

Data Admin Concepts & Database Management

Lab 08 – Database Programming

Table of Contents

Data Admin Concepts & Database Management	1
Lab 08 – Database Programming	1
Overview	2
Learning Objectives.....	2
Lab Goals	2
What You Will Need to Begin.....	2
Part 1 – Introducing Functions, Views, and Stored Procedures	3
Setup	3
Functions.....	3
Views	10
Stored Procedures.....	11
Part 2 – Putting All Together.....	15
Coding Your Own Functions.....	15
Coding Your Own Views.....	17
Coding Your Own Stored Procedures.....	19
What to Submit.....	21
Appendix A – VidCast Logical Model Diagram	22
Appendix B – What is Scope?.....	23

Overview

This lab is the eighth of ten labs in which we will build a database using the systematic approach covered in the asynchronous material. Each successive lab will build upon the one before and can be a useful guide for building your own database projects.

In this lab, we will combine structured query language (SQL) DDL and DML to construct some helpful programming objects for our VidCast database.

Read this lab document once through before beginning.

Learning Objectives

In this lab you will

- Demonstrate proficiency in coding and using SQL Server database objects such as Functions, Views, and Stored Procedures

Lab Goals

This lab consists of two sections. The first section is a walkthrough of creating programming objects. In the second part of the lab, you will code your own view, functions, and stored procedures to solve the problems presented.



TIP: If you are new to SQL or programming in general, you may benefit from run through of the SQL Tutorial at <https://www.w3schools.com/sql/>. While not required reading, it can be a helpful resource for new programmers to get some coding in.

What You Will Need to Begin

- This document
- An active Internet connection (if using iSchool Remote lab)
- A blank Word (or similar) document into which you can place your answers. Please include your name, the current date, and the lab number on this document. Please also number your responses, indicating which part and question of the lab to which the answer pertains. Word docx format is preferred. If using another word processing application, please convert the document to pdf before submitting your work to ensure your instructor can open the file.
- To have completed Lab 07 – Advanced Querying
- Understanding of database tables and have reviewed the asynchronous material through Week 8

- One of the following means of accessing a SQL Server installation
 - A connection to the iSchool Remote Lab (<https://rds.syr.edu/>)
 - A local installation of SQL Server (see Developer edition here <https://www.microsoft.com/en-us/sql-server/sql-server-downloads-free-trial>)
 - Regardless of how you access SQL Server, you will need to use SQL Server Management Studio to do so.

Part 1 – Introducing Functions, Views, and Stored Procedures

Setup

While we can leave the basic SQL coding up to our application developers, we would like to create some programming objects to help them and us out. This is useful in properly securing our database as well as making it user-friendly.

Formatting Note



Look for the “To Do” icon to point out sections of the lab you will need to do to complete the tasks.

Functions

A function is a block of predefined code that (often) accepts inputs as a list of parameters, runs some process, and (often) returns a value. We have already used some functions throughout this course.

GetDate() for example, asks for no input parameters, but has some internal mechanic that figures out what the date and time are currently, and responds with that value. We don't need to know what those internal mechanics are; we just need to know that they work and can just let the function do its thing.

The ISNULL function, defined as `ISNULL(expression, result_if_null)`, accepts an expression, perhaps a column name, the result of some mathematical operation, or even another function call, and, if that expression IS NULL, it will substitute the value sent by way of *result_if_null*.

Take it for a spin:



Copy and paste the following code into a blank query editor window. You do not need to be connected to a specific database for this. Be sure to execute all these lines together!

```
-- Declare a variable (we'll talk about variables in a minute)
declare @isThisNull varchar(30) -- Starts out as NULL
SELECT @isThisNull, ISNULL(@isThisNull, 'Yep, it is null') -- See?

-- Set the variable to something other than NULL
SET @isThisNull = 'Nope. It is not NULL'
SELECT @isThisNull, ISNULL(@isThisNull, 'Yep, it is null') -- How about now?
```

Your results should look like the following. Copy a screenshot of your results to your answers doc.

Results		Messages
	(No column name)	(No column name)
1	NULL	Yep, it is null
	(No column name)	(No column name)
1	Nope. It is not NULL	Nope. It is not NULL

We used the aggregate functions, AVG(), SUM(), MIN(), MAX(), and COUNT(), to perform some math for us that would have been more difficult to do if we had to perform that work on our own. The parameters to these functions were expressions that the DBMS will use to calculate the aggregate.

These are all examples of built-in functions; functions made available to us by SQL Server for use in our **SELECT** statements. We can code our own functions to do many different things for us. Some of the most common uses for user-defined functions are to abstract the calculation of derived attributes and to provide lookups for values based on provided parameters.



TIP: SQL Server provides for the creation and use of two types of user-defined functions: *Scalar* and *Table-valued*. This lab only deals with the former. Table-valued functions differ from scalar-valued in that table-valued return tables as results, whereas scalar-valued return single values as results.

