Database Management Systems

T-SQL: STORED PROCEDURES, USER-DEFINED FUNCTIONS, TRIGGERS

September 2019

Procedural SQL programs

Type	Batches	How it's stored	How it's executed	Accepts parameters
Script	Multiple	In a file on a disk	From within a client tool such as the Management Studio or SQLCMD	No
Stored procedure	One only	In an object in the database	By an application or within a SQL script	Yes
User-defined function	One only	In an object in the database	By an application or within a SQL script	Yes
Trigger	One only	In an object in the database	Automatically by the server when a specific action query is executed	No

Part 1

STORED PROCEDURES

Stored Procedures

- Stored in compiled form in database after first execution (precompiled)
- Faster than execution of equivalent SQL script

Stored Procedure

Created with the CREATE PROC statement with the following syntax:

```
CREATE {PROC|PROCEDURE} procedure_name
[parameter_declarations]
[WITH [RECOMPILE] [, ENCRYPTION] [, EXECUTE_AS_clause]]
AS sql_statements
```

- Must be the first and only statement in a batch
- procedure_name in current database
- # procedure_name local, temporary in temdb
- ## procedure_name global, temporary in temdb

Called using EXEC statement

- Syntax: EXEC procedure_name

Precompiled: compiled and stored in database after first execution

Example

Creating a stored procedure:

```
USE AP;
GO
CREATE PROC spInvoiceReport
AS

SELECT VendorName, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices JOIN Vendors
     ON Invoices.VendorID = Vendors.VendorID
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
ORDER BY VendorName;
```

Result set from EXEC spInvoiceReport;

	VendorName	InvoiceNumber	InvoiceDate	InvoiceTotal
1	Blue Cross	547480102	2016-04-01 00:00:00	224.00
2	Cardinal Business Media, Inc.	134116	2016-03-28 00:00:00	90.36
3	Data Reproductions Corp	39104	2016-03-10 00:00:00	85.31
4	Federal Express Corporation	963253264	2016-03-18 00:00:00	52.25
5	Federal Express Corporation	263253268	2016-03-21 00:00:00	59.97

Parameters

Defined and used locally within the procedure:

- Names started with @
- Accept any type but table

Two types:

- Input: accepting values passed
- Output: storing values passed back

Optional parameters allowed

Could be passed by position or name on calling

Defining syntax:

```
@parameter_name_1 data_type [= default] [OUTPUT]
[, @parameter_name_2 data_type [= default] [OUTPUT]]...
```

Example

Input / Output parameters:

```
CREATE PROC spInvTotal1
      @DateVar smalldatetime,
      @InvTotal money OUTPUT
AS
SELECT @InvTotal = SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

Optional parameters:

```
CREATE PROC spInvTotal2
      @DateVar smalldatetime = NULL
AS
IF @DateVar IS NULL
    SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;
SELECT SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

Example (cont.)

Defining a SP with three parameters:

```
CREATE PROC spInvTotal3
        @InvTotal money OUTPUT,
        @DateVar smalldatetime = NULL,
        @VendorVar varchar(40) = '%'
AS

IF @DateVar IS NULL
   SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;

SELECT @InvTotal = SUM(InvoiceTotal)
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE (InvoiceDate >= @DateVar) AND
      (VendorName LIKE @VendorVar);
```

Example (cont.)

Calling with parameters passed by position:

```
DECLARE @MyInvTotal money;
EXEC spInvTotal3 @MyInvTotal OUTPUT, '2016-02-01', 'P%';
```

By name

```
DECLARE @MyInvTotal money;
EXEC spInvTotal3 @DateVar = '2016-02-01', @VendorVar = 'P%',
    @InvTotal = @MyInvTotal OUTPUT;
```

Calling with optional parameters omitted:

```
DECLARE @MyInvTotal money;
EXEC spInvTotal3 @VendorVar = 'M%', @InvTotal = @MyInvTotal OUTPUT;
```

Return Statement

Immediately exits the procedure & returns an optional value

- Returns 0 by default

Syntax

- RETURN [integer_expression]

Calling

- EXEC @variable_name = procedure_name [parameter list];

