

Data Admin Concepts & Database Management

Lab 09 – Data Security

Table of Contents

Data Admin Concepts & Database Management	1
Lab 09 – Data Security	1
Overview	1
Learning Objectives.....	2
Lab Goals.....	2
What You Will Need to Begin.....	2
Part 1 – Securing Data Objects.....	3
Setup	3
Some Definitions.....	3
Server Setup.....	4
Creating a Database User.....	8
Managing a User’s Permissions	10
Part 2 – Data Integrity Through Transactions.....	13
Transactions, in a Nutshell.....	13
What to Submit.....	17
Appendix A – VidCast Logical Model Diagram	18

Overview

This lab is the ninth 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 explore the tools available to us in Microsoft SQL Server to secure and maintain integrity of our data.

Read this lab document once through before beginning.

Learning Objectives

In this lab you will

- Demonstrate proficiency in creating database users and administering to their user privileges on database objects
- Demonstrate proficiency in preserving data integrity using transactions

Lab Goals

This lab consists of two sections. The first section explores using database security tools and Data Security Language (DSL) to secure the data in our VidCast database. The second part will use transactions to ensure data integrity.



TIP: *Some parts of this lab are relevant only if using your own server, some parts are relevant only on the remote lab, but most of the steps are for both. Be sure to read the headings of each section to know if that section applies to you.*

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 08 – Database Programming
- Understanding of database tables and have reviewed the asynchronous material through Week 9
- One of the following means of accessing a SQL Server installation
 - A connection to the iSchool Remote Lab (<https://remotelab.ischool.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 – Securing Data Objects

Setup

Until now, we have only used one login to access and modify our database. Regardless of whether you have used the Remote Lab or your own installation, you have been accessing your database as a user with the *db_owner* role. That is, you are the administrator of your database.

It is not a recommended practice to allow external applications to connect as administrators with rights to create and delete objects in the database. To do so would open your database up to all manner of external threats.

Although the obvious threat comes from systems and people attempting to do harm, these threats are not always malicious. An application with free reign on the data objects and data contained in your database may write code that is inadvertently harmful. In last week's content, we wrote code that protected our data from such things, but if we don't control access to those objects, a surly or impatient application programmer could code around them and cause havoc in our database.

The complete list of database roles for SQL Server 2017 can be found here:

<https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles?view=sql-server-2017>

Although we could use one or more of those roles as a starting point for our security plan, we need a more fine-grained control over what our application services can access, so we will apply permissions directly at the object level. For instance, *db_datareader* role allows the database user to read data from any table, this is less than ideal since some tables shouldn't be directly read from.

Formatting Note



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

Some Definitions

In our context, a **database user** is not necessarily a person seated at a keyboard or standing at a kiosk. Instead, when we're talking about a database user, we mean the user credentials an application uses to *connect to a database*.

As we've mentioned before, people don't work directly with a database. Instead, they use an application to work with the data and objects in a database. When you connect to SQL Server using SSMS, you are

using the SSMS application and it is passing along the credentials you provide to authenticate with the server and gain access to the database.

The last part of the previous sentence illustrates an important point. A **server login** is different from a database user in a very important way. A server login confirms to SQL Server that the user has permission to *connect to the server* at all. If you don't provide valid server login credentials, the server will not even establish a connection. We map a database user to a server login at the database level to say that once a user has authenticated with the server, they can now use our database in the prescribed way.

Each server login can be mapped to database users on zero or many databases. For instance, on the Remote Lab (and indeed by default on any SQL Server installation), your server login is your Windows credentials. This login has then been mapped to a database user in the db_owner role on your own database.

On the Remote Lab, your instructor also has a server login that is their Windows credentials. That server login is then mapped to database users of each of their student databases.

Taken together, that means that both you and your instructor connect to SQL Server in the same way, but you only have access to your database, while your instructor can access their own database as well as yours and those of your colleagues.

If you want to test this out, for *funnies*, try to access one of your colleagues' databases on SQL Server in the remote lab. You should get an error message saying you don't have permission to do that. (Pause while the author ruminates on whether it's a good idea to tell students to try to hack into other students' databases...)