Our First User-Defined Function



Code and execute the following code against your IST659 database. The comments are optional, but they're here to outline some basic ideas about how functions are built.

```
11 CREATE FUNCTION dbo.AddTwoInts(@firstNumber int, @secondNumber int)
12 RETURNS int AS
13 BEGIN
14     -- First, declare the variable to temporarily hold the result
15     DECLARE @returnValue int -- the data type matches the "RETURNS" clause
16
17     -- Do whatever needs to be done to set that variable to the
18     -- correct value
19     SET @returnValue = @firstNumber + @secondNumber
20
21     -- Return the value to the calling statement
22     RETURN @returnValue
23 END
24 GO
```



After running the code above, code and execute the following SQL SELECT statement against your database.

```
SELECT dbo.AddTwoInts(5, 10)
```

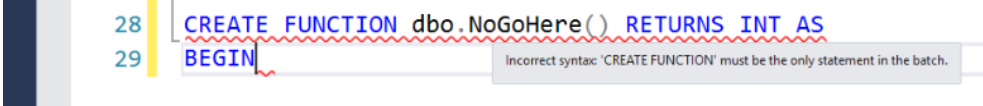
Your results should look like this:

Results		Messages	
(No column name)			
1	15		

Let's break that function into its lines to see what each does. First, the code again.

```
11 CREATE FUNCTION dbo.AddTwoInts(@firstNumber int, @secondNumber int)
12 RETURNS int AS
13 BEGIN
14     -- First, declare the variable to temporarily hold the result
15     DECLARE @returnValue int -- the data type matches the "RETURNS" clause
16
17     -- Do whatever needs to be done to set that variable to the
18     -- correct value
19     SET @returnValue = @firstNumber + @secondNumber
20
21     -- Return the value to the calling statement
22     RETURN @returnValue
23 END
24 GO
```

Line #	Purpose
11	<p>CREATE FUNCTION is the DDL starting point for creating functions.</p> <p><code>dbo.AddTwoInts</code> is what we're naming this function. Technically, we're creating a function called <code>AddTwoInts</code> in the <code>dbo</code> schema. We don't need the schema for other objects, but we do for functions.</p> <p><code>(@firstNumber int, @secondNumber int)</code> After we provide the function name, we provide a list of parameters in parentheses, each separated by a comma. To declare a parameter, you give it a name, beginning with the at sign (<code>@</code>) and a data type.</p>
12	<p>RETURNS int tells SQL Server that this function will be processing information to derive a result of the <code>int</code> data type. This can be whatever data type is appropriate for the task at hand. In our case, we are adding two <code>ints</code> together to create another <code>int</code>, so we set the return data type to <code>int</code>.</p> <p>AS is the keyword that ends the CREATE FUNCTION clause and tells SQL Server that the line(s) to follow represent the code to be executed when this function is called.</p>
13	<p>BEGIN tells SQL Server we are about to start a block of code, all of which belong together. If we didn't have the BEGIN and the subsequent END (see line 23), SQL Server would not know to group this code together and would only take the first line following the AS as the whole function.</p>
15	<p>DECLARE @returnValue int tells SQL Server to create an <code>int</code>-sized area in memory that we're calling <code>@returnValue</code> to temporarily hold a value. In this case, we're creating a variable that will hold the result of our math. The <code>@</code> is mandatory at the beginning of the variable name in SQL Server.</p> <p>You can create many variables for many reasons. In most cases, you'll want at least one in a function to hold the value you'd like to return to the calling code. You can name them just about anything you'd like, but it is best for your own sake and for the sake of others who may have to read your code to give them a name that has some meaning.</p> <p>We have named our variable <code>@returnValue</code> because it is the value to be returned to the calling code. We could just as easily have named it <code>@phil</code>, but, while I'm sure Phil is an alright person, it doesn't make for a very informative variable name.</p>
19	<p>This does the operation that the function is designed to do and assigns the result to the <code>@returnValue</code> variable.</p>

22	After setting @returnValue to the appropriate result, we need to RETURN it to the calling code.
23	This END corresponds to the BEGIN on line 13 and signals to SQL Server that we are done with this block of code.
24	<p>The GO keyword signals to SQL Server to end the current batch of commands to process and begin anew. This is not required in most cases, but if you don't isolate CREATE FUNCTION, CREATE VIEW, or CREATE PROCEDURE statements by ensuring they are between GO statements, SSMS will interpret the statement as an error stating the CREATE statement must be the only statement in the batch.</p> <p>If you see an error like the one in the following screenshot, simply add a GO to the line before the CREATE statement.</p> 

Functions That Are More... well... Functional

Among the many helpful tasks that functions can perform are abstracting the calculation of organizational metrics and performing lookups to other data based on inputs.

Abstracting Routine Calculation

Organizations rely on data to make both tactical and strategic decisions. These metrics are often calculated based on events that have happened that impact performance. Because these metrics are based on an organization's rules, they are subject to change with some frequency. Catalysts for those changes can include new management, changes within the organization that require new calculations, or changes in the organization's industry that require us to rethink how we calculate these metrics.

Instead of hand-coding the calculation in every instance it is needed (printed reports, management dashboards, displays mounted throughout the organization's work space, intranets and extranets, etc), we can code the calculation once in a function. Whenever we need to show the metric, we can call the function, passing the appropriate parameters and the output will be the result of the current math.