Example

Define

```
CREATE PROC spInvCount
       @DateVar smalldatetime = NULL,
       @VendorVar varchar(40) = '%'
AS

IF @DateVar IS NULL
   SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;

DECLARE @InvCount int;

SELECT @InvCount = COUNT(InvoiceID)
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE (InvoiceDate >= @DateVar) AND
      (VendorName LIKE @VendorVar);

RETURN @InvCount;
```

Call

- Invoice count: 6

```
DECLARE @InvCount int;
EXEC @InvCount = spInvCount '2016-02-01', 'P%';
PRINT 'Invoice count: ' + CONVERT(varchar, @InvCount);
```

Error Handling

Data validation within SPs to prevent errors

- Make data valid if not, or
- Throw errors

Use TRY … CATCH in calling programs to handle the error

Client connection terminated immediately otherwise

Syntax

- THROW [error_number, message, state]
- State argument to identity severity
- THROW without parameters must be coded in CATCH block
- THROW code within a block must be preceded by a semicolon, e.g.

```
BEGIN
    ;
    THROW 50001, 'Not a valid VendorID!', 1;
END;
```

Example

SP to test for a valid foreign key

```
CREATE PROC spInsertInvoice
       @VendorID    int,            @InvoiceNumber  varchar(50),
       @InvoiceDate smalldatetime, @InvoiceTotal   money,
       @TermsID     int,            @InvoiceDueDate smalldatetime
AS

IF EXISTS(SELECT * FROM Vendors WHERE VendorID = @VendorID)
    INSERT Invoices
    VALUES (@VendorID, @InvoiceNumber,
            @InvoiceDate, @InvoiceTotal, 0, 0,
            @TermsID, @InvoiceDueDate, NULL);
ELSE
    THROW 50001, 'Not a valid VendorID!', 1;
```

Example (cont.)

Calling script

```
BEGIN TRY
    EXEC spInsertInvoice
         799,'ZXK-799','2016-05-01',299.95,1,'2016-06-01';
END TRY
BEGIN CATCH
    PRINT 'An error occurred.';
    PRINT 'Message: ' + CONVERT(varchar, ERROR_MESSAGE());
    IF ERROR_NUMBER() >= 50000
        PRINT 'This is a custom error message.';
END CATCH;
```

System response

```
An error occurred.
Message: Not a valid VendorID!
This is a custom error message.
```

Passing Tables as Parameters to SPs

Must create a user-defined table type

- Can only used as input parameters
- Foreign keys not allowed
- Read only within SPs

Syntax for creating user-defined table type:

```
CREATE TYPE TableTypeName AS
TABLE
table_definition
```

Example

Create type

```
CREATE TYPE LineItems AS
TABLE
(InvoiceID          INT             NOT NULL,
InvoiceSequence     SMALLINT        NOT NULL,
AccountNo           INT             NOT NULL,
ItemAmount          MONEY           NOT NULL,
ItemDescription     VARCHAR(100)    NOT NULL,
PRIMARY KEY (InvoiceID, InvoiceSequence));
```

Create SP

```
CREATE PROC spInsertLineItems
    @LineItems LineItems READONLY
AS
    INSERT INTO InvoiceLineItems
    SELECT *
    FROM @LineItems;
```

Example (cont.)

Calling

```
DECLARE @LineItems LineItems;

INSERT INTO @LineItems VALUES (114, 1, 553, 127.75, 'Freight');
INSERT INTO @LineItems VALUES (114, 2, 553, 29.25, 'Freight');
INSERT INTO @LineItems VALUES (114, 3, 553, 48.50, 'Freight');

EXEC spInsertLineItems @LineItems;
```

Response

```
(1 row(s) affected)
(1 row(s) affected)
(1 row(s) affected)
(3 row(s) affected)
```

Delete & Modification

Delete

```
DROP {PROC|PROCEDURE} procedure_name [, ...]
```

Example

```
DROP PROC spVendorState;
```

Modify

```
ALTER {PROC|PROCEDURE} procedure_name
[parameter declarations]
[WITH [RECOMPILE] [, ENCRYPTION] [, EXECUTE_AS_clause]]
AS sql_statements
```

