University of Washington

iSchool Info 330

# Module 05 – Advanced SQL Programming

In this module, we will **continue** programming with the Structured Query Language (SQL), **this time looking at some advanced statements and options.**

## Outline

Here is a general outline of what we will be doing this module:

|  |
| --- |
| **Module05: Error Handling and Transaction Processing** |
| Session01 Lectures and Labs < 110 mins |
| Advanced Transactions - 20 |
| Lab 1: Processing Advanced Transactions - 20 |
| Transactional Stored Procedures - 40 |
| Lab 2: Creating Transactional Stored Procedures - 20 |
| Session02 - Lab |
| Lab 3: Creating a normalized database with abstraction layers - 50 |
| Session03 Lectures and Labs < 110 mins |
| Importing Data - 30 |
| Lab 4: Importing Data - 20 |
| Temporary Data Structures - 30 |
| Lab 5: Creating Temporary Tables - 20 |

**Note**: Times are only estimates and may change without notice!

# Session01 < 110 mins

In this session, we explore **how to perform transactions using advanced features**.

## Advanced Transactions - 20

"A transaction is a single unit of work. If a transaction is successful, all the data modifications made during the transaction are committed and become a permanent part of the database. **If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.**" (<https://technet.microsoft.com/en-us/library/ms174377(v=sql.110).aspx>, 2017)

Consider the following tables and note the constraints and the identity specification since these impact how you write your **Insert, Update, and Delete** statements.

use TempDB;

go

CREATE TABLE dbo.Contacts (

ContactId int Not Null Constraint pkContacts Primary Key IDENTITY,

FirstName varchar(100) Not Null,

LastName varchar(100) Not Null ,

EmailAddress varchar(100) Null Constraint uqContacts Unique,

);

go

CREATE TABLE dbo.ContactLog (

ContactLogId int Constraint pkContactLog Primary Key IDENTITY,

ContactDate datetime Not Null,

ContactID int Not Null,

Message varchar(8000) Not Null

);

Go

#### Insert Basics

Trying to **add an ID value on an Identity column will not work**!

Insert Into dbo.Contacts

(ContactID, FirstName, LastName, EmailAddress)

Values

(3, 'Tim', 'Thomas', 'TThomas@MyCo.Com');

go

When you add a new row to a table with an **Identity Option,** you can **see what the new ID is** using this system variable and this function:

Select @@Identity as [Last ID on the **current** connection]

,Ident\_Current('Products') as [Last ID for **any** connection]

You can add **multiple rows** at once like this:

Insert Into dbo.Contacts

(FirstName,LastName, EmailAddress)

Values

('Tim', 'Thomas', 'TThomas@MyCo.Com') ,

('Pat', 'Pruit', 'PPruit@MyCo.Com')

;

go

#### Inserting Data from Another Table

You can also add **rows from another table** using a select statement. **First**, create a select statement whose results match **the column list** of the insert (not by column name, but by column value).

Select

au\_fname as FirstName

, au\_lname as LastName

, null

From **Pubs.dbo.Authors**;

go

**Next**, you may have to **transform the data** using functions or operators!

Select

au\_fname as FirstName

, au\_lname as LastName

, SUBSTRING(au\_fname,1,1) + au\_lname + '@MyCo.com' as Email

From Pubs.dbo.Authors;

go

Finally, you can **Insert** the data into the table using you **transformed results**.

**Insert Into** Contacts

(FirstName, LastName, EmailAddress)

Select

au\_fname

, au\_lname

, SUBSTRING(au\_fname,1,1) + au\_lname + '@MyCo.com' as Email

From Pubs.dbo.Authors;

go

#### Inserting Dates and Times

Inserting **Dates and Times** is easy once you know how the **RDMS** wants you to **format** the entries. In SQL Server, it looks like this:

Insert Into dbo.ContactLog

(ContactDate,ContactID,[Message])

Values

('20170101 03:01:05', 1, 'Hey, Bob! How are things?'),

(GetDate(), 2, 'Hey, Sue! How are things?')

go

Select \* from dbo.ContactLog;

#### Using Transactional Statements

It is recommended that you use the **formal** transactional statements **Begin, Commit, or Rollback** when processing transactions. Here is an example:

Begin Transaction

Insert Into dbo.ContactLog

(ContactDate,ContactID, [Message])

Values

(GetDate(), 3, 'Hey, Tim! How are things?')

Commit Transaction

go

Select \* from dbo.ContactLog;

go

**If** you do something by **mistake**, you can undo it using a **Rollback** Transaction Statement like this:

Begin Tran --saction (you do not need the full spelling)

Insert Into dbo.ContactLog

(ContactDate,ContactID, [Message])

Values

(GetDate(), 3, 'Hey, Tim! How are things?')

-- Changed my mind!

Rollback Tran

go

Select \* from dbo.ContactLog;

go

This **only works if the transaction has not yet completed** (as it is when you use a Commit Transaction statement.) You can use the following command to **see if your transaction has Completed**:

Select @@TRANCOUNT; -- A count of 0 means that the transaction has completed (closed)

go

Note: If you **violate the table's constraints,** you will get errors. For example, trying to insert duplicate! This will **sometimes automatically roll the transaction back** for you, but you can always check with @@Trancount.

