có dạng:

```
tên_thủ_tục  [danh_sách_các_đối_số]
```

Số lượng các đối số cũng như thứ tự của chúng phải phù hợp với số lượng và thứ tự của các tham số khi định nghĩa thủ tục.

Trong trường hợp lời gọi thủ tục được thực hiện bên trong một thủ tục khác, bên trong một trigger hay kết hợp với các câu lệnh SQL khác, ta sử dụng cú pháp như sau:

```
EXECUTE tên_thủ_tục  [danh_sách_các_đối_số]
```

Thứ tự của các đối số được truyền cho thủ tục có thể không cần phải tuân theo thứ tự của các tham số như khi định nghĩa thủ tục nếu tất cả các đối số được viết dưới dạng:

```
@tên_tham_số  = giá_trị
```

Ví dụ 5.2: Lời gọi thủ tục ở ví dụ trên có thể viết như sau:

```
sp_LenDanhSachDiem  @malop='C24102',
                    @tenmonhoc='Cơ sở dữ liệu',
                    @mamonhoc='TI-005',
                    @sodvht=5
```

5.1.4 Sử dụng biến trong thủ tục

Ngoài những tham số được truyền cho thủ tục, bên trong thủ tục còn có thể sử dụng các biến nhằm lưu giữ các giá trị tính toán được hoặc truy xuất được từ cơ sở dữ liệu. Các biến trong thủ tục được khai báo bằng từ khóa DECLARE theo cú pháp như sau:

```
DECLARE  @tên_biến  kiểu_dữ_liệu
```

Tên biến phải bắt đầu bởi ký tự @ và tuân theo qui tắc về định danh. Ví dụ dưới đây minh hoạ việc sử dụng biến trong thủ tục

Ví dụ 5.3: Trong định nghĩa của thủ tục dưới đây sử dung các biến chứa các giá trị truy xuất được từ cơ sở dữ liệu.

```
CREATE PROCEDURE sp_Vidu(
          @malop1 NVARCHAR(10),
          @malop2 NVARCHAR(10))
AS
     DECLARE @tenlop1 NVARCHAR(30)
     DECLARE @namnhaphoc1 INT
     DECLARE @tenlop2 NVARCHAR(30)
     DECLARE @namnhaphoc2 INT

     SELECT @tenlop1=tenlop,
            @namnhaphoc1=namnhaphoc
     FROM lop WHERE malop=@malop1

     SELECT @tenlop2=tenlop,
            @namnhaphoc2=namnhaphoc
     FROM lop WHERE malop=@malop2

     PRINT @tenlop1+' nhap hoc nam '+str(@namnhaphoc1)
     print @tenlop2+' nhap hoc nam '+str(@namnhaphoc2)

     IF @namnhaphoc1=@namnhaphoc2
          PRINT 'Hai lớp nhập học cùng năm'
     ELSE
          PRINT 'Hai lớp nhập học khác năm'
```

5.1.5 Giá trị trả về của tham số trong thủ tục lưu trữ

Trong các ví dụ trước, nếu đối số truyền cho thủ tục khi có lời gọi đến thủ tục là biến, những thay đổi giá trị của biền trong thủ tục sẽ không được giữ lại khi kết thúc quá trình thực hiện thủ tục.

Ví dụ 5.4: Xét câu lệnh sau đây

```
CREATE PROCEDURE sp_Conghaiso(@a  INT,@b INT, @c INT)
AS
     SELECT @c=@a+@b
```

Nếu sau khi đã tạo thủ tục với câu lệnh trên, ta thực thi một tập các câu lệnh như sau:

```
DECLARE @tong INT
```

```
SELECT @tong=0
EXECUTE sp_Conghaiso 100,200,@tong
SELECT @tong
```

Câu lệnh "`SELECT @tong`" cuối cùng trong loạt các câu lệnh trên sẽ cho kết quả là: 0

Trong trường hợp cần phải giữ lại giá trị của đối số sau khi kết thúc thủ tục, ta phải khai báo tham số của thủ tục theo cú pháp như sau:

```
@tên_tham_số  kiểu_dữ_liệu  OUTPUT
```

hoặc:

```
@tên_tham_số  kiểu_dữ_liệu  OUT
```

và trong lời gọi thủ tục, sau đối số được truyền cho thủ tục, ta cũng phải chỉ định thêm từ khóa OUTPUT (hoặc OUT)

Ví dụ 5.5: Ta định nghĩa lại thủ tục ở ví dụ 5.4 như sau:

```
CREATE PROCEDURE sp_Conghaiso(
                                @a   INT,
                                @b   INT,
                                @c   INT OUTPUT)
AS
     SELECT @c=@a+@b
```

và thực hiện lời gọi thủ tục trong một tập các câu lệnh như sau:

```
DECLARE @tong INT
SELECT @tong=0
EXECUTE sp_Conghaiso 100,200,@tong OUTPUT
SELECT @tong
```

thì câu lệnh "`SELECT @tong`" sẽ cho kết quả là: 300

5.1.6 Tham số với giá trị mặc định

Các tham số được khai báo trong thủ tục có thể nhận các giá trị mặc định. Giá trị mặc định sẽ được gán cho tham số trong trường hợp không truyền đối số cho tham số khi có lời gọi đến thủ tục.