Example

```
CREATE PROC spVendorState
       @State varchar(20)
AS
SELECT VendorName
FROM Vendors
WHERE VendorState = @State;
```

```
ALTER PROC spVendorState
      @State varchar(20) = NULL
AS
IF @State IS NULL
   SELECT VendorName
   FROM Vendors;
ELSE
   SELECT VendorName
   FROM Vendors
   WHERE VendorState = @State;
```

System Stored Procedures

Hundreds …

- https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/system-stored-procedures-transact-sql

Stored in Master database (called from any database)

Create your own system procedures:

- In Master database
- Name starts with sp_

Avoid using in production programs (version changes)

Commonly Used System Stored Procedures

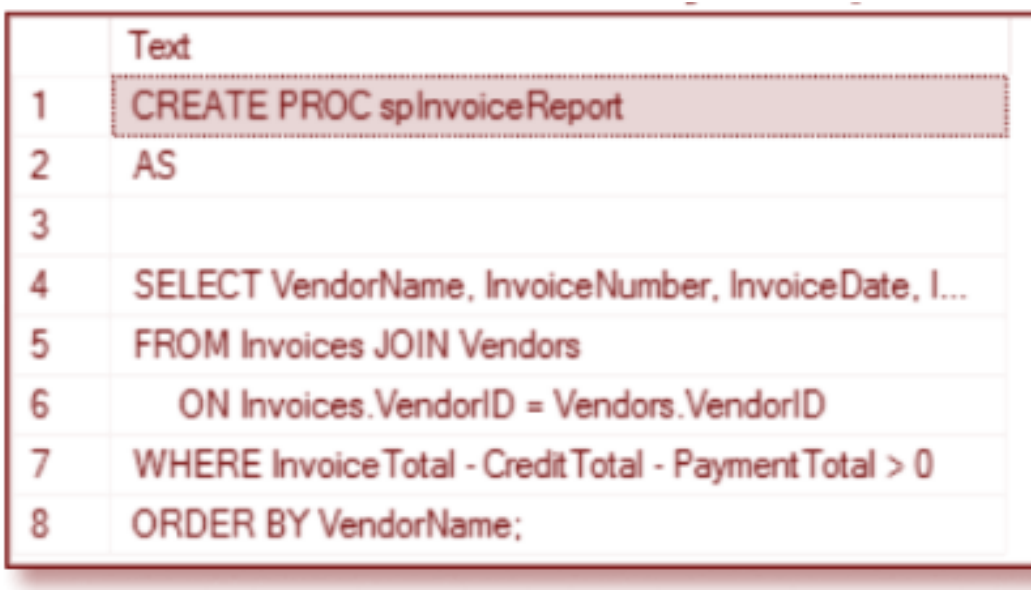
Procedure	Description
`sp_Help [name]`	Returns information about the specified database object or data type. Without a parameter, returns a summary of all objects in the current database.
`sp_HelpText name`	Returns the text of an unencrypted stored procedure, user-defined function, trigger, or view.
`sp_HelpDb [database_name]`	Returns information about the specified database or, if no parameter is specified, all databases.
`sp_Who [login_ID]`	Returns information about who is currently logged in and what processes are running. If no parameter is specified, information on all active users is returned.
`sp_Columns name`	Returns information about the columns defined in the specified table or view.

Example (using sp_HelpText)

Call

```
USE AP;
EXEC sp_HelpText spInvoiceReport;
```

Result



	Text
1	CREATE PROC spInvoiceReport
2	AS
3	
4	SELECT VendorName, InvoiceNumber, InvoiceDate, I...
5	FROM Invoices JOIN Vendors
6	ON Invoices.VendorID = Vendors.VendorID
7	WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
8	ORDER BY VendorName;

Part 2

USER-DEFINED FUNCTIONS

User-defined Functions (UDFs)

Scalar-valued function

- Return a single value of any T-SQL data type.

Simple table-valued function

- Return a table based on a single select statement.

Muti-statement table-valued functions

- Return a table based on multiple statements.

