T-SQL Programming – Part 2

Transact-SQL (T-SQL) is Microsoft's (and Sybase's – now owned by SAP) proprietary extension to SQL. SQL is a standardized computer language that was originally developed by IBM for querying, altering and defining relational databases, using declarative statements. <u>SQL is the only way to retrieve data from a relational database – period end of discussion!</u>

Startup

Perform the following startup tasks.

1. Download the *Northwind* ZIP file from the Labs/T-SQL item on Blackboard. Save the ZIP file to your USB drive or to *My Documents*. Extract the contents.
2. Attach the *Northwind* database using **SQL Server Management Studio**.

IF...ELSE Logic

Introduction

To conditionally allow different SQL code to be executed based on the results of a logical test, T-SQL supports an IF statement. The IF statement allows you to selectively execute a single line or block of SQL code based upon the results of a logical test that evaluates to Boolean expression which is either TRUE or FALSE.

```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

If the Boolean expression evaluates to TRUE, then the SQL code immediately following the expression is executed. If the condition is false, then the ELSE code is executed. Notice that the ELSE code is optional as is signified by being enclosed in square brackets. If there is a FALSE condition when the ELSE code is excluded, then the next line following the IF statement is executed, since no ELSE condition exists.

The SQL code to be executed can be a single TSQL statement or a block of SQL code (i.e., more that a single SQL statement). If a block of SQL is used, then it will need to be enclosed in a BEGIN and END statement.

This first example will show how the IF statement would look to execute a single statement, if the condition is true. Here we test whether a variable is set to a specific value. If the variable is set to a specific value, then I print out the appropriate message.

```
DECLARE @x INT;
SET @x = 29;
IF @x = 29 PRINT 'The number is 29';
IF @x = 30 PRINT 'The number is 30';
```

The above code prints out only the phrase *The number is 29*, because the first IF statement evaluates to true. Since the second IF is false, the second print statement is not executed.

The condition statement can also contain a SELECT statement. The SELECT statement will need to return value or set of values that can be tested. If a SELECT statement is used the statement needs to be enclosed in parentheses.

```
DECLARE @x INT;
SELECT @x=count(*) FROM Northwind.dbo.Customers WHERE City = 'Boston'
IF @x > 0 PRINT 'Found Boston';

/* Compared to this */
IF (SELECT count(*) FROM Northwind.dbo.Customers WHERE City = 'Boston') > 0
       PRINT 'Found Boston';
```

T-SQL allows you to execute a block of code as well. A code block is created by using a BEGIN statement before the first line of code in the code block, and an END statement after that last line of code in the code block. Here is any example that executes a code block when the IF statement condition evaluates to true.

```
IF (SELECT count(*) FROM Northwind.dbo.Customers WHERE City = 'Boston') > 0
BEGIN
       PRINT 'Count of Boston Customers:';
       SELECT count(*) FROM Northwind.dbo.Customers WHERE City = 'Boston';
END;
```

Above a series of PRINT statements will be executed if this IF statement finds we have a non-zero number of customers in the city of Boston. If not, the statements are not executed.

Sometimes you want to not only execute some code when you have a true condition, but also want to execute a different set of T-SQL statements when you have a false condition. If this is your requirement then you will need to use the IF...ELSE construct. In this case, if the condition is true then the statement or block of code following the IF clause is executed, but if the condition evaluates to false then the statement or block of code following the ELSE clause will be executed. Let's go through a couple of examples.

For the first example let's say you need to determine whether to update or add a record to the Customers table in the *Northwind* database. The decision is based on whether the customer exists in the *Northwind.dbo.Customers* table. Here is the T-SQL code to perform this existence test for two different Customer ID's.

```
IF EXISTS(SELECT * FROM Northwind.dbo.Customers WHERE [Last Name] = 'Anderson')
       PRINT 'Need to update Customer Record Anderson';
ELSE
       PRINT 'Need to add Customer Record Anderson';

IF EXISTS(SELECT * FROM Northwind.dbo.Customers WHERE ID = 'Larse')
       PRINT 'Need to update Customer Record Larse';
ELSE
       PRINT 'Need to add Customer Record Larse';
```

The first IF...ELSE checks to see if a customer Anderson exists. If they exist, then the message *Need to update Customer Record* is printed. If customer Anderson exists does not exist, the *Need to add Customer Record* is displayed. This logic is repeated for customer Larse.

Run the above code against the *Northwind* database.

As you can see from the results, the record for customer Anderson needs to be updated while the record for customer Larse needs to be added.

If you have complicated logic that needs to be performed prior to determining what T-SQL statements to execute you can either use multiple conditions on a single IF statement, or nest your IF statements. Here is a script that determines if the scope of the query is in the 'Northwind' database and if the Customers table exists. I have written this query two different ways, one with multiple conditions on a single IF statement, and the other by having nested IF statements.

```
-- Single IF Statement with multiple conditions
USE Northwind;
IF DB_NAME() = 'Northwind' AND OBJECT_ID('Customers') IS NOT NULL
       PRINT 'Table Customers Exists';
ELSE
       PRINT 'Not using the Northwind database' +
       ' or Table Customers does not exist';

-- Nested IF Statements;
IF DB_NAME() = 'Northwind'
       IF OBJECT_ID('Customers') IS NOT NULL
              PRINT 'Table Customers Exist'
       ELSE
              PRINT 'Table Customers does not exist'
ELSE
       PRINT 'Not using the Northwind Database'
```

As you can see I tested to see if the query was being run from the Northwind database and whether the Customers table exisits. If this was true, I printed the message Table Customers Exists. In the first example I had multiple conditions in a single IF statement. Since I was not able to determine which parts of the conditions in the IF statement where false the ELSE portion printed the message Not in Northwind database or Table Customer does not exist.

In the second example using a nested IF, I was able to determine whether I was in the wrong database or the object Customers did not exist. This allowed me to have two separate print statements to reflect exactly what condition was getting a false value.

Practical Example

In the coming weeks you will begin to create ETL processes to build your project data warehouse database. One cycle that is repeated many times is *drop DB*, *create DB*, *create tables*, *create table constraints*. In fact, you may have to complete this cycle dozens of times. Having a T-SQL script that can complete these steps in one fell swoop will be a significant boost to your productivity.

Since T-SQL is very finicky when it comes to runtime errors in scripts , so, you must check for potential error conditions in advance in order to avoid committing them.