In summary:

- a **server login** is required to *connect to a server*
- a **database user** is required to *access a database* and is mapped from a **server login**

Server Setup



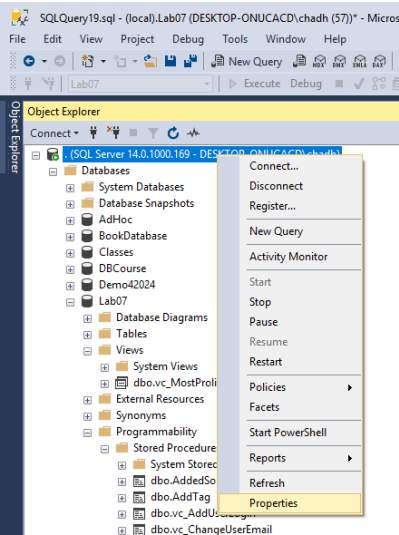
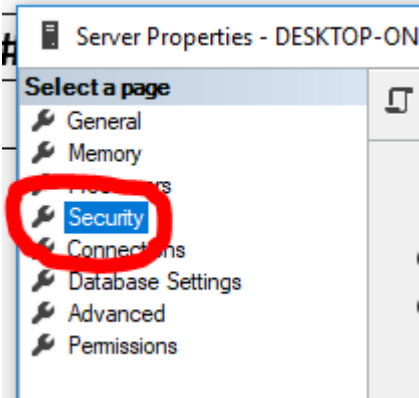
TIP: *The following section pertains only to those who are **running their own SQL Server**. If you are using the remote lab, you do not need to perform these steps. In fact, if you try, you will get error messages!*

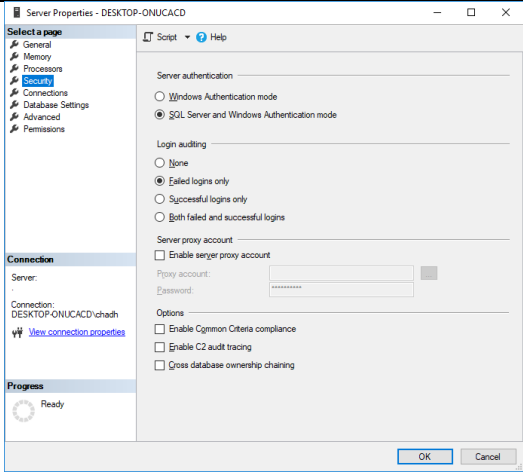
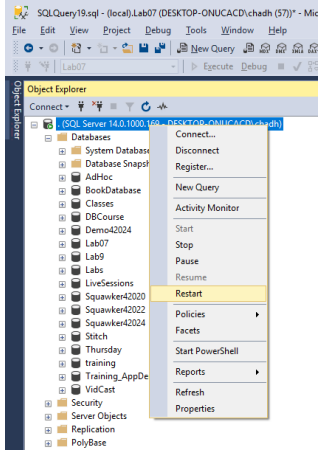
When installing SQL Server, the default server authentication only allows Windows authentication. That is, you can only create database logins for Windows users. This default is best-practice, as Microsoft recommends only allowing Windows logins for SQL Server. It also allows us to utilize Active Directory and Group Policy Objects to apply permissions to individual Windows accounts.

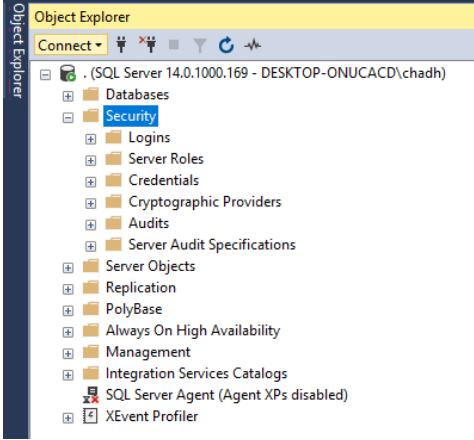
Because we don't all have Active Directories to work with, we need to enable SQL Server Authentication Mode to be able to create logins for our users. (Feel free to disable this after you're done with the course).