Our VidCast service is interested in the number of VidCasts created by VidCast users. Let's code a function that counts the number of VidCasts made by a given user and returns the count to the calling code.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```

29 -- Function to count the VidCasts made by a given User
30 CREATE FUNCTION dbo.vc_VidCastCount(@userID int)
31 RETURNS int AS -- COUNT() is an integer value, so return it as an int
32 BEGIN
33     DECLARE @returnValue int -- matches the function's return type
34
35     /*
36      * Get the count of the VidCasts for the provided userID and
37      * assign that value to @returnValue. Note that we use the
38      * @userID parameter in the WHERE clause to limit our count
39      * to that user's VidCast records.
40      */
41     SELECT @returnValue = COUNT(vc_UserID) FROM vc_VidCast
42     WHERE vc_VidCast.vc_UserID = @userID
43
44     -- Return @returnValue to the calling code.
45     RETURN @returnValue
46 END
47 GO

```

After you execute that statement, code and execute the following SQL code against your database:

```

49 SELECT TOP 10
50     *
51     , dbo.vc_VidCastCount(vc_UserID) as VidCastCount
52 FROM vc_User
53 ORDER BY VidCastCount DESC

```

Your results should look like this:

	vc_UserID	UserName	EmailAddress	UserDescription	WebSiteURL	UserRegisteredDate	VidCastCount
1	20	ecstatic	blandit.enim.consequat@loremutaliquam.co.uk	Bandwidth series A financing niche market.	NULL	2017-11-16 00:00:00.000	22
2	40	principle	ac.uma@miac.com	Business-to-business ecosystem ramen social media...	http://principle.videocast659.site	2017-11-01 12:14:24.000	19
3	24	metacarpal	et.magna.Praesent@placeraugaugueSed.org	Research & development startup long tail strategy g...	http://metacarpal.videocast659.site	2017-05-30 11:31:12.000	18
4	43	canadian	Curabitur.dictum.Phasellus@elefendnec.com	Agile development ownership business-to-consumer...	http://canadian.videocast659.site	2017-06-27 05:16:48.000	18
5	26	przewalski	amet@Maurisolestie.org	NULL	http://przewalski.videocast659.site	2017-02-11 08:52:48.000	17
6	36	silly	accumsan@gravidasagittisDuis.net	Stock founders early adopters low hanging fruit A/B...	http://silly.videocast659.site	2017-05-25 14:38:24.000	17
7	7	wood	turpis.egestas.Fusce@massanonante.net	Technology investor marketing alpha.	http://wood.videocast659.site	2017-06-21 15:36:00.000	16
8	11	doughnut	ipsum.primis@Cumsociis.com	Assets sales incubator user experience ecosystem ...	http://doughnut.videocast659.site	2017-01-31 01:12:00.000	16
9	14	groggy	omare.In.faucibus@egestas.ca	Sales niche market user experience investor social ...	http://groggy.videocast659.site	2017-04-20 09:50:24.000	16
10	47	these	parturient.montes@ipsum.ca	Entrepreneur vitality freemium crowdsourcing long tail ...	http://these.videocast659.site	2017-12-07 03:50:24.000	16

In your own words, in your answers document, describe what lines 49 through 53 above actually do. Also, how is it that this code knows that the vc_User record with vc_UserID = 20 has 22 vc_VidCast records?

Performing Data Lookups

Often, we have one piece of data and we would like to look up another piece of data in a table. This is a common task when using surrogate keys. A table where a surrogate primary key is used may have another column that serves as the natural key. A value that identifies the real-world object in real-world terms that may not be a suitable primary key candidate.

For example, our `vc_Tag` table has a surrogate primary key, `vc_TagID`, but our users do not care and may not even know about this column. Instead, they will use the values in the column, `TagText`, to specify or search for tags. This makes `TagText` a natural key. We can code a function that accepts the tag text as a parameter and looks up the `vc_TagID` for the `vc_Tag` record for that `TagText`.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```

55 GO
56 --Function to retrieve the vc_TagID for a given tag's text
57 CREATE FUNCTION dbo.vc_TagIDLookup(@tagText varchar(20))
58 RETURNS int AS -- vc_TagID is an int, so we'll match that
59 BEGIN
60     DECLARE @returnValue int -- Matches the function's return type
61
62     /*
63         Get the vc_TagID of the vc_Tag record whose TagText
64         matches the parameter and assign that value to @returnValue.
65     */
66     SELECT @returnValue = vc_TagID FROM vc_Tag
67     WHERE TagText = @tagText
68
69     -- Send the vc_TagID back to the caller
70     RETURN @returnValue
71 END
72 GO

```

After you execute that statement, code and execute the following SQL code against your database:

```

75 SELECT dbo.vc_TagIDLookup('Music')
76 SELECT dbo.vc_TagIDLookup('Tunes')

```

Your results should look like this:

Results		Messages
(No column name)		
1	4	
(No column name)		
1	NULL	

In your own words, in your answers document, describe what lines 75 and 76 above actually do. Also, when line 76 executed, we received a NULL from SQL Server. How come?

Views

Views are another programming object we can use to secure and simplify access to the data. As we'll see in a future lab, we can lock down access to certain data points to specific database users using a view. We can also use views to predefine complex SQL **SELECT** statements and allow programmers (including ourselves) to simply access the views instead of writing queries that might include esoteric **JOINS** or aggregate functions.