Tham số với giá trị mặc định được khai báo theo cú pháp như sau:

```
@tên_tham_số  kiểu_dữ_liệu  =  giá_trị_mặc_định
```

Ví dụ 5.6: Trong câu lệnh dưới đây:

```
CREATE PROC sp_TestDefault(
```

```
                    @tenlop NVARCHAR(30)=NULL,
                    @noisinh NVARCHAR(100)='HCM')
AS
     BEGIN IF @tenlop IS NULL
               SELECT hodem,ten
               FROM sinhvien INNER JOIN lop
                    ON sinhvien.malop=lop.malop
               WHERE noisinh=@noisinh
           ELSE
               SELECT hodem,ten
               FROM sinhvien INNER JOIN lop
                    ON sinhvien.malop=lop.malop
               WHERE noisinh=@noisinh AND
                     tenlop=@tenlop
     END
```

thủ tục *sp_TestDefault* được định nghĩa với tham số *@tenlop* có giá trị mặc định là *NULL* và tham số *@noisinh* có giá trị mặc định là *HCM*. Với thủ tục được định nghĩa như trên, ta có thể thực hiện các lời gọi với các mục đích khác nhau như sau:

- Cho biết họ tên của các sinh viên sinh tại *HCM*:
  ```
  sp_testdefault
  ```
- Cho biết họ tên của các sinh viên lớp *Tin K24* sinh tại *HCM*:
  ```
  sp_testdefault @tenlop='Tin K24'
  ```
- Cho biết họ tên của các sinh viên sinh tại *Long An*:
  ```
  sp_testDefault @noisinh=N'Long An'
  ```
- Cho biết họ tên của các sinh viên lớp *Tin K26* sinh tại *Cần Thơ*:
  ```
  sp_testdefault @tenlop='Tin K26',@noisinh='Cần Thơ'
  ```

5.1.7 Sửa đổi thủ tục

Khi một thủ tục đã được tạo ra, ta có thể tiến hành định nghĩa lại thủ tục đó bằng câu lệnh ALTER PROCEDURE có cú pháp như sau:

```
ALTER PROCEDURE tên_thủ_tục [(danh_sách_tham_số)]
[WITH RECOMPILE|ENCRYPTION|RECOMPILE,ENCRYPTION]
AS
     Các_câu_lệnh_Của_thủ_tục
```

Câu lệnh này sử dụng tương tự như câu lệnh CREATE PROCEDURE. Việc sửa đổi lại một thủ tục đã có không làm thay đổi đến các quyền đã cấp phát trên thủ tục cũng như không tác động đến các thủ tục khác hay trigger phụ thuộc vào thủ tục này.

5.1.8 Xoá thủ tục

Để xoá một thủ tục đã có, ta sử dụng câu lệnh DROP PROCEDURE với cú pháp như sau:

```
DROP PROCEDURE tên_thủ_tục
```

Khi xoá một thủ tục, tất cả các quyền đã cấp cho người sử dụng trên thủ tục đó cũng đồng thời bị xoá bỏ. Do đó, nếu tạo lại thủ tục, ta phải tiến hành cấp phát lại các quyền trên thủ tục đó.

5.2 Hàm do người dùng định nghĩa

Hàm là đối tượng cơ sở dữ liệu tương tự như thủ tục. Điểm khác biệt giữa hàm và thủ tục là hàm trả về một giá trị thông qua tên hàm còn thủ tục thì không. Điều này cho phép ta sử dụng hàm như là một thành phần của một biêu thức (chẳng hạn trong danh sách chọn của câu lệnh SELECT).

Ngoài những hàm do hệ quản trị cơ sở dữ liệu cung cấp sẵn, người sử dụng có thể định nghĩa thêm các hàm nhằm phục vụ cho mục đích riêng của mình.

5.2.1 Định nghĩa và sử dụng hàm

Hàm được định nghĩa thông qua câu lệnh CREATE FUNCTION với cú pháp như sau:

```
CREATE FUNCTION tên_hàm ([danh_sách_tham_số])
RETURNS (kiểu_trả_về_của_hàm)
AS
BEGIN
   các_câu_lệnh_của_hàm
END
```

Ví dụ 5.7: Câu lệnh dưới đây định nghĩa hàm tính ngày trong tuần (thứ trong tuần) của một giá trị kiểu ngày

```
CREATE FUNCTION thu(@ngay DATETIME)
RETURNS NVARCHAR(10)
AS
     BEGIN
```

```
        DECLARE @st NVARCHAR(10)
        SELECT @st=CASE DATEPART(DW,@ngay)
                        WHEN 1 THEN 'Chu nhật'
                        WHEN 2 THEN 'Thứ hai'
                        WHEN 3 THEN 'Thứ ba'
                        WHEN 4 THEN 'Thứ tư'
                        WHEN 5 THEN 'Thứ năm'
                        WHEN 6 THEN 'Thứ sáu'
                        ELSE 'Thứ bảy'
                    END
        RETURN (@st)  /* Trị trả về của hàm */
END
```

