University of Washington

iSchool Info 330

# Module 04 - SQL Programming

In this module, we will **continue programming with the Structured Query Language (SQL).**

## Outline

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

|  |
| --- |
| **Module04: SQL Programming** |
| Session01 Lectures and Labs < 110 mins |
| SQL Joins and Unions - 20 |
| Lab 1: Using Joins and Unions - 20 |
| Views - 20 |
| Lab 2: Creating Views – 20  Base Views - 10 |
| Custom Functions - 20 |
| Session02 - Lab |
| Lab 3: Creating Views and Functions - 50 |
| Session03 Lectures and Labs < 110 mins |
| Stored Procedures - 30 |
| Lab 4: Creating Stored Procedures - 30 |
| Variables - 10 |
| Loops and Conditionals - 10 |
| Lab 5: Creating KPIs - 10 |
| Dynamic SQL Code - 10 |

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

# Session01 < 110 mins

In this session, we explore **SQL Joins, Unions, and Views**.

## SQL Joins and Unions - 20

When using a relational database, data is divided into individual **tables based on subjects or events**. If you would like **to extract information on both subjects and events from multiple tables,** you can use a select statement with **a join clause**.

As an example, let's look at some information describing the sales of books. This data is recorded in the Pubs database. And can be **retrieved from two tables:** Titles and Sales.

Use Pubs;

Select \* From Titles;

Select \* From Sales;

go

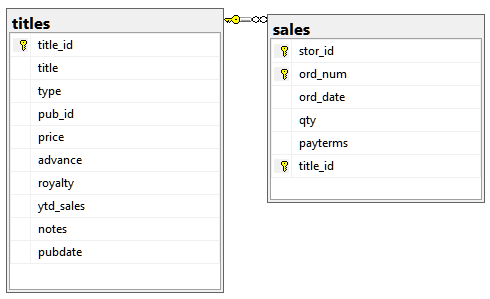


Figure: The titles and sales table of the Pubs database

When we **extract** **information** we often **focus on just the data we want** to look at by listing the only **the columns** we want.

Use Pubs;

Select title\_id, title

From Titles

Order by title\_id;

Select title\_id, ord\_date, qty

From Sales

Order by title\_id;

go

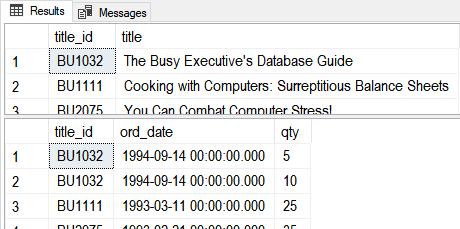


Figure: The results of the previous SQL statement

We can combine (or **join**) these five columns of **data into one result set** like this:

Select Titles.title\_Id, title, ord\_date, qty

From Titles, Sales

Where Titles.title\_id = sales.title\_id;

go

In 1992, the American National Standards Institute (**ANSI**) asked people to **change to this syntax** to this:

Select Titles.title\_Id, title, ord\_date, qty

From Titles **Join** Sales

**On** Titles.title\_id **=** sales.title\_id;

go

Or this more **formal** version:

Select Titles.title\_Id, title, ord\_date, qty

From Titles **Inner Join** Sales

On Titles.title\_id **=** sales.title\_id;

go

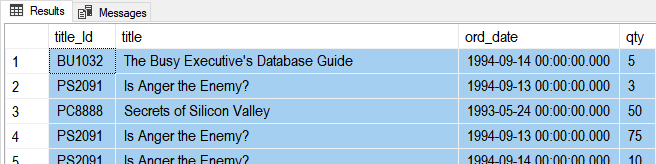


Figure: The results of the previous SQL statements

If you want to combine results from **more than two tables** you just add on another Join clause.

Select Stores.stor\_id, stor\_name, Titles.title\_Id, title, ord\_date, qty

From **Titles**

**Inner Join** **Sales**

**On** Titles.title\_id = Sales.title\_id

**Inner Join** **Stores**

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

go

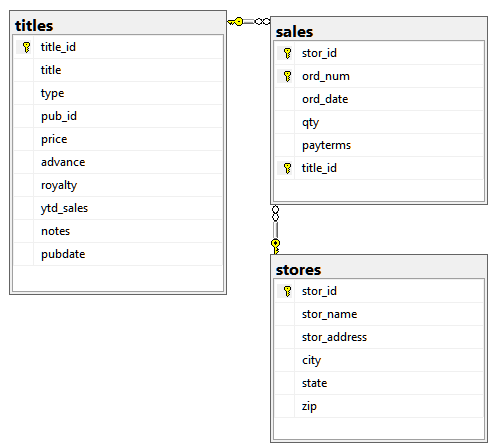
**

Figure: The Titles, Sales, and Stores tables of the Pubs database

Using **table** **aliases** may make your code more readable.

Select St.stor\_id, stor\_name, T.title\_Id, title, ord\_date, qty

From Titles as T

Inner Join Sales as S

On T.title\_id = S.title\_id

Inner Join Stores as St

On S.stor\_id = St.stor\_id;

go

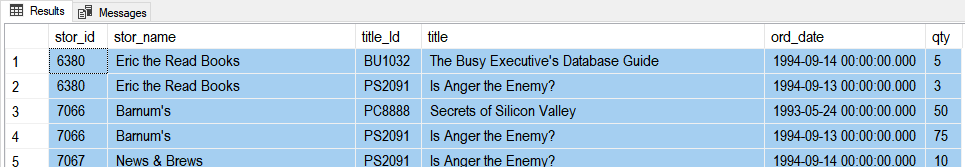


Figure: The results of the previous SQL statements

-- Using **column aliases** may make your results more readable.

Select

[Store ID] = St.stor\_id

,[Store Name] = stor\_name

,[Title ID] = T.title\_Id

,[Title] = title

,[Sales Date] = ord\_date

,[Sales Quantity] = qty

From Titles as T

Inner Join Sales as S

On T.title\_id = S.title\_id

Inner Join Stores as St

On S.stor\_id = St.stor\_id;

go

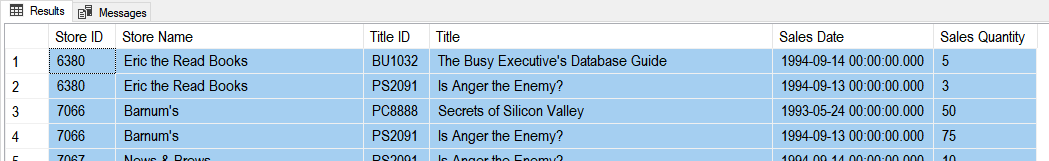


Figure: The results of the previous SQL statements

Sometimes you want **data from tables that are not directly connected**. **Most often when there is a many to many relationships** between the data. In this case you just connect 'Bridge Table' to each of the tables.

Select

[Authors Name] = A.au\_fname + ' ' + A.au\_lname

,[Title] = title

From **Authors** as A

**Inner Join TitleAuthor** as TA

On A.au\_id = TA.au\_id

**Inner Join** **Titles** as T

On T.title\_id = TA.title\_id

go

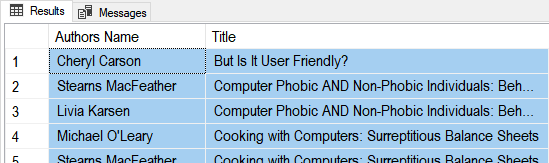


Figure: The results of the previous SQL statements

It can also happen between table that are connected through **a long chain of tables**.