Invoke from within expressions

- Must specify schema name

Parameters:

- Input only
- Passed by position only
- DEFAULT keyword instead if omitted.

Creating Scalar-valued Functions

Syntax to create:

```
CREATE FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS data_type
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
    [AS]
BEGIN
    [sql_statements]
    RETURN scalar_expression
END
```

- Created in default schema if not specified
- SCHEMABINDING: prohibit deleting of views & tables used
- ENCRYPTION: encrypt function code
- EXECUSE AS: specify security context to execute

Example

Creating a function:

```
CREATE FUNCTION fnBalanceDue()
    RETURNS money
BEGIN
    RETURN (SELECT SUM(InvoiceTotal - PaymentTotal - CreditTotal)
            FROM Invoices
            WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0);
END;
```

Calling:

```
PRINT 'Balance due: $' + CONVERT(varchar, dbo.fnBalanceDue(), 1);
```

Simple Table-valued Function

Also called inline table-valued function

Syntax to create:

```
CREATE FUNCTION [schema_name.]function_name
     ([@parameter_name data_type [= default]] [, ...])
     RETURNS TABLE
     [WITH [ENCRYPTION] [, SCHEMABINDING]]
     [AS]
RETURN [(] select_statement [)]
```

- Select statement defines the table to return

Example

Creating:

```
CREATE FUNCTION fnTopVendorsDue
    (@CutOff money = 0)
    RETURNS table
RETURN
    (SELECT VendorName, SUM(InvoiceTotal) AS TotalDue
    FROM Vendors JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
    WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
    GROUP BY VendorName
    HAVING SUM(InvoiceTotal) >= @CutOff);
```

Using in a join query:

```
SELECT Vendors.VendorName, VendorCity, TotalDue
FROM Vendors JOIN dbo.fnTopVendorsDue(DEFAULT) AS TopVendors
     ON Vendors.VendorName = TopVendors.VendorName;
```

Multi-statement Table-valued Function

Rarely used

Syntax to create:

```
CREATE FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS @return_variable TABLE
    (column_name_1 data_type [column_attributes]
    [, column_name_2 data_type [column_attributes]]...)
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
    [AS]
BEGIN
    sql_statements
    RETURN
END
```

- Defines structure of a new table
- Statements to get data into the table to return

Creating a multi-statement table-valued function

Example

```
CREATE FUNCTION fnCreditAdj (@HowMuch money)
    RETURNS @OutTable table
           (InvoiceID int, VendorID int, InvoiceNumber varchar(50),
            InvoiceDate smalldatetime, InvoiceTotal money,
            PaymentTotal money, CreditTotal money)
BEGIN
    INSERT @OutTable
        SELECT InvoiceID, VendorID, InvoiceNumber, InvoiceDate,
               InvoiceTotal, PaymentTotal, CreditTotal
        FROM Invoices
        WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;
    WHILE (SELECT SUM(InvoiceTotal - CreditTotal  - PaymentTotal)
           FROM @OutTable) >= @HowMuch
        UPDATE @OutTable
        SET CreditTotal = CreditTotal + .01
        WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;
    RETURN;
END;
```

Example

Calling in a query:

```
SELECT VendorName, SUM(CreditTotal) AS CreditRequest
FROM Vendors JOIN dbo.fnCreditAdj(25000) AS CreditTable
     ON Vendors.VendorID = CreditTable.VendorID
GROUP BY VendorName;
```

Result:

	VendorName	CreditRequest
1	Blue Cross	224.00
2	Cardinal Business Media, Inc.	90.36
3	Data Reproductions Corp	85.31
4	Federal Express Corporation	210.89
5	Ford Motor Credit Company	503.20
6	Ingram	579.42
7	Malloy Lithographing Inc	6527.26

Delete / Change Functions

Delete:

```
DROP FUNCTION [schema_name.]function_name [, ...]
```

Change Scalar-valued function:

```
ALTER FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS data_type
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
BEGIN
    [sql_statements]
    RETURN scalar_expression
END
```

Delele/Change Functions (cont.)