Một hàm khi đã được định nghĩa có thể được sử dụng như các hàm do hệ quản trị cơ sở dữ liệu cung cấp (thông thường trước tên hàm ta phải chỉ định thêm tên của người sở hữu hàm)

Ví dụ 5.8: Câu lệnh SELECT dưới đây sử dụng hàm đã được định nghĩa ở ví dụ trước:

```
SELECT masv,hodem,ten,dbo.thu(ngaysinh),ngaysinh
FROM sinhvien
WHERE malop='C24102'
```

có kết quả là:

MASV	HODEM	TEN		NGAYSINH
0241020001	Nguyễn Tuấn	Anh	Chủ nhật	1979-07-15 00:00:00
0241020002	Trần Thị Kim	Anh	Thứ năm	1982-11-04 00:00:00
0241020003	Võ Đức	Ân	Thứ hai	1982-05-24 00:00:00
0241020004	Nguyễn Công	Bình	Thứ tư	1979-06-06 00:00:00
0241020005	Nguyễn Thanh	Bình	Thứ bảy	1982-04-24 00:00:00
0241020006	Lê Thị Thanh	Châu	Thứ ba	1982-05-25 00:00:00
0241020007	Bùi Đình	Chiến	Thứ ba	1981-04-07 00:00:00
0241020008	Nguyễn Công	Chính	Chủ nhật	1981-11-01 00:00:00

5.2.2 Hàm với giá trị trả về là "dữ liệu kiểu bảng"

Ta đã biết được chức năng cũng như sự tiện lợi của việc sử dụng các khung nhìn trong cơ sở dữ liệu. Tuy nhiên, nếu cần phải sử dụng các tham số trong khung nhìn (chẳng hạn các tham số trong mệnh đề WHERE của câu lệnh SELECT) thì ta lại không thể thực hiện được. Điều này phần nào đó làm giảm tính linh hoạt trong việc sử dụng khung nhìn.

Ví dụ 5.9: Xét khung nhìn được định nghĩa như sau:

```
CREATE VIEW  sinhvien_k25
AS
     SELECT masv,hodem,ten,ngaysinh
     FROM sinhvien INNER JOIN lop
          ON sinhvien.malop=lop.malop
     WHERE khoa=25
```

với khung nhìn trên, thông qua câu lệnh:

```
SELECT * FROM sinhvien_K25
```

ta có thể biết được danh sách các sinh viên khóa 25 một cách dễ dàng nhưng rõ ràng không thể thông qua khung nhìn này để biết được danh sách sinh viên các khóa khác do không thể sử dụng điều kiện có dạng *KHOA = @thamso* trong mệnh đề WHERE của câu lệnh SELECT được.

Nhược điểm trên của khung nhìn có thể khắc phục bằng cách sử dụng hàm với giá trị trả về dưới dạng bảng và được gọi là *hàm nội tuyến* (inline function). Việc sử dụng hàm loại này cung cấp khả năng như khung nhìn nhưng cho phép chúng ta sử dụng được các tham số và nhờ đó tính linh hoạt sẽ cao hơn.

Một hàm nội tuyến được định nghĩa bởi câu lệnh CREATE TABLE với cú pháp như sau:

```
CREATE FUNCTION tên_hàm ([danh_sách_tham_số])
RETURNS TABLE
AS
     RETURN (câu_lệnh_select)
```

Cú pháp của hàm nội tuyến phải tuân theo các qui tắc sau:

- Kiểu trả về của hàm phải được chỉ định bởi mệnh đề RETURNS TABLE.
- Trong phần thân của hàm chỉ có duy nhất một câu lệnh RETURN xác định giá trị trả về của hàm thông qua duy nhất một câu lệnh SELECT. Ngoài ra, không sử dụng bất kỳ câu lệnh nào khác trong phần thân của hàm.

Ví dụ 5.10: Ta định nghĩa hàm *func_XemSV* như sau:

```
CREATE FUNCTION func_XemSV(@khoa SMALLINT)
RETURNS TABLE
AS
```

```
RETURN(SELECT masv,hodem,ten,ngaysinh
       FROM sinhvien INNER JOIN lop
            ON sinhvien.malop=lop.malop
       WHERE khoa=@khoa)
```

hàm trên nhận tham số đầu vào là khóa của sinh viên cần xem và giá trị trả về của hàm là tập các dòng dữ liệu cho biết thông tin về các sinh viên của khóa đó. Các hàm trả về giá trị dưới dạng bảng được sử dụng như là các bảng hay khung nhìn trong các câu lệnh SQL.