Select

[Store Name] = stor\_name

,[Author Name] = A.au\_fname + ' ' + A.au\_lname

From Authors as A

Inner Join TitleAuthor as TA

On A.au\_id = TA.au\_id

Inner Join Titles as T

On T.title\_id = TA.title\_id

Inner Join Sales as S

On T.title\_id = S.title\_id

Inner Join Stores as St

On S.stor\_id = St.stor\_id;

go

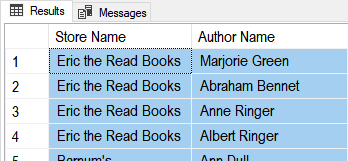


Figure: The results of the previous SQL statements

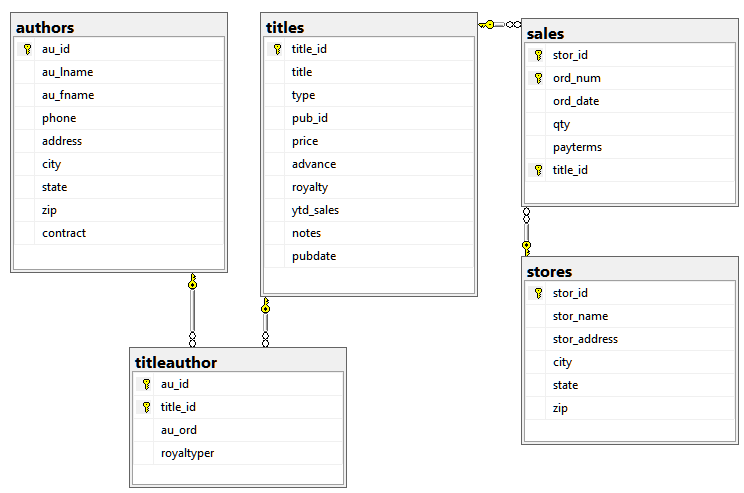
**

Figure: The Authors, TitleAuthor, Titles, Sales, and Stores tables in the Pubs database

#### Question: Why would you want to join data from so many different tables?

### Helpful Technique

Joins are **easiest created** by using this **technique**:

1. **List** the **columns** you want
2. **List** the **tables** that have those columns
3. **List** **how** those these columns are **connected**
4. **Use** these **ingredients** to **create your SQL Join**!

-- 1) I want a **list** of publisher names the titles they publish so here are my **columns**

pub\_name title

-- 2) **List** the **tables** that have those columns

Publishers Titles

-- 3) **List** **how** those these **columns** are **connected**

Titles.**pub\_id** Publishers.**pub\_id**

-- 4) **Use** these **ingredients** **to** **create** your SQL Join!

Select pub\_name, title

From Publishers Join Titles

On Titles.**pub\_id** = Publishers.**pub\_id**;

#### Question: What is the hardest part of creating a SQL Join?

### Outer Joins

Sometimes you want **all the data** from one or more tables, **even if you do not have a matching value** between them. You can display this non-matching data using an Outer Join.

-- Use an **Inner** Join to see **only** publishers **with matching** titles

**Use pubs;**

Select pub\_name, title

From Publishers **Inner** Join Titles

On Titles.pub\_id = Publishers.pub\_id

Order By pub\_Name;

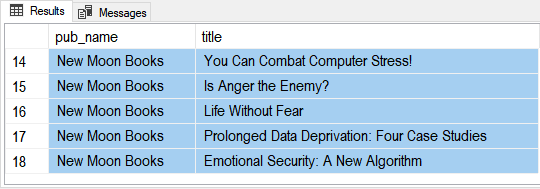


Figure: The results of the previous SQL statement

-- Use an **OUTER** Join to see **All** publishers **with or without matching** titles.

Select pub\_name, title

From Publishers Left Outer Join Titles

On Publishers.pub\_id = Titles.pub\_id

Order By pub\_Name;

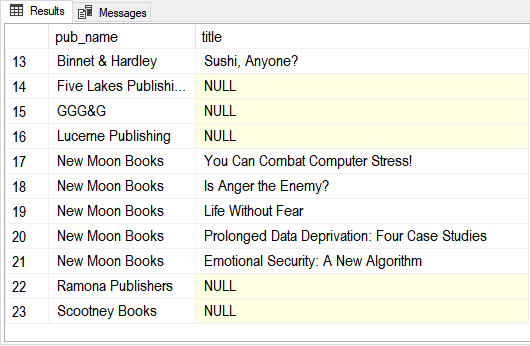


Figure: The results of the previous SQL statement

-- Use a join to see information about **All titles with or without matching publishers**.

Select pub\_name, title

From Publishers **RIGHT** OUTER Join **Titles**

On Titles.pub\_id = Publishers.pub\_id;

**Note**: For this example, I added a new title called "TestTitle" into the titles table without a publisher ID.

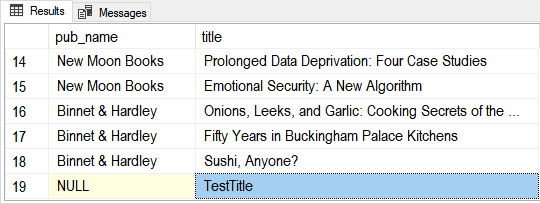


Figure: The results of the previous SQL statement

-- Use a join to see information about **All publishers, but only those without titles**.

Select pub\_name, title

From Titles

**RIGHT** OUTER Join Publishers

On Titles.pub\_id = Publishers.pub\_id

Where title **is Null**;

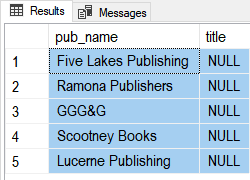


Figure: The results of the previous SQL statement

### Cross Join

Cross Joins give you **all the possible combinations of values**. While not used often, it is still a "nice to have" feature in an RDMS. Here is an example.

Select pub\_name, title

From Titles **CROSS** Join Publishers;

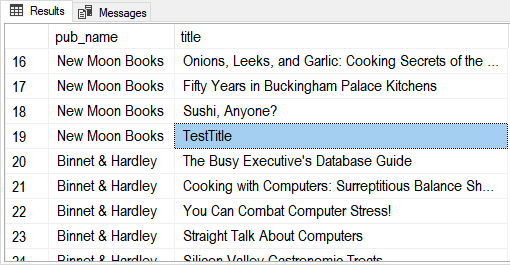


Figure: The results of the previous SQL statement

### Self Joins

A Self Join is any type of join (inner, outer, or cross) that connects to the same table more than once. Here are some examples:

Use Northwind;

-- Here is a table that is self-referencing

Select \* from Employees;

-- There is a FK on the Reports too column that references the EmployeeID column

Select ReportsTo, \* from Employees;

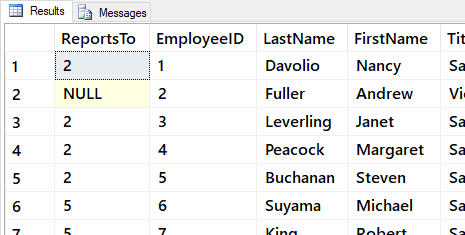


Figure: The results of the previous SQL statement

-- We can select the Name of a manager, but join the RESULTS of table twice!

Select Mgr.EmployeeId as [MgrID], Mgr.LastName, Emp.EmployeeID, Emp.LastName

From Employees as Emp Inner Join Employees Mgr

On Emp.ReportsTo = Mgr.EmployeeID

Order By 1,2,3,4;

go

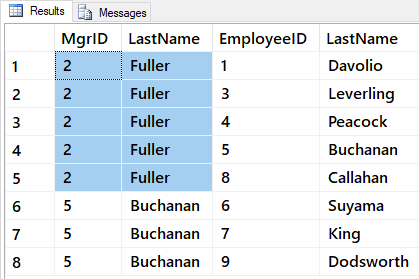


Figure: The results of the previous SQL statement

As you can see, a Self-join is just as easy to do as any other join, but there is one "gotcha!" Be careful about how you order the On clause. Here is an example of a query that is almost the same, but not exactly the same:

-- Note, this is **not** the same as ...

Select Mgr.EmployeeId as [MgrID], Mgr.LastName, Emp.EmployeeID, Emp.LastName

From Employees as Emp Inner Join Employees Mgr

On Emp.EmployeeID = Mgr.ReportsTo --< LOOK HERE

-- Wrong because, you need to look up the MANAGER's ID for EACH EMPLOYEE Row, NOT the Employee's ID

Order By 1,2,3,4;

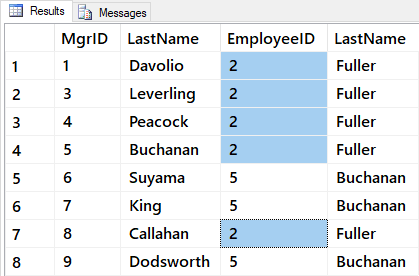


Figure: The results of the previous SQL statement

Self-joins that use the Outer Join will show non-matching values (just as they always do!). Notice the different results of the following query:

Select Mgr.EmployeeId, Mgr.LastName, Emp.EmployeeID, Emp.LastName

From Employees as Emp

**Left** Join Employees Mgr

On Emp.ReportsTo = Mgr.EmployeeID

Order By 1,2,3,4;

go

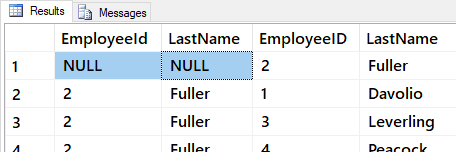


Figure: The results of the previous SQL statement

We can use the various functions and column aliases to make is look better!