A view is, simply put, a prepackaged SQL **SELECT** statement. Most of the time, these are read-only structures meant to abstract the intricacies of building complex **SELECT** statements to make our database easier to use by applications.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```
79 -- Create a view to retrieve the top 10 vc_Users and their
80 -- VidCast counts
81 CREATE VIEW vc_MostProlificUsers AS
82     SELECT TOP 10
83         *
84         , dbo.vc_VidCastCount(vc_UserID) as VidCastCount
85     FROM vc_User
86     ORDER BY VidCastCount DESC
87 GO
```

After you execute that statement, code and execute the following SQL code against your database:

```
88 SELECT * FROM vc_MostProlificUsers
```

Your results should look like this:

	vc_UserID	UserName	EmailAddress	UserDescription	WebSiteURL	UserRegisteredDate	VidCastCount
1	20	ecstatic	blandit.enim.consequat@loremutaliquam.co.uk	Bandwidth series A financing niche market.	NULL	2017-11-16 00:00:00.000	22
2	40	principle	ac.uma@miac.com	Business-to-business ecosystem ramen social media...	http://principle.videocast659.site	2017-11-01 12:14:24.000	19
3	24	metacarpal	et.magna.Praesent@placeraaugueSed.org	Research & development startup long tail strategy g...	http://metacarpal.videocast659.site	2017-05-30 11:31:12.000	18
4	43	canadian	Curabitur.dictum.Phasellus@elefendnec.com	Agile development ownership business-to-consumer...	http://canadian.videocast659.site	2017-06-27 05:16:48.000	18
5	26	przewalski	amet@Maurismolestie.org	NULL	http://przewalski.videocast659.site	2017-02-11 08:52:48.000	17
6	36	silly	accumsan@gravidasagittisDuis.net	Stock founders early adopters low hanging fruit A/B...	http://silly.videocast659.site	2017-05-25 14:38:24.000	17
7	7	wood	turpis.egestas.Fusce@massanonanite.net	Technology investor marketing alpha.	http://wood.videocast659.site	2017-06-21 15:36:00.000	16
8	11	doughnut	ipsum.primis@Cumsoicis.com	Assets sales incubator user experience ecosystem ...	http://doughnut.videocast659.site	2017-01-31 01:12:00.000	16
9	14	groggy	omare.In.faucibus@egestas.ca	Sales niche market user experience investor social ...	http://groggy.videocast659.site	2017-04-20 09:50:24.000	16
10	47	these	parturient.montes@ipsum.ca	Entrepreneur virality freemium crowdsourcing long tail ...	http://these.videocast659.site	2017-12-07 03:50:24.000	16

In your own words, in your answers document, describe what lines 79 through 87 above are doing.



TIP: Notice that lines 82 through 85 above are the same as what we previously ran when we tested our user-defined function from before. A good strategy for authoring views is to first get the *SELECT* statement correct and then just put the *CREATE VIEW* line before it, execute the whole thing, and your view is done!

Stored Procedures

Stored procedures are like functions in that they perform operations based on provided parameter values, but they are different in that they can perform more complex database activities. For instance, whereas a user-defined function can make no changes to the database in any way, a stored procedure can be used to encapsulate and abstract statements such as *INSERT*, *UPDATE*, and *DELETE*.

For example, we can prohibit a database user from running a SQL *UPDATE* statement for fear of running an *UPDATE* without a *WHERE* clause and causing untold havoc on our data. We can then author a stored procedure to require the necessary criteria be provided and we can code the *UPDATE* statement within that procedure using that criteria.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```

91 -- Create a procedure to update a vc_User's email address
92 -- The first parameter is the user name for the user to change
93 -- The second is the new email address
94 CREATE PROCEDURE vc_ChangeUserEmail(@userName varchar(20), @newEmail varchar(50))
95 AS
96 BEGIN
97     UPDATE vc_User SET EmailAddress = @newEmail
98     WHERE UserName = @userName
99 END
100 GO

```

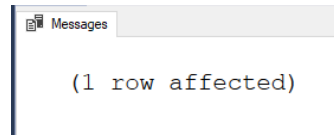
After you execute that statement, code and execute the following SQL code against your database:

```

102 EXEC vc_ChangeUserEmail 'tardy', 'kmstudent@syr.edu'

```

Your results should look like this:



To see the effect, code and execute the following statement against your database:

```
104 | SELECT * FROM vc_User WHERE UserName = 'tardy'
```

Your Results should look like this:

Results		Messages				
	vc_UserID	UserName	EmailAddress	UserDescription	WebSiteURL	UserRegisteredDate
1	6	tardy	kmstudent@syr.edu	Startup leverage growth hacking bootstrapping sc...	http://tardy.vidcast659.site	2017-03-12 15:36:00.000

In your own words, in your answers document, describe what lines 91 through 104 above are doing.

Stored procedures are also helpful in locking down access to **INSERT**ing data in a table. With practice, you should be able to accept many parameters and insert into more than one table, perform lookups using functions, views, or basic select statements.

For now, we will focus on the mechanical elements of using stored procedures to add data to our tables.

The SCOPE_IDENTITY() System Function

When a new row is added to a table and the identity property does its thing and uses the next available number to populate an integer column (per our method for using surrogate keys as primary keys), the server does its best to remember what the given number is. It technically stores this value in a server variable called **@@identity**, a variable that holds the last given identity value at the **process** level.