Với hàm được định nghĩa như trên, để biết danh sách các sinh viên khóa 25, ta sử dụng câu lệnh như sau:

```
SELECT * FROM dbo.func_XemSV(25)
```

còn câu lệnh dưới đây cho ta biết được danh sách sinh viên khóa 26

```
SELECT * FROM dbo.func_XemSV(26)
```

Đối với hàm nội tuyến, phần thân của hàm chỉ cho phép sự xuất hiện duy nhất của câu lệnh RETURN. Trong trường hợp cần phải sử dụng đến nhiều câu lệnh trong phần thân của hàm, ta sử dụng cú pháp như sau để định nghĩa hàm:

```
CREATE FUNCTION tên_hàm([danh_sách_tham_số])
RETURNS @biến_bảng TABLE định_nghĩa_bảng
AS
  BEGIN
    các_câu_lệnh_trong_thân_hàm
    RETURN
  END
```

Khi định nghĩa hàm dạng này cần lưu ý một số điểm sau:

- Cấu trúc của bảng trả về bởi hàm được xác định dựa vào định nghĩa của bảng trong mệnh đề RETURNS. Biến *@biến_bảng* trong mệnh đềRETURNS có phạm vi sử dụng trong hàm và được sử dụng như là một tên bảng.
- Câu lệnh RETURN trong thân hàm không chỉ định giá trị trả về. Giá trị trả về của hàm chính là các dòng dữ liệu trong bảng có tên là *@biếnbảng* được định nghĩa trong mệnh đề RETURNS

Cũng tương tự như hàm nội tuyến, dạng hàm này cũng được sử dụng trong các câu lệnh SQL với vai trò như bảng hay khung nhìn. Ví dụ dưới đây minh hoạ cách sử dụng dạng hàm này trong SQL.

Ví dụ 5.11: Ta định nghĩa hàm *func_TongSV* như sau:

```
CREATE FUNCTION Func_Tongsv(@khoa SMALLINT)
RETURNS @bangthongke TABLE
     (
          makhoa    NVARCHAR(5),
          tenkhoa   NVARCHAR(50),
          tongsosv  INT
     )
AS
     BEGIN
          IF @khoa=0
               INSERT INTO @bangthongke
               SELECT khoa.makhoa,tenkhoa,COUNT(masv)
               FROM (khoa INNER JOIN lop
                         ON khoa.makhoa=lop.makhoa)
                         INNER JOIN sinhvien
                         on lop.malop=sinhvien.malop
               GROUP BY khoa.makhoa,tenkhoa
          ELSE
               INSERT INTO @bangthongke
               SELECT khoa.makhoa,tenkhoa,COUNT(masv)
               FROM (khoa INNER JOIN lop
                         ON khoa.makhoa=lop.makhoa)
                         INNER JOIN sinhvien
                         ON lop.malop=sinhvien.malop
               WHERE khoa=@khoa
               GROUP BY khoa.makhoa,tenkhoa
          RETURN /*Trả kết quả về cho hàm*/
     END
```

Với hàm được định nghĩa như trên, câu lệnh:

```
SELECT * FROM dbo.func_TongSV(25)
```

Sẽ cho kết quả thống kê tổng số sinh viên khóa 25 của mỗi khoa:

MAKHOA	TENKHOA	TONGSOSV
DHTO1	Khoa Toán cơ - Tin học	5
DHTO2	Khoa Công nghệ thông tin	6
DHTO3	Khoa Vật lý	6
DHTO5	Khoa Sinh học	8

Còn câu lệnh:

```
SELECT * FROM dbo.func_TongSV(0)
```

Cho ta biết tổng số sinh viên hiện có (tất cả các khóa) của mỗi khoa:

MAKHOA	TENKHOA	TONGSOSV
DHTO1	Khoa Toán cơ - Tin học	15
DHTO2	Khoa Công nghệ thông tin	19
DHTO3	Khoa Vật lý	13
DHTO5	Khoa Sinh học	13

5.3 Trigger

Trong chương 4, ta đã biết các ràng buộc được sử dụng để đảm bảo tính toàn vẹn dữ liệu trong cơ sở dữ liệu. Một đối tượng khác cũng thường được sử dụng trong các cơ sở dữ liệu cũng với mục đích này là các trigger. Cũng tương tự như thủ tục lưu trữ, một trigger là một đối tượng chứa một tập các câu lệnh SQL và tập các câu lệnh này sẽ được thực thi khi trigger được gọi. Điểm khác biệt giữa thủ tục lưu trữ và trigger là: các thủ tục lưu trữ được thực thi khi người sử dụng có lời gọi đến chúng còn các trigger lại được "gọi" tự động khi xảy ra những giao tác làm thay đổi dữ liệu trong các bảng.

Mỗi một trigger được tạo ra và gắn liền với một bảng nào đó trong cơ sở dữ liệu. Khi dữ liệu trong bảng bị thay đổi (tức là khi bảng chịu tác động của các câu lệnh INSERT, UPDATE hay DELETE) thì trigger sẽ được tự động kích hoạt.

Sử dụng trigger một cách hợp lý trong cơ sở dữ liệu sẽ có tác động rất lớn trong việc tăng hiệu năng của cơ sở dữ liệu. Các trigger thực sự hữu dụng với những khả năng sau:

- Một trigger có thể nhận biết, ngăn chặn và huỷ bỏ được những thao tác làm thay đổi trái phép dữ liệu trong cơ sở dữ liệu.
- Các thao tác trên dữ liệu (xoá, cập nhật và bổ sung) có thể được trigger phát hiện ra và tự động thực hiện một loạt các thao tác khác trên cơ sở dữ liệu nhằm đảm bảo tính hợp lệ của dữ liệu.