Select

[Manager ID] = IsNull(Mgr.EmployeeId, 0)

,[Manager] = IIF(IsNull(Mgr.EmployeeId, 0) = 0, 'General Manager', Mgr.LastName)

,[Employee ID] = Emp.EmployeeID

,[Employee Name] = Emp.FirstName + ' ' + Emp.LastName

From Employees as Emp

Left Join Employees Mgr

On Emp.ReportsTo = Mgr.EmployeeID

Order By 1,3;

go

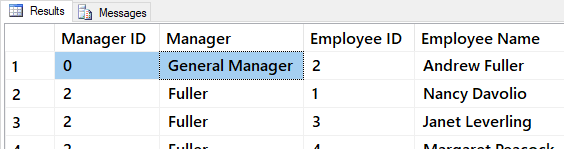


Figure: The results of the previous SQL statement

### Unions

While Joins combine columns from results, **Unions combine ROWs from multiple result sets into one result set**. Here is an example:

Use Pubs;

Select \* From Stores; -- 6 rows

Select \* From Authors; -- 23 rows

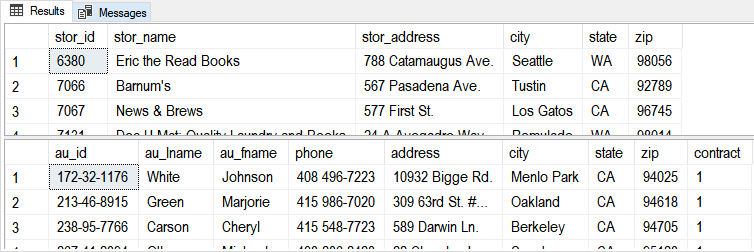


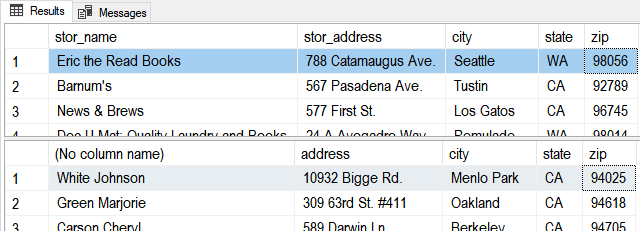
Figure: The results of the previous SQL statements

As an example, let consider how you might make a mailing list result using these two tables:

-- 1) Here is our data in two result sets

Select stor\_name, stor\_address, city, state, zip From Stores;

Select au\_lname + ' ' + au\_fname, address, city, state, zip From Authors;



*Figure: The results of the previous SQL statements*

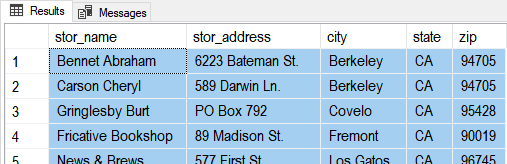
-- 2) We use Union to combine the two result sets into a single result set

Select stor\_name, stor\_address, city, state, zip From Stores

Union

Select au\_lname + ' ' + au\_fname, address, city, state, zip From Authors

Order By 4,3,1



*Figure: The results of the previous SQL statement*

-- 3) We use Aliases to make the results look better!

Select

[Name] = stor\_name

,[Address] = stor\_address

,[City] = city

,[State] = state

,[ZipCode] = zip

From Stores

Union

Select au\_lname + ' ' + au\_fname, address, city, state, zip From Authors

Order By 4,3,1

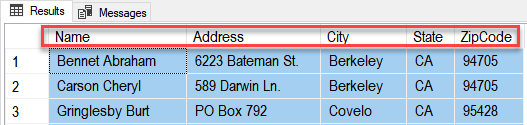


Figure: The results of the previous SQL statement

## Lab 1: Using Joins and Unions - 30

In this lab, you create some advanced select statements using the Northwind database.

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.

### Step 1: Review Database Tables

Run the following code in a SQL query editor and review the names of the tables.

Select \* From Northwind.Sys.Tables Where type = 'u' Order By Name;

### Step 2: Create Queries

Answer the following questions by writing and executing SQL code.

**Question 1**: How can you show a list of **category** names? Order the result by the category!

**Question 2**: How can you show a list of **product** names and the **price** of each product? Order the result by the product!

**Question 3**: How can you show a list of **category** and **product** names, and the **price** of each product? Order the result by the category and product!

**Question 4**: How can you show a list of **order Ids**, **category** names, **product** names, and order **quantities.** Sort the results by the Order Ids, category, product, and quantity!

**Question 5**: How can you show a list of **order ids**, **order date**, **category** names, **product** names, and order **quantities** the results by the order id, order date, category, product, and quantity!

### Step 3: Review Your Work

Now, you will review your work with your instructor.

**NOTE: Unlike assignments, labs do not need to be turned in to Canvas!**

## Views - 20

As your **SQL** select **statements** become more complex, you may decide to **save** them **in** a **text** **file** for reuse and use them repeatedly. (A text file with SQL code in it is **called a Script**!)

**Alternatively**, you can **save your select statements in a database itself as a view**. SQL Views are Named select statements stored in a database.

Here is an **example** of **a view** that **stores a complex select statement**. The select statement includes a join, column aliases, concatenation, and the choose function.

Create -- Drop

View vAuthorsByTitles

AS

Select

[Title] = T.title

,[Author]= A.au\_fname + ' ' + A.au\_lname

,[Order On Title] = Choose(TA.au\_ord, '1st', '2nd', '3rd')

From pubs.dbo.titles as T

Join pubs.dbo.titleauthor as TA

On T.title\_id = TA.title\_id

Join pubs.dbo.authors as A

On TA.au\_id = A.au\_id

go



Figure: The results of the previous SQL statement

However, we can **run** the **complex** view code **using** only a **simple Select statement**!

Select **\* From** vAuthorsByTitles;

go

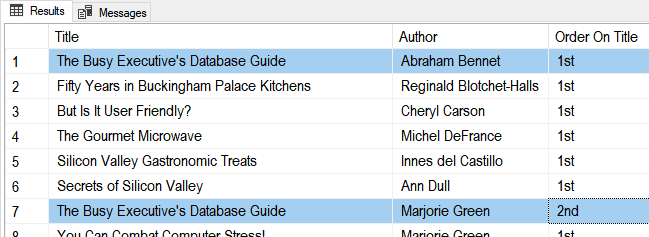


Figure: The results of the previous SQL statement

As you can see, creating a **view makes** using **complex** **SQL code** much **easier to manage and use**.

### Views and the Order By Clause

**By** **design**, **views cannot** **dictate how that data is sorted**. In some RDMS you can **use the TOP clause** to **trick** the system into allowing the Order By clause like this example:

Alter View vAuthorsByTitles

AS

Select **TOP 1000000000** --< You Need this,

[Title] = T.title

,[Author]= A.au\_fname + ' ' + A.au\_lname

,[Order On Title] = Choose(TA.au\_ord, '1st', '2nd', '3rd')

From pubs.dbo.titles as T

Join pubs.dbo.titleauthor as TA

On T.title\_id = TA.title\_id

Join pubs.dbo.authors as A

On TA.au\_id = A.au\_id

**Order By T.title, TA.au\_ord;** --< if you want to use this!

go

go

Select \* from vAuthorsByTitles

**NOTE**: This MAY not be considered a best practice. Instead, you can always use the Order By clause when you select from the View like this:

Select \* from vAuthorsByTitles Order By Title, Author;

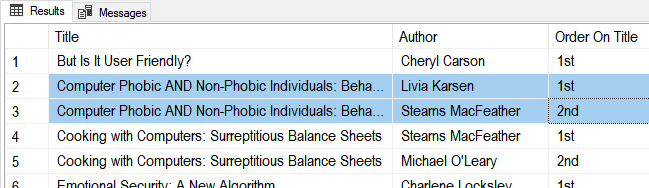


Figure: The results of the previous SQL statement

### Partitioning Data with Views

Views will also allow you to **split data by rows (horizontal partitioning)** stored in one table (At least from a visual aspect). You can divide data by rows by **using a Where clause**.