Modify simple table functions

```
ALTER FUNCTION [schema_name.]function_name
     ([@parameter_name data_type [= default]] [, ...])
     RETURNS TABLE
     [WITH [ENCRYPTION] [, SCHEMABINDING]]
RETURN [(] select_statement [)]
```

Modify multi-statement table-valued functions

```
ALTER FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS @return_variable TABLE
    (column_name_1 data_type [column_attributes]
    [, column_name_2 data_type [column_attributes]]...)
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
BEGIN
    sql_statements
    RETURN
END
```

Part 3

TRIGGERS

Triggers

Automatically executed (fired) in response to an action query

- No direct invoke
- No parameters
- No return values.

Associated with a single table / view.

Can bet set to fire:

- AFTER / FOR, or INSTEAD OF
- on a combination of INSERT, UPDATE, DELETE

AFTER triggers

- Multiple ones possible for each action on a table
- Views can’t have an AFTER trigger

INSTEAD OF triggers

- Only one for each action per view / table

Triggers (cont)

Within a trigger:

- Inserted table contains the new rows for insert and update operations
- Deleted table contains original rows for update and delete operations

Syntax to create:

```
CREATE TRIGGER trigger_name
    ON {table_name|view_name}
    [WITH [ENCRYPTION] [,] [EXECUTE_AS_clause]]
    {FOR|AFTER|INSTEAD OF} [INSERT] [,] [UPDATE] [,] [DELETE]
AS sql_statements
```

Example

```
CREATE TRIGGER Vendors_INSERT_UPDATE
     ON Vendors
     AFTER INSERT,UPDATE
AS
     UPDATE Vendors
     SET VendorState = UPPER(VendorState)
     WHERE VendorID IN (SELECT VendorID FROM Inserted);
```

An AFTER trigger to change to upper case

Insert data to fire the trigger

```
INSERT Vendors
VALUES ('Peerless Uniforms, Inc.', '785 S Pixley Rd', NULL,
        'Piqua', 'Oh', '45356', '(937) 555-8845', NULL, NULL, 4, 550);
```

Result

	VendorID	VendorName	VendorAddress1	VendorAddress2	VendorCity	VendorState	VendorZip
1	125	Peerless Uniforms, Inc.	785 S Pixley Rd	NULL	Piqua	OH	45356

AFTER Triggers

Fires after the action query is executed.

Never fires if the action query causes an error.

Can be used to archive deleted data

Example

```
CREATE TRIGGER Invoices_DELETE
    ON Invoices
    AFTER DELETE
AS
INSERT INTO InvoiceArchive
    (InvoiceID, VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
        PaymentTotal, CreditTotal, TermsID, InvoiceDueDate, PaymentDate)
    SELECT InvoiceID, VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
        PaymentTotal, CreditTotal, TermsID, InvoiceDueDate, PaymentDate
    FROM Deleted
```

A trigger to archive deleted data

Delete data to fire the trigger

```
DELETE Invoices
WHERE VendorID = 37
```

Rows inserted in to InvoiceArchive

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	Ter
1	113	37	547480102	2016-04-01 00:00:00	224.00	0.00	0.00	3
2	50	37	547479217	2016-02-07 00:00:00	116.00	116.00	0.00	3
3	46	37	547481328	2016-02-03 00:00:00	224.00	224.00	0.00	3

INSTEAD OF Triggers

Executed instead of the action query (Action query never executed) => typically contains code performing the operation.

Typically used to provide for updatable views, preventing errors e.g. constraint violations before they occur.

Only one INSTEAD OF trigger / table or view / type of action. Tables with foreign key constraints specifying CASCADE UPDATE / DELETE options could not have INSTEAD triggers for these operations.