Use the following steps to enable SQL Server Authentication for your server:

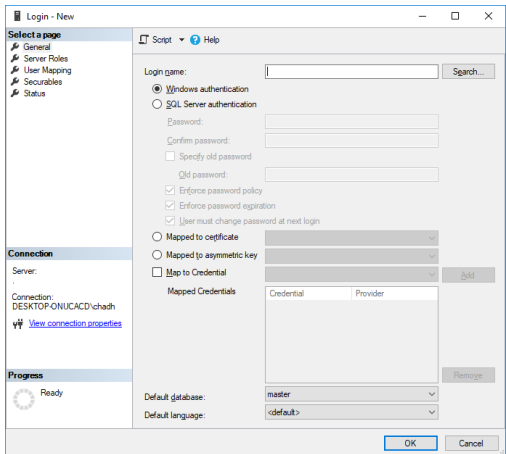
Step #	Action
1	Connect to your server using SSMS normally.
2	<p data-bbox="467 567 1403 640">In the Object Explorer, right click on your server name (the root of the tree) and click Properties.</p> 
3	<p data-bbox="467 1239 1403 1312">In the Server Properties dialog, click the Security page from the Select a Page section.</p> 
4	Under Server Authentication, select the radio button next to SQL Server and Windows Authentication Mode and click OK.

	
<p>5</p>	<p>SQL Server will prompt you to restart SQL Server after making this change. It is not necessary to restart your entire computer. Simply right click the server again in Object Explorer and click Restart.</p>  <p>There may be several prompts that you can agree to. The server service(s) will then restart.</p>
<p>6</p>	<p>In Object Explorer, expand the Security folder at the level immediately below the server. (Not a database-level security folder)</p>



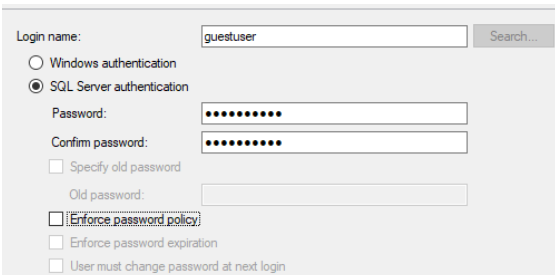
Note that the Databases folder has been minimized in the screenshot above to mitigate confusion.

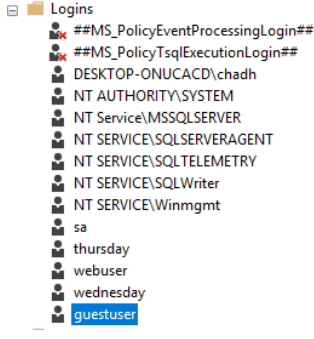
7 Right-click the Logins folder and choose New Login. The new login dialogue appears.



8 Use the following values for your login:

- Login name*: **guestuser**
- **SQL Server Authentication**
- Password*: **SU2orange!** (the ! is part of the password)
- **Uncheck Enforce password policy**
- **Leave everything else as-is**



	* = Feel free to use whatever login name and password you prefer, but the rest of this lab uses these values, so plan accordingly.
9	<p>Click OK when finished. We'll use code to manage the other aspects of a login. Our Login now appears in the list of Logins.</p> 

Creating a Database User



TIP: The following section is for everyone, regardless of which server you're using.

We can map a server login to a database user in one line of code.



Code and execute the following SQL statement against your database.

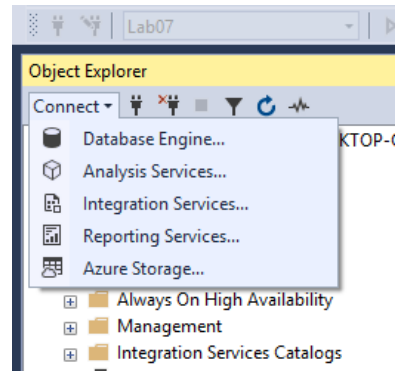
```
1 | -- Creating a guestuser database user
2 | CREATE USER guestuser FOR LOGIN guestuser
```

Copy and paste your code to your answers doc.