For very simple databases, it may be safe to use the **@@identity** variable to get the last identity value.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```
106 | INSERT INTO vc_Tag (TagText) VALUES ('Cat Videos')
107 | SELECT * FROM vc_Tag WHERE vc_TagID = @@identity
```

Your result should look like this:

Results		Messages	
	vc_TagID	TagText	TagDescription
1	15	Cat Videos	NULL

Your `vc_TagID` might be different, but the result is still correct if you only retrieved the Cat Videos TagText.

More complex database environments ~~might~~ have triggers, database objects that execute when certain events happen. These triggers can cause new records to be added to one or more other tables. This is a common tactic for databases in replication environments or where data auditing has been coded in. When these events happen, it can change the value in `@@identity` so we don't want to rely on that in our database code.

Instead, we can use a function called `SCOPE_IDENTITY()` which gives us the identity value generated by the current scope.

For more information on what is meant by scope, see [Appendix B](#) below. This isn't something that will affect us in our coursework but is good to know in any programming context!



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```
INSERT INTO vc_Tag (TagText) VALUES ('Dog Videos')
SELECT * FROM vc_Tag WHERE vc_TagID = SCOPE_IDENTITY()
```

Your result should look like this:

Results		Messages	
	vc_TagID	TagText	TagDescription
1	18	Dog Videos	NULL

Again, your `vc_TagID` might be different, but the result is still correct if you only retrieved the Dog Videos TagText.



Code and execute the following SQL against your database. You should be using the same database built using the code from Lab 6.

```

110  /* Create a procedure that adds a row to the UserLogin table
111      This procedure should be run when a user logs in. It will
112      record who they are and from where they're logging in
113  */
114
115  CREATE OR ALTER PROCEDURE vc_AddUserLogin(@userName varchar(20), @loginFrom varchar(50))
116  AS
117  BEGIN
118      -- We have the user name, but we need the user ID for the login table
119      -- First, declare a variable to hold the ID
120      DECLARE @userID int
121
122      SELECT @userID = vc_UserID FROM vc_User
123      WHERE UserName = @userName
124
125      -- Now we can add the row using an INSERT statement
126      INSERT INTO vc_UserLogin (vc_UserID, LoginLocation)
127      VALUES (@userID, @loginFrom)
128
129      -- Lastly, return the SCOPE_IDENTITY() so the
130      -- calling code knows the primary key of the row we
131      -- just added
132      RETURN SCOPE_IDENTITY()
133  END
134  GO

```

After successfully running the previous code, code and execute the following code against your database (note: code all of it and execute all of it at the same time or you will get an error).

```

136  DECLARE @addedValue int
137  EXEC @addedValue = vc_AddUserLogin 'tardy', 'localhost'
138  SELECT
139      vc_User.vc_UserID
140      , vc_User.UserName
141      , vc_UserLogin.UserLoginTimestamp
142      , vc_UserLogin.LoginLocation
143  FROM vc_User
144  JOIN vc_UserLogin ON vc_User.vc_UserID = vc_UserLogin.vc_UserID
145  WHERE vc_UserLoginID = @addedValue

```

Your results should look like this:

Results		Messages		
	vc_UserID	UserName	UserLoginTimestamp	LoginLocation
1	6	tardy	2020-09-29 12:07:49.750	localhost

Your UserLoginTimestamp value will be different than the one shown. On your answers doc, explain why this is.

On your answers doc, also identify one way we could simplify the code in the stored procedure above. (Hint: Look back at how we did a lookup with the vc_Tag table)

Part 2 – Putting All Together

In this part, you'll create functions, views, and stored procedures to round out the external model of the VidCast database.

Coding Your Own Functions

In this section, you'll code new user-defined functions to perform specific tasks. In most cases, you've already coded something similar in part 1, so refer to your previous work to see how to solve the problems below.



The code below is the beginning of a function intended to retrieve a vc_UserID from the vc_User table given a specified @userName. Complete the code to assign the correct vc_UserID to @returnValue

```
147  /*
148      Create a function to retrieve a vc_UserID for a given user name
149  */
150  CREATE FUNCTION dbo.vc_UserIDLookup(@userName varchar(20))
151  RETURNS int AS
152  BEGIN
153      DECLARE @returnValue int
154
155      -- TODO: Write the code to assign the correct vc_UserID
156      --         to @returnValue
157
158      RETURN @returnValue
159  END
160  GO
```

After creating the vc_UserIDLookup function, run the following SELECT statement against your database:

```
SELECT 'Trying the vc_UserIDLookup function.', dbo.vc_UserIDLookup('tardy')
```

Paste a screenshot of your results in your answers document. You'll paste your SQL later.



Author a function called `dbo.vc_TagVidCastCount` that calculates the count of `vc_VidCastIDs` for a given `vc_TagID`. Consult the diagram at the end of this document as a reference for the tables involved.

After you've authored the function and successfully created it, execute the following code against your database:

```
SELECT
    vc_Tag.TagText
    , dbo.vc_TagVidCastCount(vc_Tag.vc_TagID) as VidCasts
FROM vc_Tag
```

Your results should look like this:

	TagText	VidCasts
1	Art	256
2	Audio Recording	266
3	Baseball	242
4	Basketball	236
5	Cat Videos	0
6	Collectibles	258
7	Consoles	260
8	Fashion	240
9	Football	259
10	Games	254
11	Motors	0
12	Music	237
13	Personal	263
14	Professional	264
15	Sports - General	235

Paste a screenshot of your results in your answers document. You will paste your SQL later.