Example

```
CREATE TRIGGER IBM_Invoices_INSERT
    ON IBM_Invoices
    INSTEAD OF INSERT
AS
DECLARE @InvoiceDate smalldatetime, @InvoiceNumber varchar(50),
        @InvoiceTotal money, @VendorID int,
        @InvoiceDueDate smalldatetime, @TermsID int,
        @DefaultTerms smallint, @TestRowCount int;
SELECT @TestRowCount = COUNT(*) FROM Inserted;
IF @TestRowCount = 1
    BEGIN
        SELECT @InvoiceNumber = InvoiceNumber, @InvoiceDate = InvoiceDate,
            @InvoiceTotal = InvoiceTotal
        FROM Inserted;
        IF (@InvoiceDate IS NOT NULL AND @InvoiceNumber IS NOT NULL AND
            @InvoiceTotal IS NOT NULL)
            BEGIN
                SELECT @VendorID = VendorID, @TermsID = DefaultTermsID
                FROM Vendors
                WHERE VendorName = 'IBM';
```

INSTEAD OF trigger

INSTEAD OF trigger (cont.)

Example (cont.)

```
            SELECT @DefaultTerms = TermsDueDays
            FROM Terms
            WHERE TermsID = @TermsID;

            SET @InvoiceDueDate = @InvoiceDate + @DefaultTerms;

            INSERT Invoices
               (VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
                TermsID, InvoiceDueDate, PaymentDate)
            VALUES (@VendorID, @InvoiceNumber, @InvoiceDate,
               @InvoiceTotal, @TermsID, @InvoiceDueDate, NULL);
        END;
    END;
ELSE
    THROW 50027, 'Limit INSERT to a single row.', 1;
```

Insert data to fire the trigger

```
INSERT IBM_Invoices
VALUES ('RA23988', '2016-05-09', 417.34);
```

Using Triggers To Enforce Data Consistency

Enforce database rules can’t be enforced by constraints

Enforce the same rules as constraints with more flexibility

Example

- A trigger to validate line item amounts when posting a payment (next slide)

Example

```
CREATE TRIGGER Invoices_UPDATE
    ON Invoices
    AFTER UPDATE
AS
IF EXISTS            --Test whether PaymentTotal was changed
 (SELECT *
  FROM Deleted JOIN Invoices
    ON Deleted.InvoiceID = Invoices.InvoiceID
  WHERE Deleted.PaymentTotal <> Invoices.PaymentTotal)
  BEGIN
    IF EXISTS        --Test whether line items total and InvoiceTotal match
     (SELECT *
      FROM Invoices JOIN
          (SELECT InvoiceID, SUM(InvoiceLineItemAmount) AS SumOfInvoices
           FROM InvoiceLineItems
           GROUP BY InvoiceID) AS LineItems
        ON Invoices.InvoiceID = LineItems.InvoiceID
      WHERE (Invoices.InvoiceTotal <> LineItems.SumOfInvoices) AND
            (LineItems.InvoiceID IN (SELECT InvoiceID FROM Deleted)))
      BEGIN
        ;
        THROW 50113, 'Correct line item amounts before posting payment.', 1;
        ROLLBACK TRAN;
      END;
  END;
```

Example (cont.)

Firing the trigger by update data:

```
UPDATE Invoices
SET PaymentTotal = 662, PaymentDate = '2016-05-09'
WHERE InvoiceID = 98;
```

Result:

```
Msg 50113, Level 16, State 1, Procedure Invoices_UPDATE, Line 23
Correct line item amounts before posting payment.
```

Delete / Modify Triggers

Delete

```
DROP TRIGGER trigger_name [, ...]
```

e.g.

```
DROP TRIGGER Vendors_INSERT_UPDATE;
```

Modify

```
ALTER TRIGGER trigger_name
ON {table_name|view_name}
[WITH [ENCRYPTION] [,] [EXECUTE_AS_clause]]
{FOR|AFTER|INSTEAD OF} [INSERT] [,] [UPDATE] [,] [DELETE]
AS sql_statements
```

e.g.

```
ALTER TRIGGER Vendors_INSERT_UPDATE
    ON Vendors
    AFTER INSERT,UPDATE
AS
    UPDATE Vendors
    SET VendorState = UPPER(VendorState),
        VendorAddress1 = LTRIM(RTRIM(VendorAddress1)),
        VendorAddress2 = LTRIM(RTRIM(VendorAddress2))
    WHERE VendorID IN (SELECT VendorID FROM Inserted);
```