Now that we have a login and a user, we can connect as the guestuser account and try some SQL against our database!



In Object Explorer, click Connect and the Database Engine.



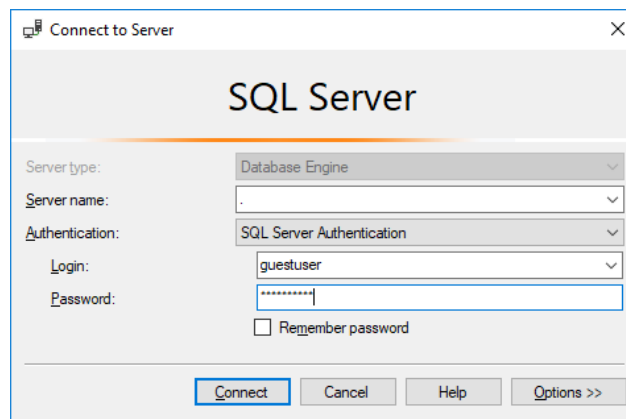
The familiar SQL Server login dialog appears. The server will stay the same, but we will make a few changes. As a reminder, the server for the remote lab is

`ist-s-students.syr.edu`

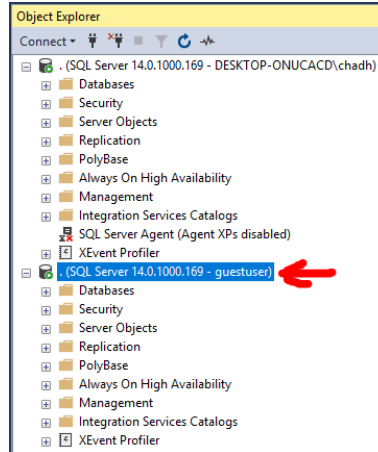
First, change Authentication from Windows Authentication to SQL Server Authentication.

For Login, use `guestuser`

For password, use `SU2orange!` (Note that the ! is part of the password.)



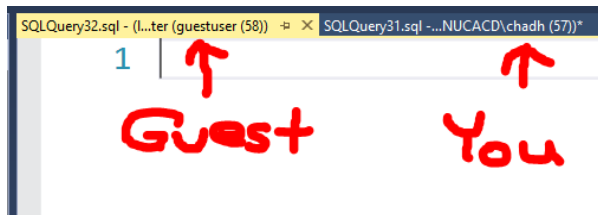
Click Connect. You should now see a second server listed in your Object explorer. The login name listed at the end of the server name is `guestuser`.



This means that anything you do using the objects below this server will be done as the guestuser login.

Right-click this new server and click New Query. A blank Query Editor window appears.

Take special note that in the tab, guestuser is indicated. This means that any code executed in this tab will be done as the guestuser login. Your other tab(s) is/are still connected as you, so we can flip back and forth between these and emulate two different users coding against the database.



TIP: This lab will refer the guestuser tab you just created as “guestuser’s tab” and the other tabs you have opened as “your tab”.

Managing a User’s Permissions



TIP: The following section is for everyone, regardless of which server you’re using.



In `guestuser`'s tab, select your database from the available databases list in the upper left-hand corner of the window. By default, the `guestuser` login will be using the master database, a system database we do not want to touch.

If you are using your own server, select the database where you have been coding the course labs.

If you are using the Remote Lab, your database name will take the following form:

```
IST659_M4??_netid
```

Where `M4??` is the section number you're enrolled in and `netid` is your netid (the user name you used to login to the remote lab). For instance, if your netid is `jpstudent` and you are enrolled in the `M405` section of `IST659`, your database name would be:

```
IST659_M405_jpstudent
```

If you're not sure what yours is, click your tab and the database listed in the Available Databases box will show your database name.

Another (optional) way to set your current database is with the SQL `USE` command.

Once `guestuser`'s tab has the appropriate database selected, code and execute the following SQL `SELECT` statement.

```
1 | -- Guestuser's tab
2 | SELECT * FROM vc_User
_ |
```

You should get the following error message (with `your_db` replaced with whatever your database is):

```
Msg 229, Level 14, State 5, Line 2
The SELECT permission was denied on the object 'vc_User', database 'your_db',
schema 'dbo'.
```

How rude, right?