Select \* from Pubs.dbo.Sales;

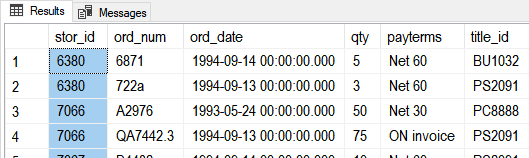


Figure: The results of the previous SQL statement

Create View **Store6380Sales** AS

Select St.stor\_name as Store, S.\*

From Pubs.dbo.Sales as S Join Pubs.dbo.stores as St

On S.stor\_id = St.stor\_id **Where S.stor\_id = 6380**;

go

Create View **Store7066Sales** AS

Select St.stor\_name as Store, S.\*

From Pubs.dbo.Sales as S Join Pubs.dbo.stores as St

On S.stor\_id = St.stor\_id **Where S.stor\_id = 7066**;

go

Select \* From Store6380Sales;

Select \* From Store7066Sales;

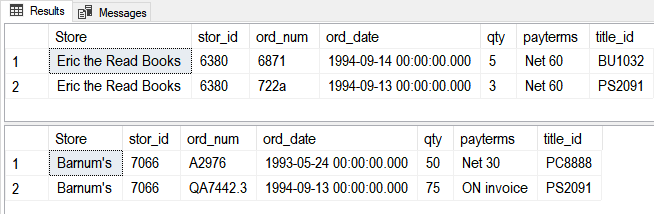


Figure: The results of the previous SQL statements

You can **divide data by Columns** **(vertical partitioning)** too, like this:

Select \* From Northwind.dbo.Employees;

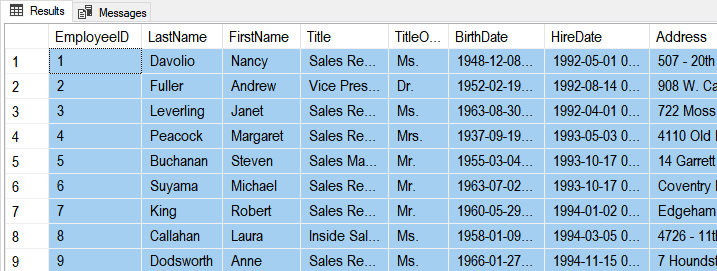


Figure: The results of the previous SQL statement

Create View **vPublicEmployeeInfo**

As

Select

TitleOfCourtesy

,FirstName

,LastName

,Title

From Northwind.dbo.Employees;

go

Create View **vPrivateEmployeeInfo**

As

Select

EmployeeID

,LastName

,FirstName

,Title

,TitleOfCourtesy

,BirthDate

,HireDate

,Address

,City

,Region

,PostalCode

,Country

From Northwind.dbo.Employees;

Go

Select \* From vPublicEmployeeInfo;

Select \* From vPrivateEmployeeInfo;

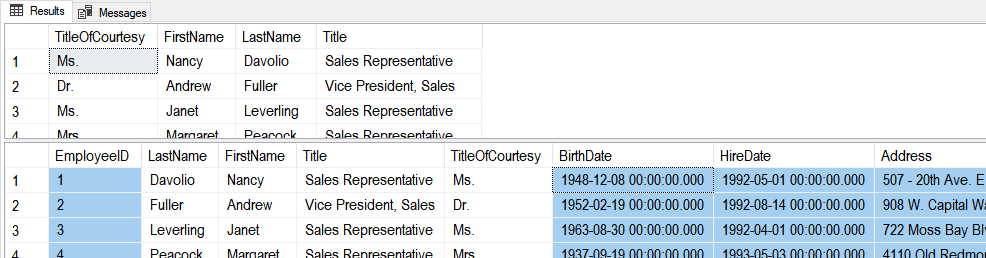


Figure: The results of the previous SQL statement

Now you would **protect the private data with permissions** and allow people to only access that employees table data through the appropriate views.

Use Northwind;

**Deny** Select On Employees to Public;

**Deny** Select vPrivateEmployeeInfo to Public;

**Grant** Select On vPublicEmployeeInfo to Public;

## Lab 2: Creating Views - 20

In this lab, you create views using the Northwind database.

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 can be done individually or with a group of up to 3 people.

### Step 1: Review Database Tables

Run the following code in a SQL query editor and review the names of the tables.

Select \* From Northwind.Sys.Tables Where type = 'u' Order By Name;

### Step 2: Create a Lab Database

Create a new database for this lab called MyLabsDB\_YourNameHere (using your own name, of course!) Modify and use the following code to accomplish this:

Create Database [MyLabsDB\_YourNameHere];

go

Use [MyLabsDB\_YourNameHere];

### Step 3: Create a Query

Answer the following questions by writing and executing SQL code.

**Question 1**: How can you create a view to show a list of **customers** names and their **locations. Use the IsNull()** function to display null region names as the name of the customer's country? Call the view **vCustomersByLocation.**

Select \* From vCustomersByLocation Order By CustomerName;

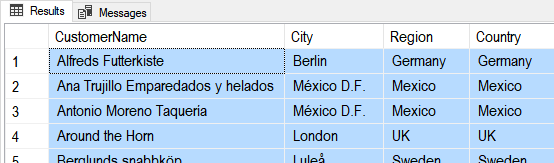


Figure: The query results of question 2-1

**Question 2**: How can you create a view to show a list of **customer** names, their **locations**, and the **number of orders** they have placed (hint: use the **count()** function)? Call the view **vNumberOfCustomerOrdersByLocation**.

Select \* From vNumberOfCustomerOrdersByLocation Order By CustomerName;



Figure: The query results of question 2-2

**Question 3**: How can you create a view to shows a list of **customer** names, their **locations**, and the **number of orders** they have placed (hint: use the **count()** function) on a given **year** (hint: use the **year()** function)? Call the view **vNumberOfCustomerOrdersByLocationAndYears**.

Select \* From vNumberOfCustomerOrdersByLocationAndYears Order By CustomerName, OrderYear;



Figure: The query results of question 2-3

### Step 4: Review Your Work

Now, you will review your work with your instructor.

**NOTE: Unlike assignments, labs do not need to be turned in to Canvas!**

## Creating Base Views - 10

In addition to any reporting views you create, **each table** in a database **should** **have a "Base" view** to show data from that table. When you **make a table**, you **create a base view** and then **restrict access to the table** while **allowing access to the view**. This forces people to use your data in the "Abstract."

Create -- Drop

Table tblCustomers -- Note: "tbl" is a common prefix in some databases

(CustomerID int Identity Primary Key, CustomerName nVarchar(100));

go

Insert Into tblCustomers (CustomerName) Values ('Bob Smith'),('Sue Jones');

go

-- Make a matching view!

Create View Customers

AS

Select CustomerID, CustomerName From dbo.tblCustomers;

go

-- Without a prefix in the name, like vCustomers, people may think your view is a table!

Select \* from Customers;

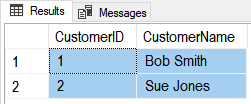


Figure: The results of the previous SQL statement

**Abstraction layer objects,** like views, **make changes to a table design** **easier.** They help you maintain the way applications access the data. When a database table needs to be changed, **applications** that use the view can **continue to work, if** **you** **modify the code in the view to hide those changes**.

-- Step 1) Make a backup code of the data and drop the existing table

Select \* Into #TempCustomers from tblCustomers;

-- Step 2) Make the changes to the original table and replace its data

-- 2.1

Drop Table tblCustomers;

go

-- 2.2

Create Table tblCustomers

( CustomerID int Identity Primary Key

, CustomerFirstName nVarchar(100)

, CustomerLastName nVarchar(100)

);

go

-- 2.3

Insert Into tblCustomers (CustomerFirstName, CustomerLastName)

Select

CustomerFirstName = Substring(CustomerName, 1,3) -- use a function to split the data

,CustomerLastName = Substring(CustomerName, 4,100) -- use a function to split the data

From #TempCustomers;

go

-- Step 3) Make it look like the data was not split!

