

Laboratory Worksheet Five

SQL SERVER STORED PROCEDURES

A Stored Procedure is an **SQL statement** which is stored in the SQL Server (most SQL statements are not stored there). It is stored for the purpose of **improved performance and security** and tends to be a statement that is used frequently. Reusability is enhanced by allowing INPUT and OUTPUT values to be defined.

General Syntax:

```
CREATE PROCEDURE procedurename
[INPUT  @parameter1 datatype [length] ]
[OUTPUT @parameter2 datatype [length] ]
AS
[BEGIN]
[@parameter1 datatype [length] ]
SQL statement(s)
[END]
```

A stored procedure can be run using **EXEC(UTE)** in the Query window

Task Create a new database - proc3

1. Log in to SQL and use the SQL code below to create a database with the following tables:

DEPT (dnum, dname, numemp) *numemp is the number of employees*
 EMP (enum, ename, salary, age, dnum)

```
CREATE DATABASE proc3;
go

use proc3;
go

CREATE TABLE EMP(
enum tinyint not null,
ename char(15),
salary decimal(7,2),
age tinyint,
dnum tinyint,
CONSTRAINT pk_emp PRIMARY KEY(enum));

INSERT INTO EMP VALUES
(1, 'Smith', 25000, 25, 11),
(2, 'Jones', 32000, 41, 12),
(3, 'White', 19000, 34, 11),
(4, 'Brown', 28000, 22, 12);
```

```
CREATE TABLE DEPT(
  dnum tinyint not null,
  dname char(15),
  numemp tinyint,
  CONSTRAINT pk_dept PRIMARY KEY (dnum)
);
```

For the purposes of these exercises, **do not** enforce referential integrity between the tables. Check the contents of both tables. EMP should have four records, DEPT should be empty.

Task Create a Stored Procedure - *GetEmp*

1. In a New Query Window paste and run the following code to create a procedure to get the EMP table records:

```
CREATE PROCEDURE GetEmp
AS
SELECT * FROM EMP;
```

2. After refreshing the database, check how **GetEmp** is stored as a database object by expanding the database in the Object explorer (click on the + sign beside the database name), expanding the Programmability subdirectory and then expanding the Stored Procedures subdirectory.

You should see dbo.GetEmp

The structure of any Stored Procedures (or triggers) can be checked using the system defined stored procedure **sp_helptext** as **EXEC sp_helptext 'objectname'**

3. Use **sp_helptext** to display the structure of GetEmp.
4. Run GetEmp using the command **EXEC GetEmp;**

Task Create a Stored Procedure with input parameters - *GetEnameBySal*

1. Use the code below to write a stored procedure, GetEnameBySal, that will return the names of employees who earn a salary greater than that supplied as an INPUT parameter.

```
CREATE PROCEDURE GetEnameBySal (@sal decimal(7,2) )
AS
SELECT ename FROM EMP WHERE salary > @sal;
```

Note that the INPUT parameter is defined as a variable starting with @ using the same datatype as the corresponding attribute in the CREATE TABLE statement.

2. Run the procedure twice with different salary values for the WHERE clause parameter:

```
EXEC GetEnameBySal @sal = 20000;           [Result: Smith, Jones, Brown]
EXEC GetEnameBySal @sal = 30000;           [Result: Jones]
```

Task Create a Stored Procedure to Insert Records – *InsDept*

1. Use the code below to write a stored procedure, *InsDept*, to insert records into the DEPT table.

```
CREATE PROCEDURE InsDept (@dnum tinyint, @dnam
char(15), @num tinyint)
AS
BEGIN
    INSERT INTO DEPT
    VALUES (@dnum, @dnam, @num)
END;
```

Note that *InsDept* uses three INPUT parameters corresponding to the three attributes in the DEPT table with data types defined as in the CREATE TABLE statement.