Code a function called `vc_VidCastDuration` that SUMs the total number of minutes of actual duration for VidCasts with a Finished status given a `vc_UserID` as a parameter. This function should return the SUM as an int.

The easiest way to calculate the VidCast duration as a number of minutes for each individual `vc_VidCast` record is using the built-in `DateDiff` function:

```
DATEDIFF(n, StartDateTime, EndDateTime)
```

You will need to incorporate the `DATEDIFF` function into your function to get the correct value. (Hint: Use the `dbo.vc_VidCastCount` function you created earlier as a starting point.)

Once you've created the function, execute the following **SELECT** statement against your database:

```
SELECT
    *
    , dbo.vc_VidCastDuration(vc_UserID) as TotalMinutes
FROM vc_User
```

Your results should look like this (not every row is shown):

	vc_UserID	UserName	EmailAddress	UserDescription	WebSiteURL	UserRegisteredDate	TotalMinutes
1	1	ethanol	Donec.tempus@penatibusetmagnis.co.uk	Agile development non-disclosure agreement equity ...	http://ethanol.vidcast659.site	2017-12-30 22:19:12.000	1859
2	2	dispatcher	quam@aptenttacilisociosqu.ca	A/B testing handshake disruptive seed money infogr...	http://dispatcher.vidcast659.site	2017-12-08 03:36:00.000	1368
3	3	camel	mauris@massanon.edu	User experience founders branding entrepreneur iter...	http://camel.vidcast659.site	2017-08-14 03:21:36.000	1859
4	4	infatuated	mollis@Nam.org	Lean startup launch party angel investor branding b...	http://infatuated.vidcast659.site	2017-06-07 17:02:24.000	1426
5	5	hygienist	magna.Ut@necumasuscipit.ca	Business model canvas accelerator pivot network ef...	http://hygienist.vidcast659.site	2017-03-17 23:16:48.000	1139
6	6	tardy	kmstudent@eyr.edu	Startup leverage growth hacking bootstrapping scru...	http://tardy.vidcast659.site	2017-03-12 15:36:00.000	2303
7	7	wood	turpis.egestas.Fusce@massanonante.net	Technology investor marketing alpha.	http://wood.vidcast659.site	2017-06-21 15:36:00.000	2292
8	8	mallard	vel.lectus.Cum@velteget.edu	Assets sales success bandwidth business model ca...	http://mallard.vidcast659.site	2017-09-15 19:55:12.000	1828
9	9	lifted	eu@elitised.net	NULL	http://lifted.vidcast659.site	2017-04-15 20:24:00.000	1828
10	10	gum	ut@pharetraQuisqueac.com	Infographic incubator hypotheses client conversion ...	http://gum.vidcast659.site	2017-02-24 09:07:12.000	1238
11	11	doughnut	ipsum.primis@Cumsocis.com	Assets sales incubator user experience ecosystem a...	http://doughnut.vidcast659.site	2017-01-31 01:12:00.000	1830
12	12	bewildered	Donec.pottitor.tellus@odioAliquamvulp...	Infrastructure research & development venture burn ...	http://bewildered.vidcast659.s	2017-12-29 09:07:12.000	1944
13	13	albite	nisi@Vitaemauris.org	Learning curve partnership buzz value proposition re...	http://albite.vidcast659.site	2017-07-25 00:00:00.000	1971
14	14	groggy	omare.In.faucibus@egestas.ca	Sales niche market user experience investor social ...	http://groggy.vidcast659.site	2017-04-20 09:50:24.000	2464
15	15	bicycle	Quisque.pottitor.eros@mi.net	Success network effects focus monetization iPhone...	http://bicycle.vidcast659.site	2017-01-17 12:00:00.000	1152
16	16	hills	vitae.nosure.at@vestitulummassa.co.uk	Angel investor technology ramen learning curve non	NULL	2017-10-07 12:43:12.000	1240

Paste a screenshot of your results in your answers document. You don't need to ensure every row is in the screenshot, just the first ten to fifteen rows will be fine. You will paste your SQL later.

Coding Your Own Views

In this section, you'll code new views and alter existing views to perform specific tasks. In most cases, you've already coded something similar in part 1, so refer to your previous work to see how to solve the problems below.



Create a view called **vc_TagReport** that executes the **SELECT** statement:

```
SELECT
    vc_Tag.TagText
    , dbo.vc_TagVidCastCount(vc_Tag.vc_TagID) as VidCasts
FROM vc_Tag
```

Code a **SELECT** statement that returns all rows from this view in descending order of **VidCasts**.

Your results should look like this:

	Tag Text	VidCasts
1	Audio Recording	266
2	Professional	264
3	Personal	263
4	Consoles	260
5	Football	259
6	Collectibles	258
7	Art	256
8	Games	254
9	Baseball	242
10	Fashion	240
11	Music	237
12	Basketball	236
13	Sports - General	235
14	Motors	0
15	Cat Videos	0

Paste a screenshot of your results in your answers document. You will paste your SQL later.



Alter the view called `vc_MostProlificUsers`, adding a column called `TotalMinutes` that calls the `vc_VidCastDuration` function we created earlier in part 2.

Hint: to alter a view, copy and the original code you wrote for part 1 to the end of your SQL file, change the word `CREATE` to `ALTER`, code in the new column, and execute the entire `ALTER VIEW` statement (be sure to execute everything from the `ALTER VIEW` line through the end of the `SELECT` statement!