Alter

View Customers

AS

Select CustomerID, CustomerName = CustomerFirstName + ' ' + CustomerLastName

From dbo.tblCustomers;

go

-- Step 5) Also, add a new view to show the new design changes

Create

View CustomersNormalized

AS

Select CustomerID, CustomerFirstName, CustomerLastName

From dbo.tblCustomers;

go

-- Step 6) You set permissions to force developers into using the view and not the actual table

Deny Select On tblCustomers to Public;

Grant Select On Customers to Public;

-- Step 7) Verify the changes

Select \* From tblCustomers;

Select \* From Customers;

Select \* From CustomersNormalized;

go

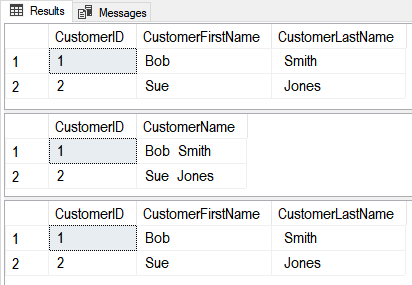


Figure: The results of the previous SQL statements

### Schema Binding

Foreign Keys will not protect a Parent table from being Dropped, so some developers will create Views using the Schema Binding option which **stop tables from changing so much the view does not work anymore**!

Use Northwind;

go

Create -- Drop

View vCategories

WITH SCHEMABINDING -- this Requires you to use the table's 2-part name!

AS

Select CategoryID, CategoryName as [CatName] From dbo.Categories; --<< 2-part name

go

Select \* From Categories;

Select \* From vCategories;

go

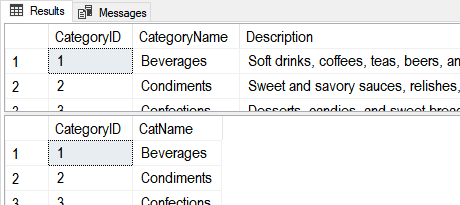


Figure: The results of the previous SQL statements

If a table's **changes will not break its View**, then the change **is allowed**...

Alter Table Categories Add IsDiscontinued int; -- SYNTAX: Don't use the word "Column"

go

Select \* From Categories;

Select \* From vCategories;

go

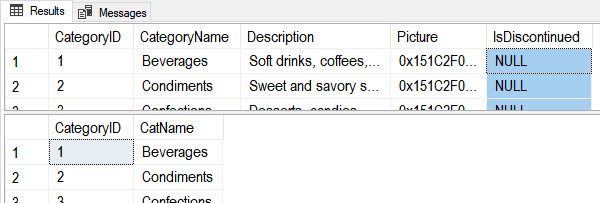


Figure: The results of the previous SQL statements

..., but **changes that would break the table's views** are **not** **allowed**.

Alter Table Categories Drop Column CategoryName; -- SYNTAX: Use the word "Column"

go

Msg 5074, Level 16, State 1, Line 15

The object 'vCategories' is dependent on column 'CategoryName'.

Msg 5074, Level 16, State 1, Line 15

The index 'CategoryName' is dependent on column 'CategoryName'.

Msg 4922, Level 16, State 9, Line 15

ALTER TABLE DROP COLUMN CategoryName failed because one or more objects access this column.

Nor can you **cannot drop the whole table**!

Alter Table Products Drop Constraint [FK\_Products\_Categories];

Drop Table Categories;

Msg 3729, Level 16, State 1, Line 26

Cannot DROP TABLE 'Categories' because it is being referenced by object 'vCategories'.

## Custom Functions - 10

In addition to SQL Server's built-in functions, you can create your own custom functions. These are often called User Defined Functions or just **UDFs**.

Functions and Views similar. Here are **two examples that show how similar**:

-- View

Create **View** vProducts

AS

Select ProductID, ProductName,CurrentPrice = UnitPrice, CategoryID, Discontinued

From Northwind.dbo.Products;

go

**Select \* from vProducts;** -- 77rows

Go

-- Function

Create **Function** fProducts()

Returns Table

AS

Return(

Select ProductID, ProductName, CategoryId, Discontinued

From Northwind.dbo.Products

);

go

**Select \* from fProducts();** -- 77rows

go

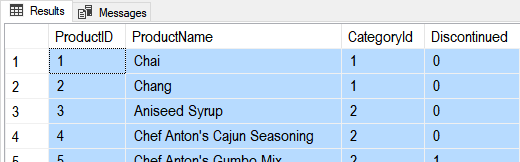


Figure: The results of the previous SQL statements

### Functions with Parameters

**Unlike views, function can use parameters** to change the results of the query as it is executed like this:

Alter Function fProducts(@CategoryId int)

Returns Table

AS

Return(

Select ProductID, ProductName

From Northwind.dbo.Products

Where CategoryID = @CategoryId

);

go

Select \* From fProducts(**1**); -- 12 rows

go

While this **may seem like a big advantage**, **but** remember that you can always apply a where clause when using a View like this:

Select \* From vProducts Where CategoryID = 1; -- 12 rows



Figure: The results of the previous SQL statements

**TIP**: Since table functions are more complex and provide similar functionality, **use Views to 'Keep It Simple' when you can!**

### Scalars Functions

Unlike views, you can create **UDFs to return a single (scalar)** **value as an expression**. (Note: In MS SQL you **must use include the schema name in scaler UDFs**, in this case, dbo).

Create Function **dbo**.**MultiplyValues**(@Value1 Float, @Value2 Float)

Returns Float

As

Begin

Return(Select @Value1 \* @Value2);

End

go

-- Calling the function

Select **dbo**.**MultiplyValues**(4, 5);

go

If you want **to apply the function to each row** of a result set, you use the new function like this:

Create table Tempdb.dbo.SalesDetails

(SalesId int, SalesLineItemId int, ProductId int, SalesPrice money, SalesQty int,

Primary key(SalesId, SalesLineItemID)

);

go

Insert Into Tempdb.dbo.SalesDetails (SalesId,SalesLineItemId,ProductId,SalesPrice,SalesQty)

Values (1,1,100,$9.99,10),(1,2,200,$1.00,5)

Go

**Select**

SalesId

,SalesLineItemId

,ProductId

,SalesPrice

,SalesQty

,**dbo**.**MultiplyValues**(SalesPrice,SalesQty) as ExtendedPrice

**From dbo.SalesDetails**

Here are the results:

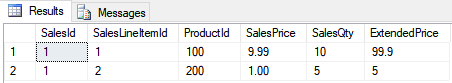


Figure 5: Results of using the MultiplyValues function in a query

# Session02 - Lab

## Lab 3: Creating Views and Functions - 50

In this lab, you create views and functions using a new Lab database.

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

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

### Step 1: Create a Lab Database

Run the **copy and paste** 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 MyLabs Database.

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

-- Title: Mod04-Lab03

-- Desc: This file demonstrates how to select data from 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

Create Table Employees -- New Table

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

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

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

,[ManagerID] [int] NULL

);

go

Create Table Inventories

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

,[InventoryDate] [Date] NOT NULL

,[EmployeeID] [int] NOT NULL -- New Column

,[ProductID] [int] NOT NULL

,[Count] [int] NOT NULL

);

go

-- Add Constraints (Module 02) --

Begin -- Categories

Alter Table Categories

Add Constraint pkCategories

Primary Key (CategoryId);

Alter Table Categories

Add Constraint ukCategories

Unique (CategoryName);

End

go

Begin -- Products

Alter Table Products

Add Constraint pkProducts

Primary Key (ProductId);

Alter Table Products

Add Constraint ukProducts

Unique (ProductName);

Alter Table Products

Add Constraint fkProductsToCategories

Foreign Key (CategoryId) References Categories(CategoryId);

Alter Table Products

Add Constraint ckProductUnitPriceZeroOrHigher

Check (UnitPrice >= 0);

End

go

Begin -- Employees

Alter Table Employees

Add Constraint pkEmployees

Primary Key (EmployeeId);

Alter Table Employees

Add Constraint fkEmployeesToEmployeesManager

Foreign Key (ManagerId) References Employees(EmployeeId);

End

go

Begin -- Inventories

Alter Table Inventories

Add Constraint pkInventories

Primary Key (InventoryId);

Alter Table Inventories

Add Constraint dfInventoryDate

Default GetDate() For InventoryDate;

Alter Table Inventories

Add Constraint fkInventoriesToProducts

Foreign Key (ProductId) References Products(ProductId);

Alter Table Inventories

Add Constraint ckInventoryCountZeroOrHigher

Check ([Count] >= 0);

Alter Table Inventories

Add Constraint fkInventoriesToEmployees

Foreign Key (EmployeeId) References Employees(EmployeeId);

End

go

-- Adding Data (Module 03) --

Insert Into Categories

(CategoryName)

Select CategoryName

From Northwind.dbo.Categories

Order By CategoryID;

go

Insert Into Products

(ProductName, CategoryID, UnitPrice)

Select ProductName,CategoryID, UnitPrice

From Northwind.dbo.Products

Order By ProductID;

go

Insert Into Employees

(EmployeeFirstName, EmployeeLastName, ManagerID)

Select E.FirstName, E.LastName, IsNull(E.ReportsTo, E.EmployeeID)

From Northwind.dbo.Employees as E

Order By E.EmployeeID;

go

-- This Code is used to create a Random inventory value

Insert Into Inventories

(InventoryDate, EmployeeID, ProductID, [Count])

Select '20170101' as InventoryDate, 5 as EmployeeID, ProductID, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Union

Select '20170201' as InventoryDate, 7 as EmployeeID, ProductID, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Union

Select '20170301' as InventoryDate, 9 as EmployeeID, ProductID, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Order By 1, 2

go

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

Select \* From Categories;

go

Select \* From Products;

go

Select \* From Employees;

go

Select \* From Inventories;

go

### Step 2: Create Some Queries

Answer the following questions by writing and executing SQL code. We start with some review questions to get you warmed up, and then move on to new ones!