2. Run the procedure twice to add two rows into DEPT


```
EXEC InsDept @dnum = 11, @dnam = 'admin', @num = 0;
EXEC InsDept @dnum = 12, @dnam = 'sales', @num = 0;
```
3. Use the procedure *InsDept* to add a department with dnum = 13, dname = 'finance' and number of employees = 0
4. Check the DEPT table to verify that the rows have been inserted

Task Create a Stored Procedure to Update Records – *UpdEmp1*

As well as INSERT statements DELETE and UPDATE statements are commonly selected for Stored Procedures

1. Use the code below to write a stored procedure, *UpdEmp1*, that given an employee number will change that employee's name.

```
CREATE PROCEDURE UpdEmp1 (@enam char(15), @eno tinyint)
AS
BEGIN
    UPDATE EMP
    SET ename = @enam
    WHERE enum = @eno
END;
```

Note that *UpdEmp1* requires two INPUT parameters for employee name and employee number (enam and eno) defined using the same data type as used in the CREATE TABLE statement.

2. Run *UpdEmp1* to change the name of employee number 4 to 'Green' (currently this employee has ename 'Brown')
3. Check that the update has been made correctly

Task Now Try These Exercises

- A. Call the procedure `InsDep` to add a department with `dnum = 13`, `dname = 'finance'` and number of employees = 0.
- B. Create a stored procedure **UpdEmp2** to update the employee age to a new value for an employee with a particular employee name.
- C. Change the age of employee name 'Smith' to a new age of 42 and check the table to verify the result.

SQL SERVER TRIGGERS

A Database Trigger is a specialised form of stored procedure which runs when an INSERT, DELETE or UPDATE operation has occurred on a specified table. It can be run INSTEAD OF or AFTER any of these three DML (Data Manipulation Language) operations.

General Syntax:

```
CREATE TRIGGER triggername ON table
INSTEAD OF | AFTER      INSERT | DELETE | UPDATE
AS
BEGIN
SQL statement(s)
END
```

Task Create an AFTER INSERT Trigger on Table EMP – tr1

1. Use the code below to write a trigger that will fire after an INSERT operation on the EMP table

```
CREATE TRIGGER tr1 ON EMP
AFTER INSERT
AS
BEGIN
    UPDATE DEPT
    SET numemp = (SELECT COUNT(*) FROM EMP
                  WHERE EMP.dnum = DEPT.dnum)
END;
```

This trigger counts the number of employee records in a department and enters the value in the `numemp` column of the DEPT table AFTER a new EMP record is added. The `numemp` value has initially been set to 0.

2. Insert a new record into EMP

```
INSERT INTO EMP VALUES (8, 'Watts', 30000, 37, 11);
```

3. Check the `numemp` attribute in the DEPT table to verify the effect of trigger `tr1`
4. Insert a second new record into EMP

```
INSERT INTO EMP VALUES (9, 'allen', 20000, 52, 13);
```

5. Check the *numemp* attribute in the DEPT table to verify the effect of trigger tr1

The following tables will be created to illustrate UPDATE and DELETE triggers:

ACCOUNT (accnum, balance)	a basic account table with account number and balance amount
ARCHACC (accnum, balance)	an archive copy of the account table – a trigger will insert any records deleted from the account table into the archive table
AUDIT (id, accnum, old, new, adate)	an audit table holding details of any change made to the account balance – each time the account balance is changed a trigger will store a new record in the audit table with the account number, old balance (old), new balance (new) and the date and time of the change (adate).

Task Create the ACCOUNT, ARCHACC and AUDIT Tables

1. Use the SQL code below to create the following tables in the proc3 database:

```
ACCOUNT (accnum, balance)
ARCHACC (accnum, balance)
AUDIT (id, accnum, old, new, adate)

use proc3;
go

CREATE TABLE ACCOUNT (
accnum int not null,
balance smallint,
CONSTRAINT pk_account PRIMARY KEY (accnum));

INSERT INTO ACCOUNT VALUES
(12345, 150),
(23456, 1400),
(34567, 850),
(45678, 2300),
(56789, 1865);

CREATE TABLE ARCHACC (
accnum int not null,
balance smallint,
CONSTRAINT pk_archacc PRIMARY KEY (accnum));

CREATE TABLE AUDIT (
id int not null IDENTITY,
accnum int,
old smallint,
new smallint,
atime datetime,
CONSTRAINT pk_audit PRIMARY KEY(id));
```