#### *Handling Errors*

When you perform a transaction, you should **use code to handle errors**. In MS SQL Server, it is best to use **Try-Catch blocks** for Error handling.

**Begin Try**

Begin Tran

Insert Into dbo.ContactLog

(ContactDate, ContactID, [Message])

Values

(GetDate(), 4, 'Hey, Pat! How are things?')

Commit Tran

**End Try**

**Begin Catch**

Rollback Transaction

**End Catch**

go

Select @@TRANCOUNT;

go

Select \* from dbo.ContactLog;

go

If there is an error, you can **add Custom Error Messages** to help someone understand the possible causes.

Begin Try

Begin Tran

Insert Into dbo.Contacts

(FirstName, LastName, EmailAddress)

Values

('Pat', 'Pruit', 'PPruit@MyCo.Com')

;

Commit Tran

End Try

Begin Catch

Rollback Transaction

**Print 'There was an Error! Please check the data you are entering!'**

**Print Error\_Message()** -- prints the original RDMS error message

End Catch

go

Select @@TRANCOUNT;

go

Select \* from dbo.Contacts;

go

#### Updates

Update statements allow you to **change existing data**. Here is a simple example:

Update Contacts

Set LastName = 'Smith'

Where ContactId = 2;

You can update **multiple columns** at a time like this:

Update Contacts

Set LastName = 'Smith'

,EmailAddress = 'SSmith@MyCo.com'

Where ContactId = 2;

Select @@ROWCOUNT;

And, **multiple rows** like this!

Begin Tran;

Update Contacts

Set LastName = 'Smith';

~~-- Where ContactId = 2;~~

**NOTE**: This is a really **BAD** thing to do by **accident**! You can add logic to protect yourself from this mistake like this:

Begin Tran;

Update Contacts

Set LastName = 'Smith';

~~-- Where ContactId = 2;~~

If (@@ROWCOUNT > 1) Rollback Tran

Else Commit Tran

Just like an Insert you can also **use a Try-Catch block for error handling** like this:

Begin Try

Begin Tran

Update Contacts

Set LastName = 'Smith';

~~-- Where ContactId = 2;~~

**If(@@ROWCOUNT > 1) RaisError('Do not change more than one row!', 15,1);**

Commit Tran

End Try

Begin Catch

Rollback Transaction

Print Error\_Message()

End Catch

go

Select @@TRANCOUNT

Select \* from dbo.Contacts;

go

#### Deletes

Delete statements allow you to **remove one or more rows**. Here is an example with transactions and error handling:

Begin Try

Begin Tran

Delete

From dbo.Contacts

-- **Where** ContactId = 2;

If(@@ROWCOUNT > 1) RaisError('Do not change more than one row!', 15,1);

Commit Tran

End Try

Begin Catch

Rollback Transaction

Print Error\_Message()

End Catch

go

Select \* from dbo.Contacts;

go

You can use **multiple transaction statements within one transaction**.

Begin Try

Begin Tran

Delete

From dbo.**ContactLog**

Where ContactID = 1;

Delete

From dbo.**Contacts**

Where ContactID = 1;

Commit Tran

End Try

Begin Catch

Rollback Transaction

Print Error\_Message()

End Catch

go

Select \* from dbo.Contacts;

Select \* from dbo.ContactLog;

Go

#### Question: What is this example doing, and why is it doing it like this?

## Lab 1: Processing Advanced Transactions - 20

In this lab, you create transaction statements, using a new Lab database.

You will work on your own for the first 10 minutes, then we will review the answers together in the last 10 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Create a Lab Database

Run the following SQL code into a code window, then modify it to use your own name instead of "YourNameHere." Afterward, execute the code to make the Module 05 labs database.

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

-- Title: Mod05 Labs Database

-- Author: YourNameHere

-- Desc: This file demonstrates how to process data in a database

-- Change Log: When,Who,What

-- 2017-01-01,YourNameHere,Created File

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

Use Master;

go

If Exists(Select Name from SysDatabases Where Name = 'MyLabsDB\_YourNameHere')

Begin

Alter Database [MyLabsDB\_YourNameHere] set Single\_user With Rollback Immediate;

Drop Database MyLabsDB\_YourNameHere;

End

go

Create Database MyLabsDB\_YourNameHere;

go

Use MyLabsDB\_YourNameHere;

go

-- Create Tables (Module 01)--

Create Table Categories

([CategoryID] [int] IDENTITY(1,1) NOT NULL

,[CategoryName] [nvarchar](100) NOT NULL

);

go

Create Table Products

([ProductID] [int] IDENTITY(1,1) NOT NULL

,[ProductName] [nvarchar](100) NOT NULL

,[CategoryID] [int] NULL

,[UnitPrice] [money] NOT NULL

);

go

-- Add Constraints (Module 02) --

Alter Table Categories

Add Constraint pkCategories

Primary Key (CategoryId);

go

Alter Table Categories

Add Constraint ukCategories

Unique (CategoryName);

go

Alter Table Products

Add Constraint pkProducts

Primary Key (ProductId);

go

Alter Table Products

Add Constraint ukProducts

Unique (ProductName);

go

Alter Table Products

Add Constraint fkProductsToCategories

Foreign Key (CategoryId) References Categories(CategoryId);

go

Alter Table Products

Add Constraint ckProductUnitPriceZeroOrHigher

Check (UnitPrice >= 0);

go

-- Show the Current data in the Categories, Products, and Inventories Tables

Select \* from Categories;

go

Select \* from Products;

go

### Step 2: Create SQL Transaction Statements

**NOTE:** This is the same lab we did in Module 3, but this time make sure to add the **Begin, Commit, and Rollback transaction statements with your Try-Catch block**!