**Notes**:

* You can use any name you like for your views but be descriptive and consistent!
* Quantities may vary since I use a random function to create the data!
* Make sure your code is well-formatted!
* You must use the BASIC views for each table (created in question 1)

**Question 1**: How can you **create BASIC views** to show data from each table in the database.

1. Do not use a \*, list out each column!
2. Create one View per table!
3. Use SchemaBinding to protect the views from being orphaned!

**Question 2**: How can you **set permissions**, so that the public group CANNOT select data from each table, but can select data from each View?

**Question 3**: How can you **create a View** to **show** a list of **Category** and **Product** names and the price of each product? **Order** the result by the Category and Product!

**Question 4**: How can you create a view to **show** a list of **Product** names and Inventory **Counts** by each Inventory Date? **Order** the results by the Product, Date, and Count!

**Question 5**: How can you create a view to **show** a list of Inventory **Dates** and the **Employee** that took the count? **Order** the results by the Date and return only one row per date!

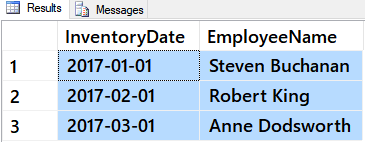


Figure 2: The results desired for question 5

**Question 6**: How can you create a function to **show** a list of Inventory **Dates** and the **Employee** that took the count? **Order** the results by the Date and return only one row per date! Add a parameter to **filter** by the employee's first and last name.

Test your Views and Function with code similar to this:

Select \* From [dbo].[vCategories]

Select \* From [dbo].[vProducts]

Select \* From [dbo].[vInventories]

Select \* From [dbo].[vEmployees]

ff

### Step 3: Review Your Work

Now, you will review your work with your instructor.

# Session03 Lectures and Labs < 110 mins

## Stored Procedures - 30

Stored Procedures are a **named collection of SQL programming code**. While similar to views and functions, stored procedures have much **more flexibility** as to what code can be used and therefore are more flexibleas to what they can do.

### Creating and Executing

Here is **how to create** a simple Stored Procedure (Sproc or Proc):

Create --Drop

Procedure pAddValues

As

Begin

Select [Sum] = 1 + 2;

End

go

However, unlike Views or Function, you do not use them in a Select statement. Instead, **you Execute them like this**:

Execute pAddValues;

go

-- or this ...

Exec pAddValues;

go

-- or even this..

pAddValues;

go

Stored procedures can **Print or Select** data, but views and function can only use Select:

Alter **Proc** pAddValues -- TIP: you can use either Proc or Procedure

As

Begin

Select [Sum] = 2 + 3;

Print 2 + 4;

End

go

Exec pAddValues;

go

### Stored Procedure Parameters

Like Functions, Stored Procedures **can** **use Input Parameters:**

Alter Procedure pAddValues

(@Value1 float, @Value2 float)

As

Begin

Select [Sum] = @Value1 + @Value2;

End

go

You **pass arguments** to the parameter like this:

Exec pAddValues @Value1 = 4, @Value2 = 5;

go

-- Or this ...

Exec pAddValues 4,5;

go

-- **BUT, NOT** like this...

Exec pAddValues(4,5); -- Note that Functions can use Parentheses, but Sprocs cannot!

go

A Store Procedure's parameters can have **default Parameter values**:

Alter Procedure pAddValues

(@Value1 float = 0, @Value2 float = 0)

As

Begin

Select [Sum] = @Value1 + @Value2;

End

go

Now you can execute it with a new argument value or just use the **default argument** **values**:

Exec pAddValues @Value1 = 5, @Value2 = 3;

Or without

Exec pAddValues;

-- in a number of ways

Exec pAddValues @Value1 = 5;

Exec pAddValues @Value2 = 3;

Exec pAddValues @Value1 = 5, @Value2 = Default;

### Reporting Stored Procedures

**Like Views and Functions**, developers use stored procedures to **save complex reporting SQL statements**.

-- View

Create **View** vProductsByCategory

AS

Select CategoryName, ProductName, CurrentPrice = UnitPrice

From Products as P Join Categories as C On P.CategoryId = C.CategoryId;

go

**Select \* from vProductsByCategory;**

Go

-- Function

Create **Function** fProductsByCategory()

Returns Table

AS

Return(

Select CategoryName, ProductName, CurrentPrice = UnitPrice

From Products as P Join Categories as C On P.CategoryId = C.CategoryId;

);

go

**Select \* from fProductsByCategory();**

go

-- Stored Procedure

Create Proc pSelProductsByCategory

AS

Select CategoryName, ProductName, CurrentPrice = UnitPrice

From Products as P Join Categories as C On P.CategoryId = C.CategoryId;

go

**Execute pSelProductsByCategory;**

Go

And, like functions, you can use **Parameters** to create **filtered reporting results**. Stored Procedures with parameters are often used to access report data (**Reporting Stored Procedures**). We will look at these in depth, in the next module, but for now, here is an example:

Create Procedure pSelQuanitiesByProductsPerCategory (@CategoryName nvarchar(50))

AS

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

From Northwind.dbo.Categories as C

Join Northwind.dbo.[Products] as P

On C.CategoryID = P.CategoryID

Join Northwind.dbo.[Order Details] as OD

On OD.ProductID = P.ProductID

Where C.CategoryName = @CategoryName

Group By C.CategoryName, P.ProductName

Order By 1,2;

go

Exec pSelQuanitiesByProductsPerCategory @CategoryName = 'Seafood';

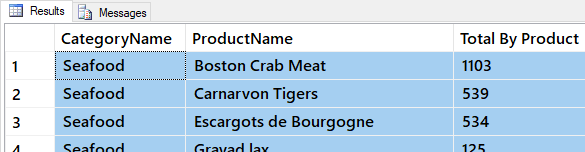


Figure: The result of the previous SQL statements

However, there are several advantages when using Stored Procedures instead of Views or User Defined Functions (UDF). For instance, stored **procedures** **can return results from multiple statements**.

Create Proc pSelProductsAndCategories

AS

Select \* From Northwind.dbo.Products;

Select \* From Northwind.dbo.Categories;

go

Execute pSelProductsAndCategories;

Go

In fact, store **procedures can hold hundreds of statements** at a time, though all statements must be part of a **single SQL batch!**

Create Proc pSelProductsAndCategories

AS

Select \* From Northwind.dbo.Products;

**Go -- This won't work!**

Select \* From Northwind.dbo.Categories;

go

Execute pSelProductsAndCategories;

Go

### Getting Info About Stored Procedures

As with all objects in a database, Stored Procedures **METADATA info** can be found in the SysObjects View. Here is an example:

Select \* From **SysObjects** Where Name = 'pAddValues';

The SysComments View **shows the Code** for the Stored Procedure cross-referenced by its object Id.

Select \* From **SysComments** Where id = Object\_id('pAddValues');

Microsoft provides a system stored procedure that **display's an overview** of a store procedure.

Exec sp\_Help 'pAddValues';

go

Microsoft also provides a system stored procedure that **display's the code** of a store procedure **in its** **original format**:

Exec sp\_HelpText 'AddValues';

go

## Lab 4: Creating Stored Procedures - 20

In this lab, you create stored procedures using the Northwind database.

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 can be done individually or with a group of up to 3 people.