- Thông qua trigger, ta có thể tạo và kiểm tra được những mối quan hệ phức tạp hơn giữa các bảng trong cơ sở dữ liệu mà bản thân các ràng buộc không thể thực hiện được.

5.3.1 Định nghĩa trigger

Một trigger là một đối tượng gắn liền với một bảng và được tự động kích hoạt khi xảy ra những giao tác làm thay đổi dữ liệu trong bảng. Định nghĩa một trigger bao gồm các yếu tố sau:

- Trigger sẽ được áp dụng đối với bảng nào?
- Trigger được kích hoạt khi câu lệnh nào được thực thi trên bảng: INSERT, UPDATE, DELETE?
- Trigger sẽ làm gì khi được kích hoạt?

Câu lệnh CREATE TRIGGER được sử dụng để đinh nghĩa trigger và có cú pháp như sau:

```
CREATE TRIGGER tên_trigger
ON tên_bảng
FOR {[INSERT][,][UPDATE][,][DELETE]}
AS
   các_câu_lệnh_của_trigger
```

Ví dụ 5.12: Ta định nghĩa các bảng như sau:

Bảng MATHANG lưu trữ dữ liệu về các mặt hàng:

```
CREATE TABLE mathang
(
mahang    NVARCHAR(5)    PRIMARY KEY,   /*mã hàng*/
tenhang   NVARCHAR(50)   NOT NULL,      /*tên hàng*/
soluong   INT,      /*số lượng hàng hiện có*/
)
```

Bảng NHATKYBANHANG lưu trữ thông tin về các lần bán hàng

```
CREATE TABLE   nhatkybanhang
(
stt       INT  IDENTITY  PRIMARY KEY,
ngay      DATETIME,      /*ngày bán hàng*/
```

```
nguoimua  NVARCHAR(30),  /*tên người mua hàng*/
mahang    NVARCHAR(5)    /*mã mặt hàng được bán*/
          FOREIGN KEY REFERENCES mathang(mahang),
soluong   INT,           /*giá bán hàng*/
giaban    MONEY          /*số lượng hàng được bán*/
)
```

Câu lệnh dưới đây định nghĩa trigger *trg_nhatkybanhang_insert*. Trigger này có chức năng tự động giảm số lượng hàng hiện có khi một mặt hàng nào đó được bán.

```
CREATE TRIGGER trg_nhatkybanhang_insert
ON nhatkybanhang
FOR INSERT
AS
   UPDATE mathang
   SET mathang.soluong=mathang.soluong-inserted.soluong
   FROM mathang INNER JOIN inserted
                ON mathang.mahang= inserted.mahang
```

Với trigger vừa tạo ở trên, nếu dữ liệu trong bảng MATHANG là:

MAHANG	TENHANG	SOLUONG
H1	Xà phòng	30
H2	Kem đánh răng	45

thì sau khi ta thực hiện câu lệnh:

```
INSERT INTO nhatkybanhang (ngay,nguoimua,mahang,soluong,giaban)
VALUES('5/5/2023','Tran Ngoc Thanh','H1',10,5200)
```

dữ liệu trong bảng MATHANG sẽ như sau:

MAHANG	TENHANG	SOLUONG
H1	Xà phòng	20
H2	Kem đánh răng	45

Trong câu lệnh CREATE TRIGGER ở ví dụ trên, sau mệnh đề ON là tên của bảng mà trigger cần tạo sẽ tác động đến. Mệnh đề tiếp theo chỉ định câu lệnh sẽ kích hoạt trigger (FOR INSERT). Ngoài INSERT, ta còn có thể chỉ định UPDATE hoặc DELETE cho mệnh đề này, hoặc có thể kết hợp chúng lại với nhau. Phần thân của trigger nằm sau từ khóa AS bao gồm các câu lệnh mà trigger sẽ thực thi khi được kích hoạt.

Chuẩn SQL định nghĩa hai bảng logic INSERTED và DELETED để sử dụng trong các trigger. Cấu trúc của hai bảng này tương tự như cấu trúc của bảng mà trigger tác động. Dữ liệu trong hai bảng này tuỳ thuộc vào câu lệnh tác động lên bảng làm kích hoạt trigger; cụ thể trong các trường hợp sau:

- Khi câu lệnh DELETE được thực thi trên bảng, các dòng dữ liệu bị xoá sẽ được sao chép vào trong bảng DELETED. Bảng INSERTED trong trường hợp này không có dữ liệu.
- Dữ liệu trong bảng INSERTED sẽ là dòng dữ liệu được bổ sung vào bảng gây nên sự kích hoạt đối với trigger bằng câu lệnh INSERT. Bảng DELETED trong trường hợp này không có dữ liệu.
- Khi câu lệnh UPDATE được thực thi trên bảng, các dòng dữ liệu cũ chịu sự tác động của câu lệnh sẽ được sao chép vào bảng DELETED, còn trong bảng INSERTED sẽ là các dòng sau khi đã được cập nhật.

5.3.2 Sử dụng mệnh đề IF UPDATE trong trigger

Thay vì chỉ định một trigger được kích hoạt trên một bảng, ta có thể chỉ định trigger được kích hoạt và thực hiện những thao tác cụ thể khi việc thay đổi dữ liệu chỉ liên quan đến một số cột nhất định nào đó của cột. Trong trường hợp này, ta sử dụng mệnh đề IF UPDATE trong trigger. IF UPDATE không sử dụng được đối với câu lệnh DELETE.