After you have coded and executed the `ALTER VIEW`, execute the following SQL against your database:

```
SELECT UserName, VidCastCount, TotalMinutes FROM vc_MostProlificUsers
```

Your results should look like this:

	UserName	VidCastCount	TotalMinutes
1	ecstatic	22	2682
2	principle	19	3413
3	canadian	18	1928
4	metacarpal	18	3053
5	przewalski	17	2664
6	silly	17	2851
7	archives	16	2374
8	doughnut	16	1830
9	groggy	16	2464
10	sines	16	2316

Paste a screenshot of your results in your answers document. You will paste your SQL later.

Coding Your Own Stored Procedures

In this section, you'll code new stored procedures to perform specific tasks. In most cases, you've already coded something similar in part 1, so refer to your previous work to see how to solve the problems below.



The following is the beginning of a stored procedure to use in adding a row to the vc_Tag table. All but the INSERT statement has been provided for you. Finish this procedure by coding the INSERT statement. Then execute the entire CREATE PROCEDURE block.

```

225 /*
226     Create a stored procedure to add a new Tag to the database
227     Inputs:
228         @tagText : the text of the new tag
229         @description : a brief description of the tag (nullable)
230     Returns:
231         @@identity with the value inserted
232 */
233 CREATE PROCEDURE vc_AddTag(@tagText varchar(20), @description varchar(100)=NULL) AS
234 BEGIN
235     -- Code the procedure here!
236
237
238     RETURN @@identity
239 END
240 GO

```

After you have coded and executed the completed procedure, execute the following SQL against your database:

```

DECLARE @newTagID int
EXEC @newTagID = vc_AddTag 'SQL', 'Finally, a SQL Tag'
SELECT * FROM vc_Tag where vc_TagID = @newTagID

```

Your results should look like this:

Results		Messages	
	vc_TagID	TagText	TagDescription
1	16	SQL	Finally, a SQL Tag

Paste a screenshot of your results in your answers document. You will paste your SQL later.



Code a stored procedure called `vc_FinishVidCast` that accepts an int as a parameter that will be a `vc_VidCastID` that we will need to mark as finished. The act of finishing a VidCast means we must change its `EndDateTime` to be the current Date and Time (think `GetDate()`) and change the `vc_StatusID` to the `vc_StatusID` for the 'Finished' status.

All the work can be done in a single `UPDATE` statement inside the stored procedure. Be sure to code the `WHERE` clause!

After you have coded and executed the completed procedure, execute the following SQL against your database:

```
DECLARE @newVC int
INSERT INTO vc_VidCast
    (VidCastTitle, StartDateTime, ScheduleDurationMinutes, vc_UserID,
    vc_StatusID)
VALUES (
    'Finally done with sprocs'
    , DATEADD(n, -45, GETDATE())
    , 45
    , (SELECT vc_UserID FROM vc_User WHERE UserName = 'tardy')
    , (SELECT vc_StatusID FROM vc_Status WHERE StatusText='Started')
)

SET @newVC = SCOPE_IDENTITY()
SELECT * FROM vc_VidCast WHERE vc_VidCastID = @newVC
EXEC vc_FinishVidCast @newVC
SELECT * FROM vc_VidCast WHERE vc_VidCastID = @newVC
```



TIP: Execute all the code from `DECLARE @newVC int` until the last `SELECT` statement all at once.

Your results should look like this (your dates, times and IDs may differ):

Results		Messages							
	vc_VidCastID	VidCastTitle	StartDateTime	EndDateTime	ScheduleDurationMinutes	RecordingURL	vc_UserID	vc_StatusID	
1	859	Finally done with sprocs	2018-05-27 22:50:28.297	NULL	45	NULL	6	2	

	vc_VidCastID	VidCastTitle	StartDateTime	EndDateTime	ScheduleDurationMinutes	RecordingURL	vc_UserID	vc_StatusID	
1	859	Finally done with sprocs	2018-05-27 22:50:28.297	2018-05-27 23:35:28.427	45	NULL	6	3	