### Step 1: Review Database Tables

Run the following code in a SQL query editor and review the names of the tables.

Select \* From Northwind.Sys.Tables Where type = 'u' Order By Name;

### Step 2: Reuse the Lab Database

You have already created a database for this lab called MyLabsDB\_YourNameHere (using your own name, of course!) Use the following code to force your SQL code to use this database:

**Use [MyLabsDB\_YourNameHere];**

### Step 3: Create a Query

Answer the following questions by writing and executing SQL code.

**Question 1**: How can you create a stored procedure to show a list of **customer** names and their **locations**? Call the procedure **pSelCustomersByLocation**.

**Question 2**: How can you create a stored procedure to show a list of **customer** names, their **locations**, and the **number of orders** they have placed (hint: use the **count()** function)? Call the procedure **pSelNumberOfCustomerOrdersByLocation**.

**Question 3**: How can you create a stored procedure to show a list of **customer** names, their **locations**, and the **number of orders** they have placed (hint: use the **count()** function) on a given **year** (hint: use the **year()** function)? Call the procedure **pSelNumberOfCustomerOrdersByLocationAndYears**.

### Step 4: Review Your Work

Now, you review your work with the instructor.

**NOTE: Unlike assignments, labs do not need to be turned in to Canvas!**

## Variables - 10

As you continue to learn how to program **in SQL, you will find variables to be useful**. The following are some examples of how they work.

### Creating Variables

You **create a variable by declaring its name and data type** like this:

DECLARE @var1 int = 4; -- Setting an initial value only works in some versions of SQL

You can also use either **SELECT** or the **SET** statement to **add a value** of the variable.

DECLARE @var1 int;

SELECT @var1 = 4;

PRINT @var1;

-- or --

DECLARE @var1 int;

SET @var1 = 4;

PRINT @var1;

Variables invisibly hold data in memory, but you can **display a variable's value** using **PRINT** or **SELECT**.

DECLARE @var1 int;

SET @var1 = 4;

PRINT @var1;

SELECT @var1;

If you use **Select,** you display the values as a **row of results**, but if you use **Print,** it only displays as a list of **text**. Note that **Print** **cannot display** **multiple values at once** and **cannot use aliases**.

DECLARE @var1 int, @var2 int;

SELECT @var1 = 4 , @var2 = 8; -- you can set multi values with SELECT

SELECT @var1 as "col1" , @var2 as "col2"; -- you can display multi values with SELECT

Print @var1**,** @var2 **as "col2"**; -- print **cannot** use multiple values or aliases

You cannot set multiple values with the **Set** command,

DECLARE @var1 int, @var2 int;

SET @var1 = 4**,** @var2 = 8; -- you **cannot** set multiple values with SET

### Using Variables in Statements

You can use a variable **in a Select** statement's WHERE clause:

DECLARE @EmpIDVar int = 3;

SELECT \* FROM Northwind.dbo.Employees WHERE EmployeeID = @**EmpIDVar**;

You can use variables **in Insert, Updates, or Delete** statements too.

Create table Products (ProductID int, ProductName varchar(100))

Declare @ProdID int = 1, @ProdName varchar(100) = 'ProdA';

Insert Into Products (ProductID, ProductName) Values (@ProdID, @ProdName);

Update Products Set ProductName = 'ProdA-1' Where ProductID = @ProdID;

Delete From Products Where ProductID = @ProdID;

A variable **can also** **be** **filled by using a select statement** with a From clause, but if more than one value is returned only the last value is retained.

Declare @LastOrder date;

Select @LastOrder = Max(OrderDate)

From Northwind.dbo.Orders

Where CustomerID = 'ALFKI';

Print @LastOrder;

Variables are **removed from memory after a TSQL batch completes**. So, if you need to run the above statement with the one batch of code. Placing a "go" before the Print statement will cause it to fail.

Declare @LastOrder date;

Select @LastOrder = Max(OrderDate)

From Northwind.dbo.Orders

Where CustomerID = 'ALFKI';

Print @LastOrder;

**go**

Print @LastOrder;

...

Msg 137, Level 15, State 2, Line 7

Must declare the scalar variable "@LastOrder".

## Loops and Conditionals - 10

The SQL language focuses on working with data in tables, but sometimes you want to **work** **data not stored as columns and rows**.

### The While Loop

If you want to **perform an action or evaluate an expression repeatedly**, without a Select/From statement, you can use a While loop. Here is an example of a While loop printing a message multiple times.

DECLARE @i int = 65

WHILE (@i <= 67) -- If True

BEGIN

PRINT NCHAR(@i)

SET @i = @i + 1

END

-- Results --

A

B

C

### The IF Else Statement

The If-Else block allows you to **control the flow of a set of one or more statements**.

IF (null = null)

BEGIN

PRINT 'true'

END

ELSE

BEGIN

PRINT 'false'

END

Here is a **more practical example** that determines whether a customer has any orders before deleting the customer from the customer list:

IF EXISTS (SELECT OrderID FROM Northwind.dbo.Orders WHERE CustomerID = 'Frank')

PRINT '\*\*\* Customer cannot be deleted \*\*\*'

ELSE

BEGIN

DELETE Northwind.dbo.Customers WHERE CustomerID = 'Frank'

PRINT '\*\*\* Customer deleted \*\*\*'

END

You can **combine both IF and While** to create more complex programming logic like this:

DECLARE @i int = 65

WHILE (5 = 5) -- create an ALWAYS true condition

BEGIN

PRINT NCHAR(@i)

SET @i = @i + 1

IF @i > 67

BREAK -- The loop stops or "breaks out" of the loop

ELSE

CONTINUE -- The loop continues from the top

END;

### The IIF() Function

The immediate **IIf()** function evaluates a **Boolean expression** in its **1st argument** and returns either one of two arguments based on the results. If the expression evaluates to **True** the **2nd argument** returned, but the **3rd argument** is returned if the Boolean expression is **False** or unknown. Here are some examples:

Print IIF(5=5, 'T', 'F');

Select IIF(5=5, 'T', 'F');

Select ProductName, Discontinued, IIF(Discontinued = 0, 'T', 'F')

From Northwind.dbo.Products

**Note**: Unlike a standard If-Else statement, you cannot conditionally perform an action. **Both IIF and Select-Case** (shown next) **return the result of an expression** based on a Boolean outcome.

### The Select Case Clause

The Select - Case statement is similar to an If-Else. Here are two examples, **one for each of the two versions of this statement**.

Declare @i int = 5;

SELECT CASE 20/@i -- Ask ONE QUESTION then compare the answers

WHEN 5 THEN '@i was 5'

WHEN 4 THEN '@i was 4'

ELSE '@i was something else'

END;

SELECT CASE -- Ask DIFFERENT QUESTIONS looking for the first true answer

WHEN 20/@i = 4 THEN '@i was five'

WHEN 10 + @i = 15 THEN '@i was five as well'

ELSE '@i was something else'

END

Here is **a practical example** of how you would use a SELECT-CASE statement to review the inventory status of products. It queries the Products table and returns messages based on the quantities available, quantities back ordered, and whether the product has been discontinued.

SELECT ProductID, 'Product Inventory Status' =

CASE WHEN (UnitsInStock < UnitsOnOrder AND Discontinued = 0)

THEN 'Negative Inventory - Order Now!'

WHEN ((UnitsInStock-UnitsOnOrder) < ReorderLevel AND Discontinued = 0)

THEN 'Reorder level reached- Place Order'

WHEN (Discontinued = 1)

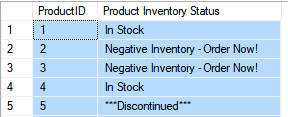
THEN '\*\*\*Discontinued\*\*\*'

ELSE 'In Stock'

END

FROM Northwind.dbo.Products

Order By ProductID;



##### **Figure 6:** The current stock level of products

Reports like this one are known as **Key Performance Indicator** reports. They use a simplified, yet descriptive, set of values to provide a **condensed digest of information** that is easier to understand than a large table of numbers.