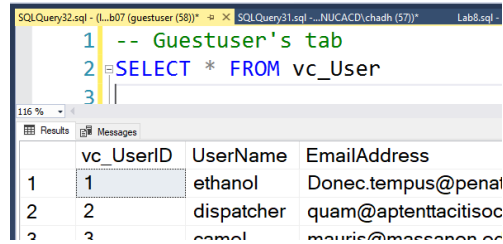
As we saw previously, we can connect to the database, but we don't have permissions to do anything against the database.



In your tab, code and execute the following SQL `GRANT` statement.

```
4 | -- Grant read permission on the user table
5 | GRANT SELECT ON vc_User to guestuser
```

In **guestuser's** tab, execute the code we tried previously. You should get some results instead of a surly error message.



```

1  -- Guestuser's tab
2  SELECT * FROM vc_User
3

```

	vc_UserID	UserName	EmailAddress
1	1	ethanol	Donec.tempus@pena
2	2	dispatcher	quam@aptentacitisoc
3	3	camel	mauric@messenan

This is great, but we created a view for a reason, so let's revoke **guestuser's SELECT** permission on **vc_User**, and instead grant them **SELECT** permission on one of the views we created in lab 8, **vc_MostProlificUsers**.



In your tab, code and execute the following statements to revoke the select permission on **vc_User** and grant the select permission on the **vc_MostProlificUsers** view.

```

7  -- Revoke the select permission!
8  REVOKE SELECT ON vc_User to guestuser
9
10 -- Give them the view instead
11 GRANT SELECT ON vc_MostProlificUsers to guestuser

```

In **guestuser's** tab, code and execute the following SQL statement.

```

4  SELECT * FROM vc_MostProlificUsers

```

You should see the exact same results you saw when working on Lab 8!

If we wanted to, we could also grant **UPDATE**, **INSERT**, and **DELETE** permissions on tables. The Syntax is the same as with **SELECT**. In fact, you can grant them all at once by separating them with commas as such (the following is illustrative only. Don't run it):

```
GRANT SELECT, UPDATE, INSERT, DELETE ON vc_User to guestuser
```

However, we have carefully crafted some external model programming objects that will take the place of all those things. To allow a user to run a stored procedure, we grant them the **EXECUTE** permission.



In your tab, code and execute the following SQL statements:

```
13 | -- Allow guestuser to run some stored procedures
14 | GRANT EXECUTE ON vc_AddUserLogin TO guestuser
15 |
16 | GRANT EXECUTE ON vc_FinishVidCast TO guestuser
```

In guestuser's tab, code and execute the EXEC statement to add a user login for the user with UserName 'TheDoctor' with a login from 'Gallifrey'. Refer to Lab 8 if you need the syntax for this.

Also, in guestuser's tab, code and execute the EXEC statement to finish the VidCast titled 'Rock Your Way To Success'. You may have to do some SQL in your tab to find what you need for this.

In your tab, code and execute a SELECT statement to retrieve all rows from the vc_UserLogin table. Paste a screenshot of the results in your answers doc.

In your tab, code and execute a SELECT statement to retrieve ONLY the vc_VidCast record that should have been modified by guestuser's stored procedure call. Paste a screenshot of the results in your answers doc.

Copy and paste the code from both your tab and guestuser's tab to your answers doc.



TIP: *We are done with security for today. If you're using the Remote Lab, it is advisable to revoke any permissions you have granted to guestuser, as every student is using that account for this lab and would have access to your database using it!*

Part 2 – Data Integrity Through Transactions

In this part, you'll use transactions to ensure data integrity.

Transactions, in a Nutshell

Very often, our code will manipulate the data in more than one table in rapid sequence. In almost all these cases, we would like the effect of these changes to occur simultaneously. Owing to the way RDBMS' operate, and with a little help from the limitations foisted on us by time and space, these changes actually happen sequentially.