**Question 1**: How would you add data to the Categories table?

**Question 2**: How would you add data to the Products table?

**Question 3**: How would you update data in the Products table?

**Question 4**: How would you delete data from the Categories table?

### Step 3: Review Your Work

Now, you will review your work with your instructor.

## Transactional Procedures - 30

Let's look at how professionals use stored procedures for processing transactions.

### Stored Procedures for Transaction

It is **considered a best practice to use Stored procedures for transaction processing**. You do this by creating transaction processing procedures like this one.

Create Proc **pIns**Customers

(@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

As

Begin

Begin Tran;

Insert Into Customers (CustomerID, CustomerFirstName, CustomerLastName, CustomerEmail)

Values (@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail);

Commit Tran;

End

go

Once the procedure is created it is stored in the database until it is removed and may be **executed** at any time **with code like this**:

Exec **pIns**Customers

@CustomerID = 1

,@CustomerFirstName = 'Sue'

,@CustomerLastName = 'Jones'

,@CustomerEmail = 'SJones@MyCo.com'

;

go

**Each** **table in a database should have an insert, update, and delete stored procedure** (**sproc or proc**), but select procedures are optional.

We already have an Insert sproc, so we create and execute an **Update sproc** like this:

Create Proc pUpdCustomers

(@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

As

Begin

Begin Tran

Update Customers

Set CustomerFirstName = @CustomerFirstName

,CustomerLastName = @CustomerLastName

,CustomerEmail = @CustomerEmail

Where CustomerID = @CustomerID;

Commit Tran

End

select \* from vCustomers

Exec pUpdCustomers

@CustomerID = 1

,@CustomerFirstName = 'Susan'

,@CustomerLastName = 'Jones'

,@CustomerEmail = 'SJones@MyCo.com'

;

go

And then a **Delete sproc** like this:

Create Proc **pDelCustomers**

(@CustomerID int

)

As

Begin

Begin Tran

Delete

From Customers

Where CustomerID = @CustomerID;

Commit Tran

End

go

#### Adding Try-Catch Error Handling

When Stored Procedures process data, they **should include Error Handling code**. Currently, our Stored Procedures only have transaction code. So, we would add a **Try-Catch block** around the existing code like this:

**Alter** Proc pInsCustomers

(@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

As

Begin

Begin Try

Begin Tran;

Insert Into Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

Values (@CustomerFirstName, @CustomerLastName, @CustomerEmail);

Commit Tran;

End Try

Begin Catch

Print 'There was an error. Common issues include: Duplicate Email Addresses!'

Print Error\_Number();

Print Error\_Message();

Rollback Tran;

End Catch

End

go

-- Now we test that the Sproc works (this code should cause an error!)

Exec pInsCustomers

@CustomerFirstName = 'Tim'

,@CustomerLastName = 'Thomas'

,@CustomerEmail = 'TThomas@MyCo.com';

go

#### Adding Return Codes

In addition to the Error Messaging, you **should include "Return Code"**. Return codes indicates the Status of your Sproc and are used by developers t**o troubleshoot and track the branches of logic begin processed**.

Alter Proc pInsCustomers

(@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

As

Begin

Declare @RC int = 0; -- Return Codes are Always an Integer!

Begin Try

Begin Tran;

Insert Into Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

Values (@CustomerFirstName, @CustomerLastName, @CustomerEmail);

Commit Tran;

Set @RC = +1; -- You can use any number you wish to, but I recommend using a positive one

End Try

Begin Catch

Print Error\_Number();

Print Error\_Message();

Set @RC = -1; -- Except when thing go wrong! For that I use a negative number

Rollback Tran;

End Catch

Return @RC; -- The Return Statement is always the last one in the Sproc!

End

go

Now we test that the Sproc works by capturing the Return Code in a variable like this:

Declare @Status int;

Exec @Status = pInsCustomers

@CustomerFirstName = 'Tim'

,@CustomerLastName = 'Thomas'

,@CustomerEmail = 'TThomasz@MyCo.com';

Select [The Return Code Was] = @Status;

go

Declare @Status int;

Exec @Status = pInsCustomers

@CustomerFirstName = 'Sue'

,@CustomerLastName = 'Jones'

,@CustomerEmail = 'SJones@MyCo.com';

Select [The Return Code Was] = @Status;

go

An Application Developer will often **use Return Codes to create their own Custom Error Messages.** Here is an example:

Declare @Status int;

Exec @Status = pInsCustomers

@CustomerFirstName = 'Sue'

,@CustomerLastName = 'Jones'

,@CustomerEmail = 'SJones@MyCo.com';

Select Case @Status

When +1 Then 'Insert was successful!'