Select

ProductID

, Sum(Quantity) as 'Total Qty'

, Case When Sum(Quantity) >= 1000 Then 1

When Sum(Quantity) Between 500 And 1000 Then 0

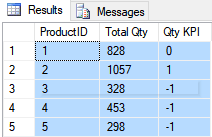
When Sum(Quantity) < 500 Then -1

End as 'Qty KPI'

From Northwind.dbo.[Order Details]

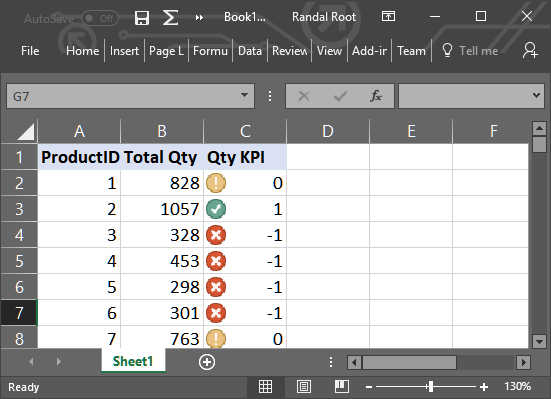
Group By ProductID

Order By ProductID



##### **Figure 7:** The current quantities of products ordered

Often, they will be reduced even further from numbers into images in reporting applications. Here is an example:



##### **Figure 8:** The current quantities of products ordered with KPI images

## Lab 5: Creating KPIs - 10

In this lab, you create a simple KPI report query using the Northwind database.

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

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

### Step 1: Review the Database Table

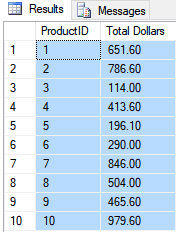
Run the following code in a SQL query editor and review this table's data.

**Use [MyLabsDB\_YourNameHere];**

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

### Step 2: Create an Aggregated Result

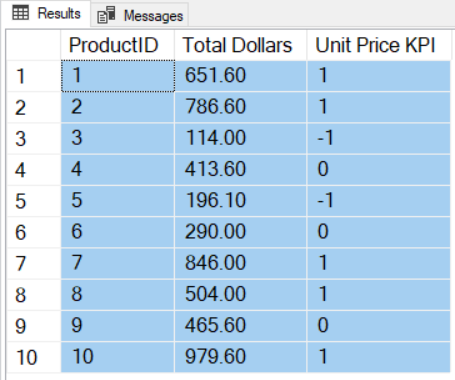
Add code to a script to create a select statement that shows only the top 10 ProductIDs and the total sum of sales dollars for each.



##### **Figure 9:** The year and the total sum of sales for the top 10 product Ids query results

### Step 3: Create a KPI Result

Add code to a script to create a select statement that shows only the top 10 ProductIDs and the total sum of sales dollars for each, and a KPI based on a medium sum of sales dollars being between $250 and $500.



##### **Figure 10:** The year and the total sum of sales for the top 10 product Ids with KPIs query results

### Step 4: Review Your Work

Now, you will review your work with your instructor.

## Dynamic SQL Code - 20

"Dynamic SQL is **a programming technique that enables you to build SQL statements dynamically** at runtime. You can create more general-purpose, flexible applications by using dynamic SQL" (<https://docs.oracle.com/cd/A97630_01/appdev.920/a96590/adg09dyn.htm>, 2017)(external site)

Here is code that **executes a simple select statement** with dynamically assigns the choice of columns:

Declare @ColumnNames varchar(1000) = 'OrderID, ProductId, Quantity'

Exec (

'Select ' + @ColumnNames + ' From Northwind.dbo.[Order Details] Order By ProductID ;'

);

If you want to **see what the code will evaluate into** before you execute it you can replace exec **with select** like this:

Declare @ColumnNames varchar(1000) = 'OrderID, ProductId, Quantity'

Select (

'Select ' + @ColumnNames + ' From Northwind.dbo.[Order Details] Order By ProductID ;'

);

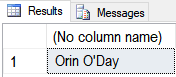


##### **Figure 11:** The result of a dynamically generated select statement

One thing to be aware of is how to **escape single quotes**, which are used to indicate a string of characters. Here is an example:

Declare @FirstName varchar(100) = 'Orin', @LastName varchar(100) = 'O''Day';

Select @FirstName + ' ' + @LastName;



##### **Figure 12:** The result of a dynamically generated select with an escaped single quote

Sometimes developers use dynamic code to perform transaction processing like the examples below:

Create Table Contacts

(FirstName varchar(100)

,LastName varchar(100)

,EmailAddress varchar(100));

go

Declare

@FirstName varchar(100) = 'Bob',

@LastName varchar(100) = 'Smith',

@EmailAddress varchar(100) = 'BSmith@MyCo.com'

Declare @Code varchar(1000) = 'Insert Into dbo.Contacts

(FirstName, LastName, EmailAddress)

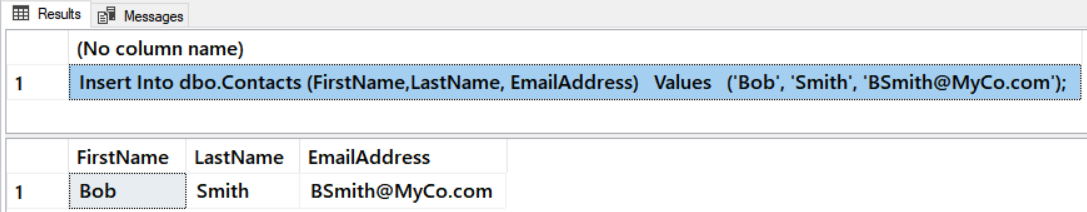
Values

(''' + @FirstName + ''', ''' + @LastName + ''', ''' + @EmailAddress + ''');'

Select @Code

Exec(@Code)

Select \* From Contacts;



**Note**: keep in mind that **stored procedures are a better and safer option**!

### SQL Injection Attacks

Be careful! **These types of statements represent a high-security risk**.

"Many studies have been made around security breaches and SQL injection attacks come among the most popular year after year. **According to the 2011 data breach investigations report produced by Verizon Business, SQL injection attacks were responsible of nearly 25% of all compromised records**. It is obvious that this is a major security issue and it must not be taken lightly, but its consequences are frequently overlooked since those attacked are misunderstood." (<http://www.sqlinjection.net/risks/>, 2017)

-- Example of a SQL Injection Attack

Declare

@FirstName varchar(100) = 'Bob',

@LastName varchar(100) = 'Smith',

@EmailAddress varchar(100) = 'BSmith@MyCo.com'');Select \* From Pubs.dbo.Sales--'

Declare @Code varchar(1000) = 'Insert Into dbo.Contacts (FirstName,LastName, EmailAddress)

Values

(''' + @FirstName + ''', ''' + @LastName + ''', ''' + @EmailAddress + ''');'

Select @Code



**Important:** Never let an application create a dynamic SQL code like this!

### Using Stored Procedures for Transactions

Instead of using a dynamically generated SQL, we should use a stored procedure like this one:

Create Proc pInsContacts

(@FirstName nvarchar(100),@LastName nvarchar(100),@EmailAddress nvarchar(100))

As

Begin

Begin Tran;

Insert Into Contacts (FirstName, LastName, EmailAddress)

Values (@FirstName, @LastName, @EmailAddress);

Commit Tran;

End

go

-- Test that the Sproc works:

Exec pInsContacts

@FirstName = 'Bob'

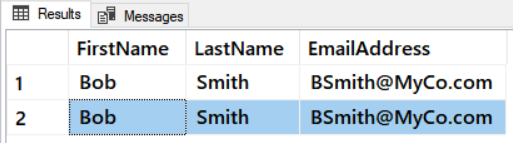
, @LastName = 'Smith'

, @EmailAddress = 'BSmith@MyCo.com';

go

Select \* From Contacts;

Go



Now, if I try to use a **SQL Injection Attack it does not work**!

Exec pInsContacts

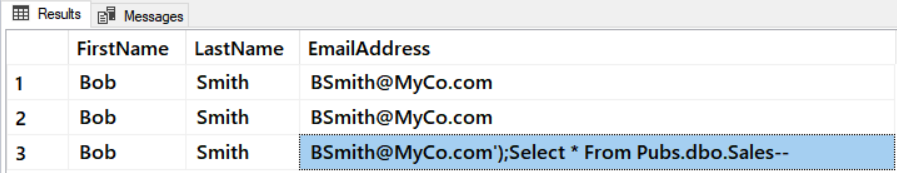
@FirstName = 'Bob',

@LastName= 'Smith',

@EmailAddress = 'BSmith@MyCo.com'');Select \* From Pubs.dbo.Sales--'

go

Select \* From Contacts;



**Important:** Always use Store Procedures whenever you let applications perform transactions!