STORED PROCEDURES

Execution Environment

- Code can be executed at two locations
  - Client
  - Server
- Advantages of client-side programming
  - Can more easily serve users with vastly different needs
  - Can avoid additional network traffic
  - Can distribute processor load to local environments
  - Can create better user-interface interaction
- Advantages of server-side programming
  - Can more easily ensure uniform application of business rules
  - Can update code more easily

What Are Stored Procedures?

A collection of T-SQL statements stored on the server

Stored Procedures Can:

- Accept input parameters
- Return output parameters or rowset
- Return a status value to indicate success or failure

Benefits of using Stored Procedures:

- Promotes modular programming
- Provides security attributes and permission chaining
- Allows delayed binding and code reuse
- Reduces network traffic

Types of Stored Procedures

- User-defined stored procedures
  - User-defined procedures that must be explicitly called
- Triggers
  - User-defined procedures that execute automatically when data in a given table is modified
- System procedures
  - Built in procedures that read or modify one or more system tables

Guidelines for Creating Stored Procedures

Rules for designing stored procedures:

- Qualifying names inside of stored procedures
- Obfuscating procedure definitions
- SET statement options
- Naming conventions
- Execution Context
- Using @@nestlevel

Syntax for Creating Stored Procedures

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
        [ { @parameter [ type_schema_name. ] data_type }
         [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
        ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { <sql_statement> [;][ ...n ] | <method_specifier> }
[;]
<procedure_option> ::=
        [ ENCRYPTION ]
        [ RECOMPILE ]
        [ EXECUTE_AS_Clause ]

<sql_statement> ::=
{ [ BEGIN ] statements [ END ] }
<method_specifier> ::=
EXTERNAL NAME assembly_name.class_name.method_name
```

Syntax for Altering Stored Procedures

```
ALTER { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
        [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ [ OUT [ PUT ]    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS
        { <sql_statement> [ ...n ] | <method_specifier> }

<procedure_option> ::=
[ ENCRYPTION ]
[ RECOMPILE ]
[ EXECUTE_AS_Clause ]

<sql_statement> ::=
{ [ BEGIN ] statements [ END ] }

<method_specifier> ::=
EXTERNAL NAME
assembly_name.class_name.method_name
```

Syntax for Dropping Stored Procedures

- DROP { PROC | PROCEDURE } { [schema_name.] procedure } [,...n]

Removes one or more stored procedures or procedure groups from the current database.

How Are Stored Procedures Created?

Creating a Stored Procedure

```
CREATE PROCEDURE HumanResources.usp_GetEmployeesName
@NamePrefix char(1)
AS
BEGIN
SELECT BusinessEntityID, FirstName, LastName,
EmailAddress
FROM HumanResources.vEmployee
WHERE FirstName LIKE @NamePrefix + '%'
ORDER BY FirstName
END
```

Calling a Stored Procedure

```
EXECUTE HumanResources.usp_GetEmployeesName 'A‘
```

Creating Parameterized Stored Procedures

- Stored Procedure Parameters
- Table-valued Parameters

Passing Parameters

- Two methods for passing values to parameters:
  - Passing by parameter position
  - Passing by parameter name

Default Value

- A default value is a value assigned to a parameter for which no value has been received from the **exec** statement
- Example:

```
create proc proc_state_authors
    (@state char(2) = "CA")
as
    select au_lname, au_fname, state
    from authors
    where state = @state
return

exec proc_state_authors         -- No state value
passed

au_lname  au_fname        state
--------- ---------       -----
White          Johnson         CA
Green          Marjorie        CA
...
```

Guidelines for Handling Exceptions

TRY/CATCH requirements:

- Each TRY...CATCH construct must be inside a single batch stored procedure, or trigger
- A TRY block must be immediately followed by a CATCH block
- TRY...CATCH constructs can be nested

Implementing Triggers

- What Are Triggers?
- How an INSERT Trigger Works
- How a DELETE Trigger Works
- How an UPDATE Trigger Works
- How an INSTEAD OF Trigger Works
- How Nested Triggers Work
- Considerations for Recursive Triggers

What Are Triggers?

Triggers are:

- Special stored procedures that execute when INSERT, UPDATE, or DELETE statements modify a table
- Part of a single transaction along with the initiating statement

Two categories:

- AFTER triggers execute after an INSERT, UPDATE, or DELETE statement
- INSTEAD OF triggers execute instead of an INSERT, UPDATE, or DELETE statement

Creating Triggers

- Simplified syntax:

```
create trigger trigger_name
on table_name
for {insert | update | delete} [, {insert | update | delete} ...]
as
        sql_statements
```

How an INSERT Trigger Works

1 INSERT statement executed

2 INSERT statement logged

3 AFTER INSERT trigger statements executed

```
create trigger trg_i_sales
    on sales
    for insert
    as
     SET NOCOUNT ON;
      if datename (dw,getdate()) = "Sunday"
        begin
          raiserror 40070, "Sales cannot be
            processed on Sunday."
          rollback tran
          return
        end
```

How a DELETE Trigger Works

1 DELETE statement executed

2 DELETE statement logged

3 AFTER DELETE trigger statements executed

```
CREATE TRIGGER [delCategory] ON [Categories]
    AFTER DELETE AS
    BEGIN
      UPDATE P SET [Discontinued] = 1
      FROM [Products] P INNER JOIN deleted as d
      ON
      P.[CategoryID] = d.[CategoryID]
    END;
```

How an UPDATE Trigger Works

1 UPDATE statement executed

2 UPDATE statement logged

3 AFTER UPDATE trigger statements executed

```
CREATE TRIGGER [updtProductReview] ON [Production].[ProductReview]
    AFTER UPDATE NOT FOR REPLICATION AS
    BEGIN
      UPDATE [Production].[ProductReview]
      SET [Production].[ProductReview].[ModifiedDate] =
        GETDATE() FROM inserted
      WHERE inserted.[ProductReviewID] =
        [Production].[ProductReview].[ProductReviewID];
    END;
```

How an INSTEAD OF Trigger Works

1. UPDATE, INSERT, or DELETE statement executed
2. Executed statement does not occur
3. INSTEAD OF trigger statements executed

```
CREATE TRIGGER [delEmployee] ON [HumanResources].[Employee]
    INSTEAD OF DELETE NOT FOR REPLICATION AS
    BEGIN
      SET NOCOUNT ON;
      DECLARE @DeleteCount int;
      SELECT @DeleteCount = COUNT(*) FROM deleted;
      IF @DeleteCount > 0
      BEGIN …
      END;
    END;
```

Dropping Triggers

- Simplified syntax:
  drop trigger *trigger_name*
- Example:
  ```
  drop trigger trg_i_sales
  ```