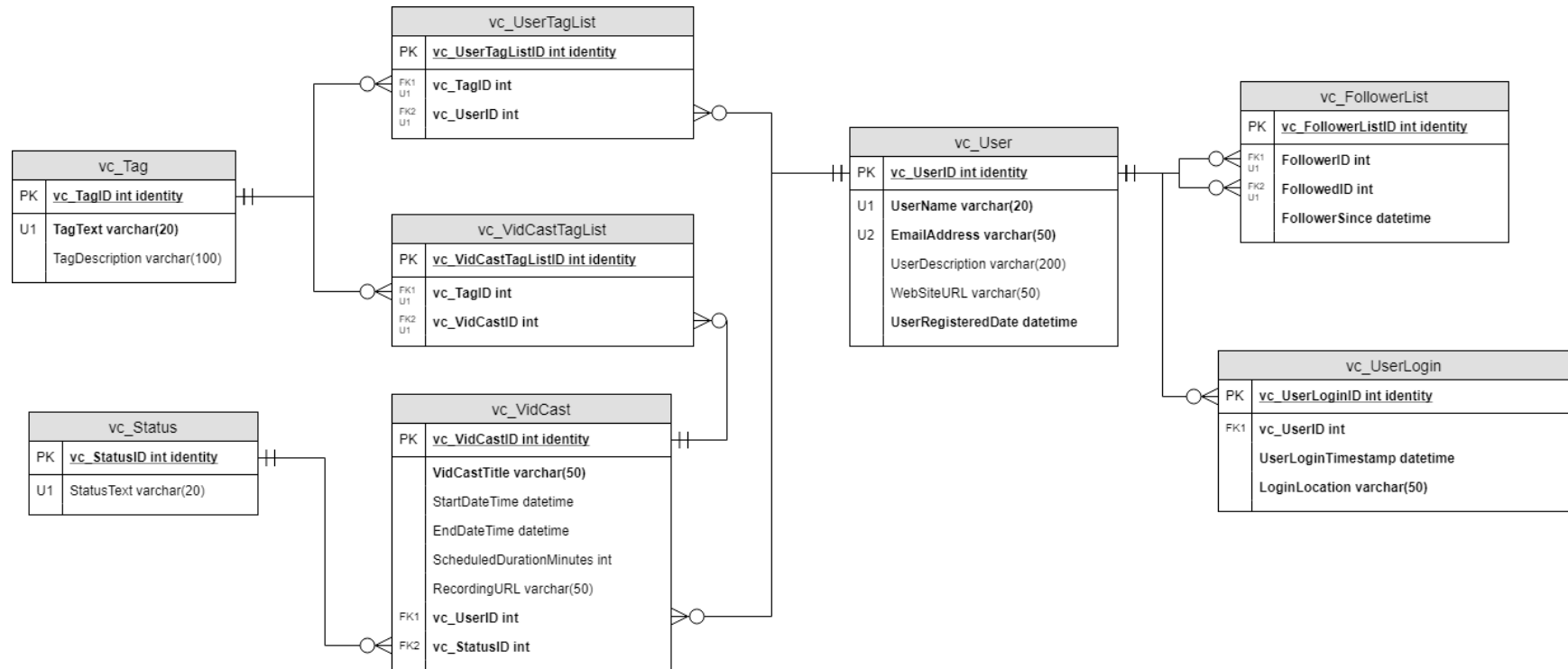
Paste a screenshot of your results in your answers document. You will paste your SQL later.

What to Submit



After completing Part 2, copy and paste the text of your SQL query file at the end of your answers document. Save this document and submit it to the appropriate section on the LMS.

Appendix A – VidCast Logical Model Diagram



For the full diagram, see <https://drive.google.com/file/d/1KRqkSvQABuTMXqYAZojTct9etTSR8Vea/view?usp=sharing>

Appendix B – What is Scope?

Throughout this topic, we have seen examples of code that asks other code to run. This is a core property of well-designed software systems for reasons we don't go into here. Suffice to say, encapsulating code to perform routine tasks saves us lots of precious time and ensures deterministic execution of frequently executed code.

There are also many types of scope, but let's distill it down to its core conceit.

Whenever we execute code, it runs in its own scope. Whenever we ask other code to run by calling a function or stored procedure, we are creating a new scope. This is called **functional scope**. While that code is executing, that scope is active and distinct from the original scope. Unless care is taken to define them otherwise, any variables declared, used, or modified in that scope exist in and only in that scope. Once the code within that scope stops executing, the scope and all volatile storage (RAM) allocated for that scope are destroyed (for the most part – some languages like C and C++ require programmers to clean up their own memory, SQL does not).

That last fact is why we have the RETURN clause on functions and some stored procedures. This is how we allow code from one scope to make use of values derived within another scope. RETURN passes back the value and the calling code in the other scope can do with it what it pleases. To be complete, you can also specify parameters to stored procedures as output parameters, which does a similar thing, but we don't cover that here.

You may have noticed that the functions and stored procedures we've created have BEGIN and END lines at the top and bottom of the procedural code within the definition. In t-SQL, BEGIN and END are demarcations of a **code block**. This also defines another scope imaginatively named **block scope**. A block of code is code designed to execute in a sequence, usually until some condition is met. Those familiar with other programming languages have seen this construct before. C and C adjacent languages use curly braces { } to begin and end their blocks. Python and some others use indentation to define block levels (hence why Python is so persnickety with indentation – there's a structural reason for it!).

Blocks of code are created to execute as a result of conditions (IF A is true, execute block Y, else execute block B) or in loops (execute block Z until some condition is met). Volatile storage allocated within those blocks is destroyed after any single execution of that block and is inaccessible to code outside that block. However, these kinds of scopes are considered a sub-scope of the code that started the block code running. This means that code within the block has access to the variables and such of the parent scope(s).

So how does this affect us? In our examples above, we used one bit of code to call other bits of code. That set off the chain of events that added to or retrieved data from the database. Looking at the INSERT procedures specifically, we saw how adding a row to a table with an integer with the identity property changed the @@identity variable. This variable exists at the process scope.

Each connection to the database is its own process (the number in the tab of your query editor tells you what the process id is if you ever need to know). All code run by that process has access to that variable.

The `SCOPE_IDENTITY()` function is a way to get the value generated by the current function or block scope. So other code can modify `@@identity` all it wants, but we can still see our own number.

If you're planning to write more code, it can be important to think about scope as it can have a real effect on performance, reliability, and readability.