When -1 Then 'Insert failed! Common Issues: Duplicate Data'

End as [Status]

go

#### Capturing a New Identity ID

Whenever you insert data into a table with an Identity option on its primary key, a new id is automatically generated. You can capture that ID in several ways, but let's consider how we can make use of that captured id.

Here is an example of capturing the new ID value created during an Insert to test all the transactions Stored Procedures (ins, upd, del) at once:

Declare @Status int;

Select \* From vCustomers;

-- Test Insert

Exec @Status = pInsCustomers

@CustomerFirstName = 'Jim'

,@CustomerLastName = 'James'

,@CustomerEmail = 'JJames@MyCo.com';

Select Case @Status

When +1 Then 'Insert was successful!'

When -1 Then 'Insert failed! Common Issues: Duplicate Data'

End as [Status];

Select \* From vCustomers Where CustomerID = @@IDENTITY;

go

-- Test Update

Declare @Status int;

Exec @Status = pUpdCustomers

@CustomerID = @@IDENTITY

,@CustomerFirstName = 'James'

,@CustomerLastName = 'James'

,@CustomerEmail = 'JJames@MyCo.com';

Select Case @Status

When +1 Then 'Update was successful!'

When -1 Then 'Update failed! Common Issues: Duplicate Data or Foreign Key Violation'

End as [Status]; -- Will be Null unless we add a Return Code to this Sproc!

Select \* From vCustomers Where CustomerID = @@IDENTITY;

go

-- Test Delete

Declare @Status int;

Exec @Status = pDelCustomers

@CustomerID = @@IDENTITY

Select Case @Status

When +1 Then 'Delete was successful!'

When -1 Then 'Delete failed! Common Issues: Foreign Key Violation'

End as [Status]; -- Will be Null unless we add a Return Code to this Sproc!

Select \* From vCustomers Where CustomerID = @@IDENTITY;

go

#### Creating a Stored Procedure Template

As your stored procedure code becomes more complex it is **important to be organized**! You should **create a template** like this one and use it to create all the sproc in your database. Here is an example:

Create Procedure <pTrnTableName>

(<@P1 int = 0>)

/\* Author: <YourNameHere>

\*\* Desc: Processes <Desc text>

\*\* Change Log: When,Who,What

\*\* <2020-01-01>,<Your Name Here>,Created Sproc.

\*/

AS

Begin -- Body

Declare @RC int = 0;

Begin Try

Begin Transaction;

-- Transaction Code --

Commit Transaction;

Set @RC = +1;

End Try

Begin Catch

If(@@Trancount > 0) Rollback Transaction;

Print Error\_Message();

Set @RC = -1

End Catch

Return @RC;

End -- Body

go

/\* **Informal** Testing Code:

Declare @Status int;

Exec @Status = pTrnTableName @P1 = 1;

Print @Status;

\*/

-- **Formal** Testing Code

Declare @Status int;

Exec @Status = <Sproc Name>

                <@P1 = 'Val1'>;

Select Case @Status

  When +1 Then 'Insert was successful!'

  When -1 Then 'Insert failed! Common Issues: Duplicate Data'

  End as [Status]

Select [The New ID Was:] = @@IDENTITY

Select \* From TableName;

go

## Lab 2: Creating Transactional Stored Procedures - 30

In this lab, you create and test advanced stored procedures using the 'MyLabsDB\_YourNameHere' database you created in Module05-Lab01.

You will work on your own for the first 20 minutes, and then we will review the answers together in the last 10 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Review the current data

Review the structure and data of the tables in the database.

Use Mod05\_LabsDB\_YourNameHere;

go

Select \* from Categories;

Exec sp\_help Categories;

### Step 2: Create SQL Transaction Stored Procedures

Use the provided stored procedure template to create code answering the following questions.

**Question 1**: How would you add data to the Categories table using a store procedure?

**Question 2**: How would you update data in the Categories table using a store procedure?

**Question 3**: How would you delete data from the Categories table using a store procedure?

### Step 3: Create SQL Transaction Stored Procedures

Modify and use formal testing code to test your stored procedures.

Declare @Status int;

Exec @Status = pInsCategories @CategoryName = 'CatC';

Select Case @Status

  When +1 Then 'Insert was successful!'

  When -1 Then 'Insert failed! Common Issues: Duplicate Data'

  End as [Status]

Select [The New ID Was:] = @@IDENTITY

Select \* From Categories;

### Step 4: Review Your Work

Now, you will review your work with your instructor.

# Session02 - Lab

## Lab 3: SQL Programming with Stored Procedures - 50

In this lab, you create and test advanced stored procedures using the 'MyLabsDB\_YourNameHere' database you created in Module05-Lab01. This lab is like the module's assignment and will give you a running start to complete it!

You will work on your own for the first 30 minutes, and then we will review the answers together in the last 20 minutes.

**Note**: This lab should be done **individually**.

### Step 1: Review the Lab Database

Review the current data and design of the Categories and Products tables.

Select \* From Categories;

go

Select \* From Products;

go

Exec sp\_help Categories;

go

Exec sp\_help Products;

go

### Step 2: Create SQL Transaction Stored Procedures

Use the provided stored procedure template to create one insert, one update, and one delete stored procedure for the Products table (as you did with the Categories table in Lab 2).

### Step 3: Test the Transaction Stored Procedures

Modify and use formal testing code to test your stored procedures.