Ví dụ 5.13: Xét lại ví dụ với hai bảng MATHANG và NHATKYBANHANG, trigger dưới đây được kích hoạt khi ta tiến hành cập nhật cột SOLUONG cho một bản ghi của bảng NHATKYBANHANG (lưu ý là chỉ cập nhật đúng một bản ghi)

```
CREATE TRIGGER trg_nhatkybanhang_update_soluong
ON nhatkybanhang
FOR UPDATE
AS
IF UPDATE(soluong)
  UPDATE mathang
  SET mathang.soluong = mathang.soluong –
       (inserted.soluong-deleted.soluong)
  FROM (deleted INNER JOIN inserted ON
        deleted.stt = inserted.stt) INNER JOIN mathang
        ON mathang.mahang = deleted.mahang
```

Với trigger ở ví dụ trên, câu lệnh:

```
UPDATE nhatkybanhang
SET soluong=soluong+20
WHERE stt=1
```

sẽ kích hoạt trigger ứng với mệnh đề IF UPDATE (soluong) và câu lệnh UPDATE trong trigger sẽ được thực thi. Tuy nhiên câu lệnh:

```
UPDATE nhatkybanhang
SET nguoimua='Mai Hữu Toàn'
WHERE stt=3
```

lại không kích hoạt trigger này.

Mệnh đề IF UPDATE có thể xuất hiện nhiều lần trong phần thân của trigger. Khi đó, mệnh đề IF UPDATE nào đúng thì phần câu lệnh của mệnh đề đó sẽ được thực thi khi trigger được kích hoạt.

Ví dụ 5.14: Giả sử ta định nghĩa bảng R như sau:

```
CREATE TABLE R
(
     A    INT,
     B    INT,
     C    INT
)
```

và trigger *trg_R_update* cho bảng R:

```
CREATE TRIGGER trg_R_test
ON R
FOR UPDATE
AS
     IF UPDATE(A)
          Print 'A updated'
     IF UPDATE(C)
          Print 'C updated'
```

Câu lệnh:

```
UPDATE R SET A=100 WHERE A=1
```

sẽ kích hoạt trigger và cho kết quả là:

```
A updated
```

và câu lệnh:

```
UPDATE R SET C=100 WHERE C=2
```

cũng kích hoạt trigger và cho kết quả là:

```
C updated
```

còn câu lệnh:

```
UPDATE R SET B=100 WHERE B=3
```

hiển nhiên sẽ không kích hoạt trigger

5.3.3 ROLLBACK TRANSACTION và trigger

Một trigger có khả năng nhận biết được sự thay đổi về mặt dữ liệu trên bảng dữ liệu, từ đó có thể phát hiện và huỷ bỏ những thao tác không đảm bảo tính toàn vẹn dữ liệu. Trong một trigger, để huỷ bỏ tác dụng của câu lệnh làm kích hoạt trigger, ta sử dụng câu lệnh[1]:

```
ROLLBACK TRANSACTION
```

Ví dụ 5.15: Nếu trên bảng MATHANG, ta tạo một trigger như sau:

```
CREATE TRIGGER trg_mathang_delete
ON mathang
FOR DELETE
AS
     ROLLBACK TRANSACTION
```

Thì câu lệnh DELETE sẽ không thể có tác dụng đối với bảng MATHANG. Hay nói cách khác, ta không thể xoá được dữ liệu trong bảng.

Ví dụ 5.16: Trigger dưới đây được kích hoạt khi câu lệnh INSERT được sử dụng để bổ sung một bản ghi mới cho bảng NHATKYBANHANG. Trong trigger này kiểm tra điều kiện hợp lệ của dữ liệu là số lượng hàng bán ra phải nhỏ hơn hoặc bằng số lượng hàng hiện có. Nếu điều kiện này không thoả mãn thì huỷ bỏ thao tác bổ sung dữ liệu.

```
CREATE TRIGGER trg_nhatkybanhang_insert
ON NHATKYBANHANG
FOR INSERT
AS
   DECLARE @sl_co int  /* Số lượng hàng hiện có */
   DECLARE @sl_ban int /* Số lượng hàng được bán */
   DECLARE @mahang nvarchar(5) /* Mã hàng được bán */

   SELECT @mahang=mahang,@sl_ban=soluong
   FROM inserted

   SELECT @sl_co = soluong
   FROM mathang where mahang=@mahang

   /*Nếu số lượng hàng hiện có nhỏ hơn số lượng bán
     thì huỷ bỏ thao tác bổ sung dữ liệu */
```

[1] Cách sử dụng và ý nghĩa của câu lệnh ROLLBACK TRANSACTION được bàn luận chi tiết ở chương 6.

```
IF @sl_co<@sl_ban
        ROLLBACK TRANSACTION
/* Nếu dữ liệu hợp lệ
  thì giảm số lượng hàng hiện có */
ELSE
          UPDATE mathang
          SET soluong=soluong-@sl_ban
          WHERE mahang=@mahang
```