Note that the Primary Key definition of id in the AUDIT table uses **IDENTITY** which creates an attribute that automatically generates a value starting at 1 and rising each time a new row is inserted (so no value can or need be inserted). If manual insertion of a value is required, then SET IDENTITY_INSERT tablename OFF can be used to disable the IDENTITY then switched on again using SET IDENTITY_INSERT tablename ON

IDENTITY implies the default setting of IDENTITY (1, 1) which means starting at 1 and incrementing by 1 on each insertion. If you wished to start from 1000 you would use IDENTITY (1000, 2) would start at 1000 an increment by 2 on each insertion.

Task Create an AFTER DELETE Trigger on Table ACCOUNT – tr2

1. Use the code below to write a trigger that will fire after a DELETE operation on the ACCOUNT table. The trigger will insert a copy of the record into the ARCHACC table after it is deleted from the account table.

```
CREATE TRIGGER tr2 ON ACCOUNT AFTER DELETE AS
BEGIN
    INSERT INTO ARCHACC (accnum, balance)
        SELECT DELETED.accnum, DELETED.balance
        FROM DELETED
END;
```

SQL server uses a temporary table called DELETED which contains records that are being deleted and a temporary table called INSERTED for new records created. This is an alternative way of using the INSERT command – rather than using VALUES to enter values by the user, SELECT..FROM.. clauses are used to get the values from another table:

2. Delete the record for account number 56789 from the ACCOUNT table:

```
DELETE FROM ACCOUNT WHERE accnum = 56789;
```

3. Check both the ACCOUNT and ARCHACC tables to see the effect.

Task Create an AFTER UPDATE Trigger on Table ACCOUNT – tr3

1. Use the code below to write a trigger that will fire after an UPDATE operation on the ACCOUNT table. The trigger will create a record in the AUDIT table whenever an account balance is updated.

```
CREATE TRIGGER tr3 ON ACCOUNT AFTER UPDATE AS
BEGIN
    INSERT INTO AUDIT(accnum, old, new, atime)
        SELECT DELETED.accnum, DELETED.balance,
            INSERTED.balance, getdate()
        FROM DELETED, INSERTED
END;
```

Note that both the temporary DELETED and INSERTED tables are use here since both old and new values of *balance* are needed.

2. Update the balance of account 12345 to 300 (original balance is 150)

```
UPDATE ACCOUNT SET balance = 300 WHERE accnum = 12345;
```

3. Check both the ACCOUNT and AUDIT tables to see the effect.
4. UPDATE a different account balance and recheck the tables.

Task Now Try These Exercises

- D. Create and run a stored procedure **GetAcc** to display all the ACCOUNT table values (see previous **GetEmp** example).

E.

- i. Create a stored procedure **InsAcc** for inserting new records in the ACCOUNT table.
- ii. Execute **InsAcc** to insert account number 67890 with a balance of 2500 (see previous **InsDept** example for inserting records).

F.

- i. Create an archive copy of the employee table called **ARCHEMP**
- ii. Create a trigger **tr4** to copy deleted records from EMP into ARCHEMP (see trigger tr2 for help).
- iii. Test the trigger by deleting rows from the EMP table and checking the EMP and ARCHEMP tables.

G.

- i. Create an audit table

EMPAUDIT (id, enum, oldsalary , newsalary, changedate)

With *id* defined as an IDENTITY column (see previous CREATE TABLE AUDIT for help)

- ii. Create a trigger **tr5** which fires when EMP is updated and records the employee number affected, the old salary, new salary and the current date/time in EMPAUDIT.
- iii. Test the trigger by updating the salary of an employee and checking the EMP and EMPAUDIT tables.