Declare @Status int;

Exec @Status = pInsCategories @CategoryName = 'CatC';

Select Case @Status

  When +1 Then 'Insert was successful!'

  When -1 Then 'Insert failed! Common Issues: Duplicate Data'

  End as [Status]

Select [The New ID Was:] = @@IDENTITY

Select \* From Categories;

### Step 4: Review Your Work

Now, you will review your work with your instructor.

# Session03 Lectures and Labs < 110 mins

## Importing Data - 30

There are many ways to add data to a table beyond using the insert command. Here are a few examples

### Select Into

Importing data to tables with SELECT INTO **automatically creates a table** with the results of a query.

SELECT ProductName, UnitPrice AS Price, (UnitPrice \* 0.1) AS Tax

INTO NEWPriceTable

FROM Northwind.dbo.Products;

Go

SELECT \* FROM NewPriceTable;

Often SELECT INTO tables are **created to gather reporting or exporting data**.

SELECT DISTINCT CompanyName, Convert(Date, OrderDate) AS OrderDate

INTO TempDB.dbo.**OrdersReport** -- New Demo table

FROM Northwind.dbo.Orders INNER JOIN Northwind.dbo.Customers

ON Northwind.dbo.Orders.CustomerID = Northwind.dbo.Customers.CustomerID

Like a View, this can **turn a complex query into a simpler one**. Unlike a view, the **data is stored on the hard drive**!

SELECT \* FROM TempDB.dbo.**OrdersReport**;

You can **also** use SELECT INTO to create **"TEMP" tables** as shown here:

SELECT ProductName, UnitPrice AS Price, (UnitPrice \* 0.1) AS Tax

INTO **#**PriceTable

FROM Northwind.dbo.Products;

Go

SELECT \* FROM #PriceTable;

Temp Tables only exist for the life of a connection to the database. Once the connection is closed, the **temp table is deleted automatically**. Temp Tables with a single # can only be used by one connection. However, Temp Tables with a **double ##** can only be used by **many connections**.

SELECT ProductName, UnitPrice AS Price, (UnitPrice \* 0.1) AS Tax

INTO **##**PriceTable

FROM Northwind.dbo.Products;

Go

SELECT \* FROM **##**PriceTable -- TEST THIS IN A DIFFERENT QUERY WINDOW!

**Note**: We will look at more examples of temporary tables later in this module

### IMPORTING and EXPORTING with BCP

The BCP utility can both **EXPORT** and IMPORT data from data files and query results. Here is an example of how you would import data **from a table to a file**:

**NOTE:** This program must be installed separately with Azure Data Studio or SQL Server Management Studio**.**

**For Windows and Linux:**

<https://docs.microsoft.com/en-us/sql/tools/bcp-utility?view=sql-server-2017>

**For Mac:**

<https://cloudblogs.microsoft.com/sqlserver/2017/04/03/sql-server-command-line-tools-for-mac-preview-now-available/>

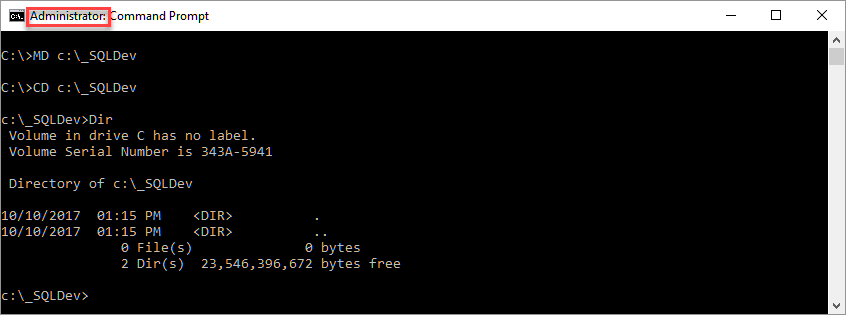
**However, you do not need it for any of the labs or assignments!**

**Step 1**: Make a **folder** for our work using Operating Systems Command Shell.

**MD** C:\\_SQLDev

**CD** C:\\_SQLDev

**Dir**

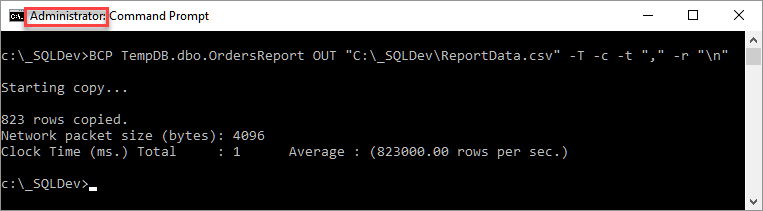


##### Figure 1: Creating a folder in the Windows command shell

**Tip:** Mac uses forward slashes and LS instead of Dir

**Step 2**: Run the Bulk Copy Program from Operating Systems Shell using the command **BCP**

BCP TempDB.dbo.**OrdersReport** **OUT** "C:\\_SQLDev\ReportData.csv" **-T -c -t "," -r "\n"**



##### Figure 2: Exporting data using BCP

Here is syntax for the BCP command:

" BCP {[[database\_name.][owner].]{table\_name | view\_name} | "query"}

{in | out | queryout | format} data\_file

[-m max\_errors] [-f format\_file] [-x] [-e err\_file]