5.3.4 Sử dụng trigger trong trường hợp câu lệnh INSERT, UPDATE và DELETE có tác động đến nhiều dòng dữ liệu

Trong các ví dụ trước, các trigger chỉ thực sự hoạt động đúng mục đích khi các câu lệnh kích hoạt trigger chỉ có tác dụng đối với đúng một dòng dữ liêu. Ta có thể nhận thấy là câu lệnh UPDATE và DELETE thường có tác dụng trên nhiều dòng, câu lệnh INSERT mặc dù ít rơi vào trường hợp này nhưng không phải là không gặp; đó là khi ta sử dụng câu lệnh có dạng INSERT INTO ... SELECT ... Vậy làm thế nào để trigger hoạt động đúng trong trường hợp những câu lệnh có tác động lên nhiều dòng dữliệu?

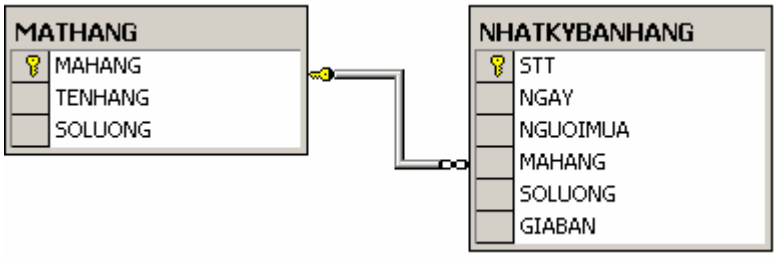
Có hai giải pháp có thể sử dụng đối với vấn đề này:

- Sử dụng truy vấn con.
- Sử dụng biến con trỏ.

5.3.4.1 Sử dụng truy vấn con

Ta hình dung vấn đề này và cách khắc phục qua ví dụ dưới đây:

Ví dụ 5.17: Ta xét lại trường hợp của hai bảng MATHANG và NHATKYBANHANG như sơ đồ dưới đây:



MAHANG	TENHANG	SOLUONG
H1	Xà phòng	30
H2	Kem đánh răng	45

STT	NGAY	NGUOIMUA	MAHANG	SOLUONG	GIABAN
1	1-1-2004	Ha	H1	10	10000.0000
2	2-2-2004	Phong	H2	20	5000.0000
3	3-3-2004	Thuy	H2	30	6000.0000

Trigger dưới đây cập nhật lại số lượng hàng của bảng MATHANG khi câu lệnh UPDATE được sử dụng để cập nhật cột SOLUONG của bảng NHATKYBANHANG.

```
CREATE TRIGGER trg_nhatkybanhang_update_soluong
ON nhatkybanhang
FOR UPDATE
AS
IF UPDATE(soluong)
  UPDATE mathang
  SET mathang.soluong = mathang.soluong -
       (inserted.soluong-deleted.soluong)
  FROM (deleted INNER JOIN inserted ON
        deleted.stt = inserted.stt) INNER JOIN mathang
        ON mathang.mahang = deleted.mahang
```

Với trigger được định nghĩa như trên, nếu thực hiện câu lệnh:

```
UPDATE nhatkybanhang
SET soluong = soluong + 10
WHERE stt = 1
```

thì dữ liệu trong hai bảng MATHANG và NHATKYBANHANG sẽ là:

MAHANG	TENHANG	SOLUONG
H1	Xà phòng	20
H2	Kem đánh răng	45

Bảng MATHANG

STT	NGAY	NGUOIMUA	MAHANG	SOLUONG	GIABAN
1	1-1-2004	Ha	H1	20	10000.0000
2	2-2-2004	Phong	H2	20	5000.0000
3	3-3-2004	Thuy	H2	30	6000.0000
4	4-4-2004	Dung	H1	40	9000.0000

Bảng NHATKYBANHANG

Tức là số lượng của mặt hàng có mã *H1* đã được giảm đi *10*. Nhưng nếu thực hiện tiếp câu lệnh:

```
UPDATE nhatkybanhang
SET soluong=soluong + 5
WHERE mahang='H2'
```

dữ liệu trong hai bảng sau khi câu lệnh thực hiện xong sẽ như sau:

MAHANG	TENHANG	SOLUONG
H1	Xà phòng	20
H2	Kem đánh răng	40

Bảng MATHANG

STT	NGAY	NGUOIMUA	MAHANG	SOLUONG	GIABAN
1	1-1-2004	Ha	H1	20	10000.0000
2	2-2-2004	Phong	H2	25	5000.0000
3	3-3-2004	Thuy	H2	35	6000.0000

Bảng NHATKYBANHANG

Ta có thể nhận thấy số lượng của mặt hàng có mã *H2* còn lại *40* (giảm đi 5) trong khi đúng ra phải là *35* (tức là phải giảm 10). Như vậy, trigger ở trên không hoạt động đúng trong trường hợp này.