We need a mechanism to ensure that all the changes we make occur in the database, and to unravel anything that has occurred should any one change fail. We can use transactions to do this.

What follows is but a primer on the topic, but the flow is the same, even for larger transactions.

Generally, the flow of a transaction is as follows:

1. Begin the transaction
2. Establish the current state of things
3. Make the change
4. Check the new state of things
5. Is it what was expected?
 - a. Yes: Commit the transaction
 - b. No: Rollback the transaction

In 5b above, when we see that something is wrong, we can revert everything back to the state it was before we did anything. This is crucial. It also speaks to a couple of things we want to think about with our transactions.

The desired properties of transactions are referred to broadly as the ACID properties. ACID is an acronym for atomicity, consistency, isolation, and durability.

- **Atomicity**
 - The steps of a transaction can not be subdivided. All the operations performed by the code within a transaction must successfully execute, or none of them can be allowed to succeed. In fact, if a single line in a 100-line sequence fails, we MUST roll back the changes made by the other 99 lines.
- **Consistency**
 - Once a transaction has completed, successfully or otherwise, all data must be left in a state consistent with the explicit or implicit data rules in our database. If something we've done would violate any rule, such as a **FOREIGN KEY** or **CHECK** constraint, the transaction must be rolled back.
- **Isolation**
 - For the duration of the transaction, from **BEGIN TRANSACTION** until **COMMIT** or **ROLLBACK**, no other process should be privy to the machinations occurring in the transaction.
- **Durability**
 - Once the transaction has completed, whatever changes it made are permanently in place.

The following is a simple coding exercise that shows how transactions can be used.

The Setup

We're going to set up two simple tables separate from our VidCast tables to mess with. From here on out, make sure you're using YOUR tab. Not guestuser's.



Code and execute the following CREATE TABLE statements against your database:

```
1  -- Creating a new table
2  CREATE TABLE lab_Test (
3      lab_TestID int identity primary key,
4      lab_testText varchar(20) unique not null
5  )
6
7  /*
8      This will be a table to keep a log of
9      created lab_Test records.
10     We don't want to add a row to this if
11     the inserty into lab_Test fails
12  */
13 CREATE TABLE lab_Log (
14     lab_LogID int identity primary key,
15     lab_logInt int unique not null
16 )
```

In this, admittedly obtuse, example, we're going to add records to lab_Test and, if they succeed, we're going to insert the ID generated for lab_TestID.



Code and execute the following SQL INSERT INTO statements against your database.

```
19 INSERT INTO lab_Test (lab_testText) VALUES ('One'), ('Two'), ('Three')
20 INSERT INTO lab_Log (lab_logInt) SELECT lab_TestID FROM lab_Test
```

Now let's use a transaction make sure our data conform to our weird rules.

Code and execute the following against your database:

```

25 -- Step 1: Begin the transaction
26 BEGIN TRANSACTION
27 -- Step 2: Assess the state of things.
28 DECLARE @rc int
29 SET @rc = @@ROWCOUNT -- Initially 0
30
31 -- Step 3: Make the change
32 -- On success, @@ROWCOUNT is incremented by 1
33 -- On failure, @@ROWCOUNT does not change
34 INSERT INTO lab_Test (lab_testText) VALUES ('One')
35
36 -- Step 4: Check the new state of things
37 IF(@rc = @@ROWCOUNT) -- If @@ROWCOUNT was not changed, fail
38 BEGIN
39     -- Step 5, if failed
40     SELECT 'Bail out! It Failed!'
41     ROLLBACK
42 END
43 ELSE -- Success! Continue.
44 BEGIN
45     -- Step 5 if succeeded
46     SELECT 'Yay! It worked!'
47     INSERT INTO lab_Log (lab_logInt) VALUES (@@identity)
48     COMMIT
49 END
50
51 SELECT * FROM lab_Log
52 SELECT * FROM lab_Test

```

Your results should look like this:

The screenshot shows the results of the SQL script. The first result is a message: 'Bail out! It Failed!'. The second result is a table with columns 'lab_LogID' and 'lab_logInt', containing three rows: (1, 1), (2, 2), and (3, 3). The third result is a table with columns 'lab_TestID' and 'lab_testText', containing three rows: (1, One), (2, Three), and (3, Two).