```
/* This script checks for the existance of a database.
 * If it exists, the DB is dropped (i.e., deleted.
 * Two tables are added where one table contains
 * a foreign key into the other.
 */
USE [master]
```

```
GO

IF DB_ID('MyDB') IS NOT NULL
BEGIN
       PRINT 'Dropping DB';
       ALTER DATABASE MyDB SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
       DROP DATABASE MyDB;
END;

PRINT 'Creating DB';

CREATE DATABASE MyDB
ON  PRIMARY
 ( NAME = 'MyDB', FILENAME = 'E:\MyDB.mdf')
LOG ON
 ( NAME = 'MyDB_log', FILENAME = 'E:\MyDB.ldf')
GO

USE MyDB;

IF OBJECT_ID('aTable') IS NOT NULL
BEGIN
       PRINT 'Deleting atable';
       DROP TABLE aTable;
END

CREATE TABLE aTable (
       aT_Id INT NOT NULL IDENTITY PRIMARY KEY,
       aT_String varchar(50)
);

IF OBJECT_ID('bTable') IS NOT NULL
BEGIN
       PRINT 'Deleting btable';
       DROP TABLE bTable;
END

CREATE TABLE bTable (
       bT_Id INT NOT NULL IDENTITY PRIMARY KEY,
       bT_aId INT NULL
);

ALTER TABLE bTable
ADD CONSTRAINT FK_aT_ID FOREIGN KEY (bT_aId) REFERENCES aTable(aT_Id);
```

How to handle errors in scripts

SQL Server 2005 introduced a TRY…CATCH statement to provide for error handling (also known as exception handling). When an error occurs in a statement within a TRY block, control is passed to the CATCH block where the error can be processed. If no error occurs in the TRY block, the CATCH block is skipped. Errors that have a severity of 10 or lower are considered warnings and are not handled by TRY…CATCH blocks. Errors that have a severity of 20 or higher and cause the database connection to be closed are not handled by TRY…CATCH blocks. Within a CATCH block, you can use ERROR functions to return data about the error that occurred. The syntax of the TRY…CATCH statement follows.

```
BEGIN TRY
```

```
     { sql_statement | statement_block }
END TRY
BEGIN CATCH
     [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Example usage.

```
BEGIN TRY
    INSERT Invoices
    VALUES (799, 'ZXK-799', '2008-07-01', 299.95)
    PRINT 'SUCCESS: Record was inserted.'
END TRY
BEGIN CATCH
    PRINT 'FAILURE: Record was not inserted.'
    PRINT 'Error ' + CONVERT(varchar, ERROR_NUMBER(), 1)
        + ': ' + ERROR_MESSAGE()
END CATCH
```

Transactions

A transaction is a sequence of one or more SQL operations that are treated as a unit. Specifically, each transaction appears to run in isolation, and furthermore, if the system fails, each transaction is either executed in its entirety or not all. The concept of transactions is actually motivated by two completely independent concerns. One has to do with concurrent access to the database by multiple clients and the other has to do with having a system that is resilient to system failures. Transaction support, what are known as the ACID properties of any DBMS:

A – atomicity
C – consistency
I – isolation
D – durability

Thus, a transaction is a group of database operations that are combined into a logical unit. By default, every SQL statement is treated as a separate transaction. However, you can combine any number of SQL statements into a single transaction. When you commit a transaction, the operations performed by the SQL statements become a permanent part of the database. Until it's committed, you can undo all of the changes made to the database since the beginning of the transaction by rolling back the transaction. A transaction is either committed or rolled back in its entirety. Once you commit a transaction, it cannot be rolled back.

Transactions can be used in conjunction with TRY-CATCH blocks to implement simple yet fool proof mechanisms to build and load data into a database (including a data warehouse database like your project.). Here's an example.

```
DECLARE @InvoiceID int
BEGIN TRY
       BEGIN TRAN

       INSERT Invoices
       VALUES (34,'ZXA-080','2008-08-30',14092.59,0,0,3)
```

```
        SET @InvoiceID = @@IDENTITY; -- @@IDENTITY returns the last identity value created

        INSERT InvoiceLineItems VALUES (@InvoiceID,1,160,4447.23,'HW upgrade');
        INSERT InvoiceLineItems VALUES (@InvoiceID,2,167,9645.36,'OS upgrade');

        COMMIT TRAN
        -- At this point, everything between the BEGIN TRAN and
        -- COMMIT TRAN is guaranteed to be in the DB.
END TRY

BEGIN CATCH
        ROLLBACK TRAN;
        -- At this point, nothing between the BEGIN TRAN and COMMIT TRAN will be in the
DB.
END CATCH
```

Exercise

Now it's your turn. Start with the T-SQL above that created a database (*MyDB*) and two tables (*aTable* and *bTable*). Now, use a transaction along with TRY-CATCH to insert a single row into *aTable* and single row into *bTable*. The T-SQL script immediately above is a very good "model" for this task. For the attribute `aT_String` use the string literal `'MyString'` or any other string you want.