[-F first\_row] [-L last\_row] [-b batch\_size]

[-n] [-c] [-w] [-N] [-V (60 | 65 | 70 | 80)] [-6]

[-q] [-C { ACP | OEM | RAW | code\_page } ] [-tfield\_term]

[-rrow\_term] [-i input\_file] [-o output\_file] [-a packet\_size]

[-S server\_name[\instance\_name]] [-U login\_id] [-P password]

[-T] [-v] [-R] [-k] [-E] [-h "hint [,...n]"] "

**-c** Performs the operation using a character data type

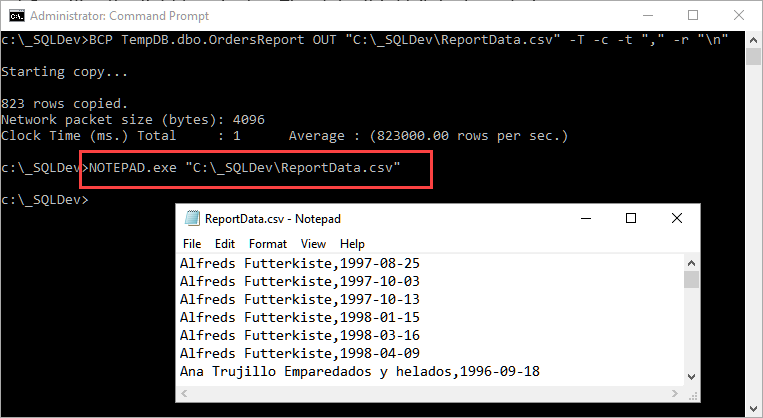
**-r** Specifies the row terminator. The default is \n (newline character).

**-t** Specifies the field terminator. The default is \t (tab character).

**-T** Specifies that the BCP utility connects to SQL Server with a trusted (Window's Security) connection." ([**http://msdn.microsoft.com/en-us/library/ms162802.aspx**](http://msdn.microsoft.com/en-us/library/ms162802.aspx), 2017)

**Step 3**: Review the **file**.

NOTEPAD.exe "C:\\_SQLDev\ReportData.csv"



##### Figure 3: The exported data file

**Note**: You cannot export data from a Single # table, but you can with ##'

You can **Import data** in a similar way, but **a table must exist before** you can fill it with data. SELECT INTO can be used to **create an empty table using a false where condition.** Here is an example:

SELECT DISTINCT CompanyName, OrderDate

INTO TempDB.dbo.**New**OrdersReport -- New data table

FROM Northwind.dbo.Orders INNER JOIN Northwind.dbo.Customers

ON Northwind.dbo.Orders.CustomerID = Northwind.dbo.Customers.CustomerID

WHERE 5 = 4; -- NOTE: THIS WILL NEVER BE TRUE and so create an empty table!!!

If you check the table will be empty:

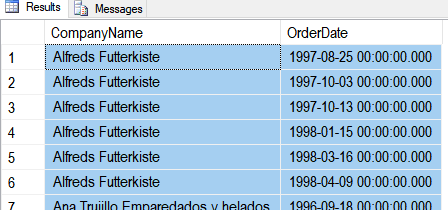
SELECT \* FROM TempDB.dbo.**New**OrdersReport;

To **import** data, you use **almost the same command**.

BCP TempDB.dbo.**New**OrdersReport **IN** "C:\\_SQLDev\ReportData.csv" **-T -c -t "," -r "\n"**

You can now check again and see if the data imported correctly:

SELECT \* FROM TempDB.dbo.**New**OrdersReport;



##### Figure 4: Data imported using BCP

### Back to Using Inserts

You often cannot import the data using BCP or another similar program since you do not have administrator access to the computer's command shell. The workaround is to import the data using one or more SQL Insert command.

Insert Into TempDB.dbo.NewOrdersReport

(CompanyName, OrderDate)

Values

('Alfreds Futterkiste','1998-04-09')

,('Ana Trujillo Emparedados y helados','1996-09-18')

## Lab 4: Importing Data - 20

In this lab, you import data to a new table using an Insert command and some data generated for you by the website, <https://www.mockaroo.com/>.

You will work on your own for the first 10 minutes, and then we will review the answers together in the last 10 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Create a new table

Use the following SQL code to create an import table.

Create Table Contacts

( ContactID int identity Primary Key

, ContactFirstName nvarchar(100)

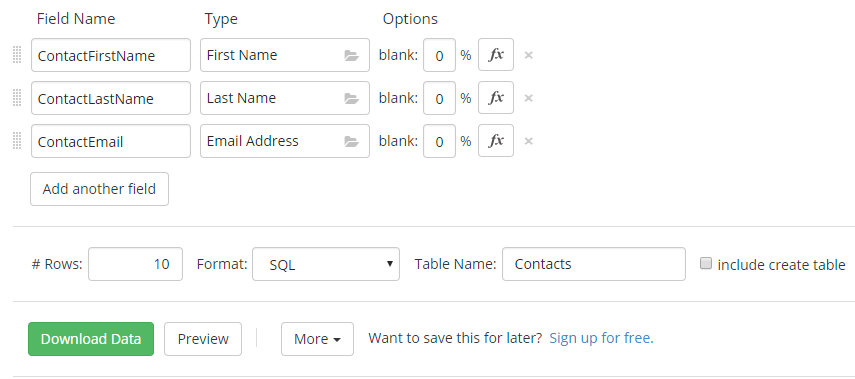
, ContactLastName nvarchar(100)

, ContactEmail nvarchar(100)

);

### Step 2: Generate some new data

Generate some fake data using the Mockaroo website. Configure the data to include a first name, last name, and email address. Generate only ten rows and have the output create a SQL format for a table named contacts.