(No column name)	
1	Bail out! It Failed!

	lab_LogID	lab_logInt
1	1	1
2	3	2
3	2	3

	lab_TestID	lab_testText
1	1	One
2	3	Three
3	2	Two

Change line 34 to insert your last name. Re-execute lines 25 through 52. Did it work?

On your answers doc, paste a screenshot of your results.

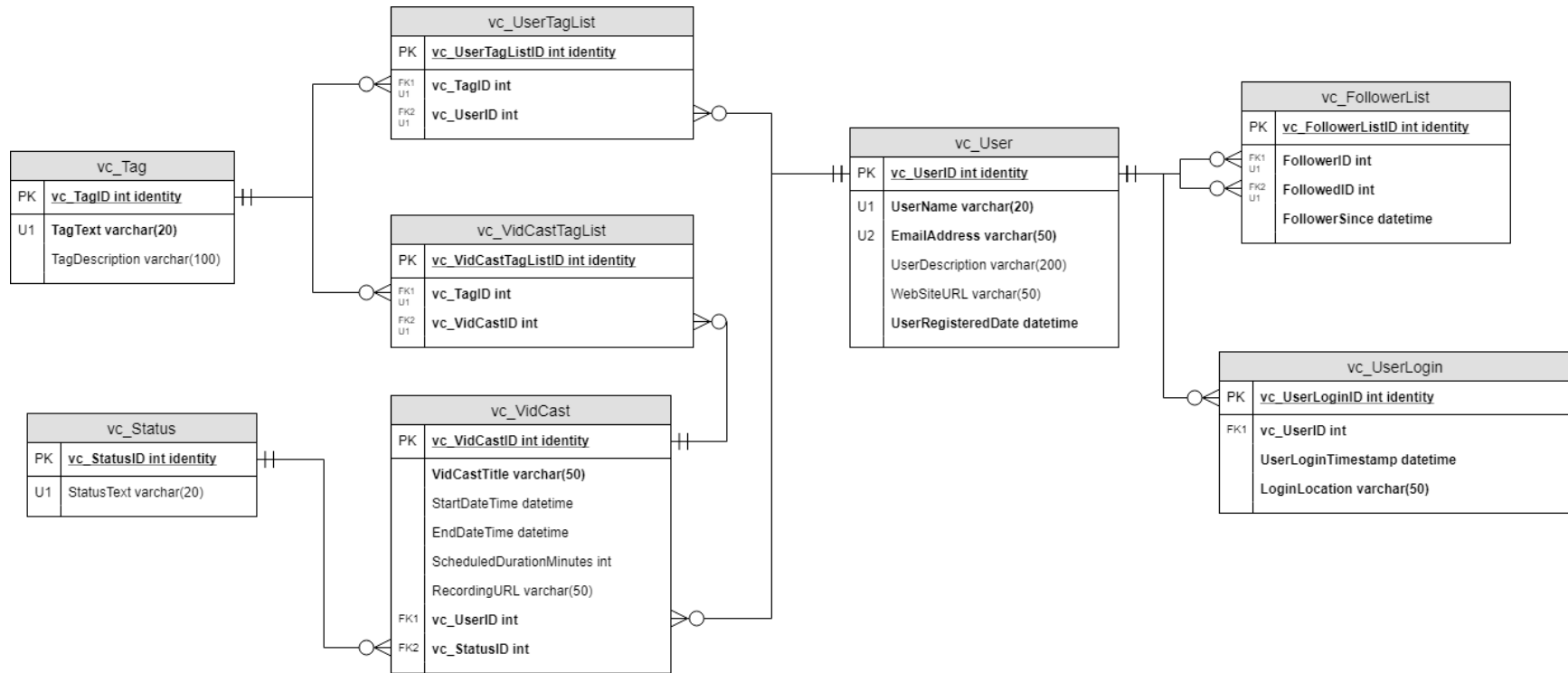
Also, in your own words, explain the reason the first execution failed, but the second did not. Was there anything that happened that you didn't expect?

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>