Để khắc phục lỗi gặp phải như trên, ta định nghĩa lại trigger như sau:

```
CREATE TRIGGER  trg_nhatkybanhang_update_soluong
ON nhatkybanhang
FOR UPDATE
AS
IF UPDATE(soluong)
  UPDATE mathang
  SET mathang.soluong = mathang.soluong -
        (SELECT SUM(inserted.soluong-deleted.soluong)
         FROM inserted INNER JOIN deleted
                       ON inserted.stt=deleted.stt
         WHERE inserted.mahang = mathang.mahang)
  WHERE mathang.mahang IN (SELECT mahang
                           FROM inserted)
```

hoặc:

```
CREATE TRIGGER trg_nhatkybanhang_update_soluong
ON nhatkybanhang
FOR UPDATE
AS
IF UPDATE(soluong)
/* Nếu số lượng dòng được cập nhật bằng 1 */
 IF @@ROWCOUNT = 1
 BEGIN
   UPDATE mathang
   SET mathang.soluong = mathang.soluong -
        (inserted.soluong-deleted.soluong)
   FROM (deleted INNER JOIN inserted ON
        deleted.stt = inserted.stt) INNER JOIN mathang
        ON mathang.mahang = deleted.mahang
```

```
 END
ELSE
 BEGIN
   UPDATE mathang
   SET mathang.soluong = mathang.soluong -
       (SELECT SUM(inserted.soluong-deleted.soluong)
        FROM inserted INNER JOIN deleted
                      ON inserted.stt=deleted.stt
        WHERE inserted.mahang = mathang.mahang)
   WHERE mathang.mahang IN (SELECT mahang
                            FROM inserted)
END
```

5.3.4.2 Sử dụng biến con trỏ

Một cách khác để khắc phục lỗi xảy ra như trong ví dụ 5.17 là sử dụng con trỏ để duyệt qua các dòng dữ liệu và kiểm tra trên từng dòng. Tuy nhiên, sử dụng biến con trỏ trong trigger là giải pháp nên chọn trong trường hợp thực sự cần thiết.

Một biến con trỏ được sử dụng để duyệt qua các dòng dữ liệu trong kết quả của một truy vấn và được khai báo theo cú pháp như sau:

```
DECLARE tên_con_trỏ  CURSOR
FOR câu_lệnh_SELECT
```

Trong đó câu lệnh SELECT phải có kết quả dưới dạng bảng. Tức là trong câu lệnh không sử dụng mệnh đề COMPUTE và INTO.

Để mở một biến con trỏ ta sử dụng câu lệnh:

```
OPEN tên_con_trỏ
```

Để sử dụng biến con trỏ duyệt qua các dòng dữ liệu của truy vấn, ta sử dụng câu lệnh FETCH. Giá trị của biến trạng thái @@FETCH_STATUS bằng không nếu chưa duyệt hết các dòng trong kết quả truy vấn.

Câu lệnh FETCH có cú pháp như sau:

```
FETCH [[NEXT|PRIOR|FIST|LAST] FROM] tên_con_trỏ
[INTO danh_sách_biến ]
```

Trong đó các biến trong danh sách biến được sử dụng để chứa các giá trị của các trường ứng với dòng dữ liệu mà con trỏ trỏ đến. Số lượng các biến phải bằng với số lượng các cột của kết quả truy vấn trong câu lệnh DECLARE CURSOR.

Ví dụ 5.18: Tập các câu lệnh trong ví dụ dưới đây minh hoạ cách sử dụng biến con trỏ để duyệt qua các dòng trong kết quả của câu lệnh SELECT

```
DECLARE contro CURSOR
        FOR SELECT mahang,tenhang,soluong FROM mathang
OPEN contro
DECLARE @mahang  NVARCHAR(10)
DECLARE @tenhang NVARCHAR(10)
DECLARE @soluong INT
/*Bắt đầu duyệt qua các dòng trong kết quả truy vấn*/
FETCH NEXT FROM contro
      INTO @mahang,@tenhang,@soluong
WHILE @@FETCH_STATUS=0
     BEGIN
          PRINT 'Ma hang:'+@mahang
          PRINT 'Ten hang:'+@tenhang
          PRINT 'So luong:'+STR(@soluong)
          FETCH NEXT FROM contro
                INTO @mahang,@tenhang,@soluong
     END
/*Đóng con trỏ và giải phóng vùng nhớ*/
CLOSE contro
DEALLOCATE contro
```

Ví dụ 5.19: Trigger dưới đây là một cách giải quyết khác của trường hợp được đề cập ở ví dụ 5.17

```
CREATE TRIGGER trg_nhatkybanhang_update_soluong
ON nhatkybanhang
FOR UPDATE
AS
IF UPDATE(soluong)
BEGIN
     DECLARE @mahang NVARCHAR(10)
     DECLARE @soluong INT

     DECLARE contro CURSOR FOR
       SELECT inserted.mahang,
              inserted.soluong-deleted.soluong AS soluong
       FROM inserted INNER JOIN deleted
            ON inserted.stt=deleted.stt

     OPEN contro
```

```
          FETCH NEXT FROM contro INTO @mahang,@soluong
          WHILE @@FETCH_STATUS=0
            BEGIN
               UPDATE mathang SET soluong=soluong-@soluong
               WHERE mahang=@mahang
               FETCH NEXT FROM contro INTO @mahang,@soluong
            END
          CLOSE contro
          DEALLOCATE contro
     END
END
```