##### Figure 5: Generating data on the Mockaroo website

### Step 3: Import data

Execute the insert commands generated by the Mockaroo website to fill the Contacts table.

## Temporary Data Structures - 30

"Temporary tables are just that. They are used most often to provide workspace for the intermediate results when processing data within a batch or procedure."( [https://www.red-gate.com/simple-talk/sql/t-sql-programming/temporary-tables-in-sql-server](https://www.red-gate.com/simple-talk/sql/t-sql-programming/temporary-tables-in-sql-server/), 2017)

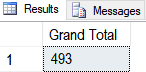
There are **several ways** you can create a table of data **to temporarily hold data** so that you can reuse its results.

### Subqueries

One of the first ways you learn to do this is though subqueries, where a result from a query is temporarily held in memory for use in an outer query. Here are some examples:

-- Query 1: Get the grand total of sale quantity

Select [Grand Total] = Sum(Sales.qty) From Sales;



##### Figure 6: The grand total of sales

-- Query 2: Compare the grand total to the totals by individual Stores with a subquery

Select

[State] = state

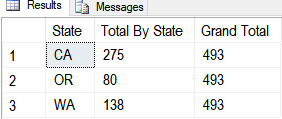
,[Total By State] = Sum(Sales.qty)

,[Grand Total] = (Select Sum(Sales.qty) From Sales)

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by state;



##### Figure 7: The sales totals by state

-- Query 3: Calculate the percentage by individual States with a subquery

Select

[State] = state

,[Total By State] = Sum(Sales.qty)

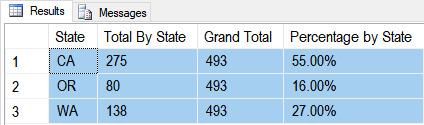
,[Grand Total] = (**Select** Sum(Sales.qty) From Sales)

,[Percentage by State] = str((100 \* Sum(Sales.qty)) / (**Select** Sum(Sales.qty) From Sales), 5, 2) + '%' -- Show 5 number with 2 after the decimal point!

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by state;



##### Figure 8: The sales totals by state with percentages

-- Query 4: Calculate the percentage by individual store with a subquery

Select

[Store] = stor\_name

,[State] = state

,[Total By Store] = Sum(Sales.qty)

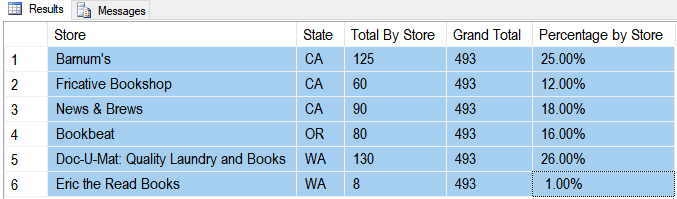
,[Grand Total] = (Select Sum(Sales.qty) From Sales)

,[Percentage by Store] = str((100 \* Sum(Sales.qty)) / (Select Sum(Sales.qty) From Sales), 5, 2) + '%' -- Show 5 number with 2 after the decimal point!

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by stor\_name, state;



##### Figure 9: The sales totals by store and state with percentages

### Temp Tables

Since queries like these can be complex and "scary" to work with you will often see developers use a temp table to store the subquery data and refer to it in another query. This can make your code easier to read!

-- Query 3: Calculate the percentage by individual state with a Temporary table

Select

[State] = state

,[Total By State] = Sum(Sales.qty)

,[Grand Total] = (Select Sum(Sales.qty) From Sales)

,[Percentage by State] = str((100 \* Sum(Sales.qty)) / (Select Sum(Sales.qty) From Sales), 5, 2) + '%'

Into #SalesByState

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by **state**;

-- Query 4: Calculate the percentage by individual store with a Temporary table

Select

[Store] = stor\_name

,[State] = state

,[Total By Store] = Sum(Sales.qty)

,[Grand Total] = (Select Sum(Sales.qty) From Sales)

,[Percentage by Store] = str((100 \* Sum(Sales.qty)) / (Select Sum(Sales.qty) From Sales), 5, 2) + '%'

Into #SalesByStore

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by **stor\_name**, **state**;

-- Query 5: Join the two temporary tables to create a complex report

Select

SS.Store

, SS.State

, SS.[Total By Store]

, S.[Total By State]

, S.[Grand Total]

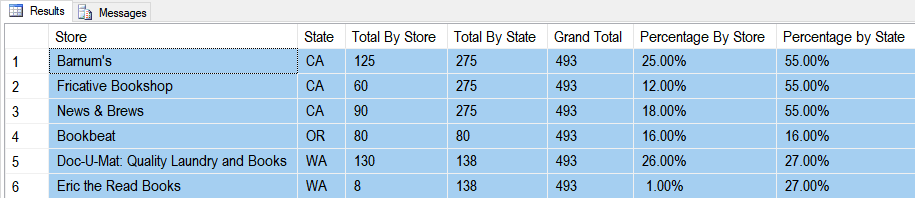
, SS.[Percentage By Store]

, S.[Percentage by State]

From #SalesByState as S

Join #SalesByStore as SS

on s.State = ss.State;



##### Figure 10: The combined data from #SalesByState and #SalesByStore

We can **drop** the Temp tables **explicitly** or just wait until our connection is closed!

If (Object\_ID('tempdb..#SalesByState') is not null) Drop Table #SalesByState;

If (Object\_ID('tempdb..#SalesByStore') is not null) Drop Table #SalesByStore;

**NOTE:** Tables made in the TempDB are automatically dropped when the SQL Server Service restarts, but temp tables are automatically dropped when the last connection using the table is closed.

### Common Table Expressions (CTEs)

CTEs are like Temp tables, but are delete once the query finishes! Here is an example:

With SalesByState AS

( -- Query 3: Calculate the percentage by individual state with a CTE

Select

[State] = state

,[Total By State] = Sum(Sales.qty)

,[Grand Total] = (Select Sum(Sales.qty) From Sales)

,[Percentage by State] = str((100 \* Sum(Sales.qty)) / (Select Sum(Sales.qty) From Sales), 5, 2) + '%'

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by state

)

, SalesByStore AS

( -- Query 4: Calculate the percentage by individual store with a CTE

Select

[Store] = stor\_name

,[State] = state

,[Total By Store] = Sum(Sales.qty)

,[Grand Total] = (Select Sum(Sales.qty) From Sales)

,[Percentage by Store] = str((100 \* Sum(Sales.qty)) / (Select Sum(Sales.qty) From Sales), 5, 2) + '%'

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by stor\_name, state

) -- Query 5: Join the two CTEs tables to create a complex report

Select

SS.Store

, SS.State

, SS.[Total By Store]

, S.[Total By State]

, S.[Grand Total]

, SS.[Percentage By Store]

, S.[Percentage by State]

From SalesByState as S

Join SalesByStore as SS

on s.State = ss.State;

### Variables

You can use a SQL variable to hold both a single (scalar) value of dataor a table of data. Here are some examples:

-- Scaler Variable --

Declare @GrandTotal as int -- Create

Select @GrandTotal = Sum(Sales.qty) From Sales; -- Fill

Select @GrandTotal; -- Use

-- Table Variable --

Declare **@StateReportData** Table ([State] Char(2) Unique, [Total By State] int) -- Create

Insert Into @StateReportData -- Fill

Select [State] = state,[Total By State] = Sum(Sales.qty)

From Sales Join Stores

On Stores.stor\_id = Sales.stor\_id

Group by state;

Select \* From @StateReportData; -- Use

-- Using Both --

Select

[State]

,[Total By State]

,[Grand Total] = @GrandTotal

,[Percentage by State] = str((100 \* [Total By State]) / @GrandTotal, 5, 2) + '%'

From @StateReportData

## Lab 5: Temporary Tables - 20

In this lab, you create a temporary table using the Northwind database.

You will work on your own for the first 15 minutes, and then we will review the answers together in the last 5 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Review the existing data

Run the following SQL code into a code window and review the results:

Select \* From Northwind.dbo.[Order Details];

Select \* From Northwind.dbo.[Products];

Select \* From Northwind.dbo.[Categories];

### Step 2: Get the total by Product Name with Category Id

Run the following SQL code into a code window and review the results:

Select CategoryID, ProductName, Sum(Quantity) as [Total By Product]

From Northwind.dbo.[Order Details] as OD

Join Northwind.dbo.[Products] as P

On OD.ProductID = P.ProductID

Group By P.CategoryID, P.ProductName

Order By 1,2;

### Step 3: Get the total by Category Name with Category Id

Run the following SQL code into a code window and review the results:

Select C.CategoryID, C.CategoryName, Sum(Quantity) as [Total By Category]

From Northwind.dbo.[Order Details] as OD

Join Northwind.dbo.[Products] as P

On OD.ProductID = P.ProductID

Join Northwind.dbo.[Categories] as C

On P.CategoryID = C.CategoryID

Group By C.CategoryID, C.CategoryName

Order By 1,2;

### Step 4: Create a Temp Table for Product Name with Category Id

Store the category Id, product name, and total product quantity in a temp table called #QtyByProduct

### Step 5: Create a Temp Table for Category Name with Category Id

Store the category id, category name, and total category quantity in a temp table called #QtyByCategory

### Step 6: Create a Temp Table for Category Name with Category Id

Join the two temporary tables to create a result showing the category, product, category total, and product totals as shown here:



##### Figure 11: The results of Mod05-Lab04 Step 6

### Step 7: Delete the temporary tables

Delete the temp tables using the following code:

If (Object\_ID('tempdb.#QtyByProduct') is not null) Drop Table #QtyByProduct;

If (Object\_ID('tempdb..#QtyByCategory') is not null) Drop Table #QtyByCategory;

***Note:*** *If you finish early, try using CTE or Variables instead of Temp Tables!*