**Databases Management Systems**

**Bet’s Pets’ Vets**

**Part B – Implementation**

Table of Contents

[**Introduction** 3](#_Toc136229747)

[**Create Tables using SQL DDL** 4](#_Toc136229748)

[**Populate Tables using SQL INSERT** 6](#_Toc136229749)

[**Retrieve Information using SQL Queries** 7](#_Toc136229750)

[**MUST** 7](#_Toc136229751)

[1. Record information for each pet, including its type / species, approximate age (DoB not always known), comments. 7](#_Toc136229752)

[2. Enable 7](#_Toc136229753)

[3. Record staff information, identifying whether they are VETs or VET NURSEs. 8](#_Toc136229754)

[4. Record appointment information, showing pet and staff information as well as date and time. Allow space for outcome of appointment to be recorded. 8](#_Toc136229755)

[**SHOULD** 9](#_Toc136229756)

[5. Show all appointments for today (including home visits) for a given staff member. Include time, pet, owner, staff, and treatment codes and cost. 9](#_Toc136229757)

[6. Record payments made, including the invoice information, amount paid, method of payment etc. A payment can only be applied to one invoice (see Case Study). Generate a unique receipt number for each payment. 9](#_Toc136229758)

[7. Show staff information, showing their line managers. Include staff who do not have line managers. 10](#_Toc136229759)

[8. Calculate the total owed by each owner (whether invoices are overdue or not). 11](#_Toc136229760)

[**COULD** 12](#_Toc136229761)

[9. Identify any unpaid invoices dated more than 28 days ago, showing the name and address of the responsible owner. Identify the owners that owe the most. 12](#_Toc136229762)

[10. Calculate the total amount owed to each surgery (consider how Home visits will be included). 13](#_Toc136229763)

[11. Find the average income generated by each staff member per month (ignore unpaid invoices). 14](#_Toc136229764)

[**Implementation of DBMS Functionality** 16](#_Toc136229765)

[**WOULD BE NICE** 16](#_Toc136229766)

[12. Create a stored procedure to input new pet information, including associating them with an existing or new owner (given the owner’s name). 16](#_Toc136229767)

[13. Create a trigger that will check how much a given owner owes when they make an appointment. If there are any unpaid invoices more than 28 days old, do not make the booking. 17](#_Toc136229768)

[14. Create an app capable of running the queries. 19](#_Toc136229769)

[**Business Justification** 22](#_Toc136229770)

[References 23](#_Toc136229771)

# **Introduction**

In this project I will attempt to construct a practical database for “Bets Pets Vets” that meet the system requirements prescribed, to do this I will be using the entity-relationship diagram (ERD) provided with some minor adjustments and I will be using SQL Server Management Studio (SSMS) as well as LocalDB which comes with the Express version of SSMS.

I’ve chosen to use LocalDB for a few reasons.

1. Ease of setup and lightweight – Easy and quick installation, much smaller than the full version of SSMS, requires less resources than the full version as the database is launched on demand rather than run as a service.
2. Fully capable – Has the exact same abilities and functionality of the full version of SQL server, while not intended for large scale databases it is great for small scale development and testing.
3. Fully compatible – Fully compatible with other SQL Server editions supporting the same T-SQL language and programming model which can provide a smooth transition from development to production.

Finally, I’d like to create a simple user-friendly graphical interface using JavaFX and SceneBuilder and then connect the database to it using Java Database Connectivity (JDBC) drivers which will be able to run the prescribed queries.

A screenshot of a computer

Description automatically generated

# A screenshot of a computer program Description automatically generated with medium confidence**Create Tables using SQL DDL**

*CREATE TABLE tblStaffType* creates the table ‘tblStaffType’ with 2 columns – **staffTypeID** and **staffTypeDesc**.

The **staffTypeID** is the primary key for the table and is an auto-incrementing integer with the ‘IDENTITY(1,1)’ meaning to start from 1 and increment by 1 each time a new staff type is added.

**staffTypeDesc** is a VARCHAR string with a maximum length of 10 and is ‘NOT NULL’ meaning if a new staff type is added it will need a description.

*tblPaymentType*, *tblPetTypes* and *tblOwners* follow the same pattern with differences in maximum length of descriptions and **petTypeID** being a string cannot be auto-incrementing in the same way as an integer ID like **staffTypeID** or **paymentTypeID** so will need to be hard-coded in for new pet types.

*CREATE TABLE tblPets* also has an auto-incrementing integer starting from 1 for its primary key, an VARCHAR with a maximum length of 25 for pet names, a CHAR with a maximum length of 2 to represent pet types, an INT to represent the owner ID and finally a DATE which will represent the pets date of birth and an VARCHAR for pet comments that has a maximum length of a thousand.

Figure 1 CREATE Statement

A screenshot of a computer program

Description automatically generated with medium confidenceBoth the **petTypeID** and **ownerID** are foreign keys referencing their respective tables primary keys.

*CREATE* *tblStaff*, *tblSurgeries* and *tblTreatments* again look very similar to the first set of tables with a couple exceptions, firstly we have a self-referencing foreign key in the *tblStaff* table, this makes **lineManagerID** dependant on the **staffID** itself which creates a hierarchical relationship within the staff table – this could get complicated depending how big your hierarchy is but as we only have 1 line manager it should be fine.

Next, we have a different data type in the *tblTreatments* with the **treatmentCost** which is ‘DECIMAL(5,2)’ this denotes the data will have a decimal point, 5 digits total and 2 digits after the decimal point (This gives a maximum treatment cost of 999.99), we also have a ‘CHECK’ constraint on the cost which makes the cost have to be equal or greater than zero – this potentially allows for a free treatment but cannot be negative.

*CREATE tblAppointments* and *tblPayments* also have a couple of differences, in *tblAppointments* we have **paidStatus** with a ‘CHECK’ constraint which forces the status to either be ‘Y’ or ‘N’, NOT NULL and by default be ‘N’ and in *tblPayments* we have a check constraint on the **paymentAmount** where the amount must be greater or equal to zero, the zero allowing for payment at a later date.

Figure 2 CREATE Statement

And this is a screenshot after the tables were created:

A screenshot of a computer

Description automatically generated

Figure 3 CREATE Execution

# **Populate Tables using SQL INSERT**

This is my INSERT script separated into two screenshots, as I’m going through the system requirements, I’m sure I’ll be inserting more data.

A screenshot of a computer program

Description automatically generated with low confidence

Figure 4 INSERT Statement

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 5 INSERT Statement

A screenshot of a computer

Description automatically generated with medium confidence

And the proof with the number of rows affected as well as a SELECT statement to show tblAppointments and its newly inserted data.

Figure 6 INSERT Execution

A screenshot of a computer

Description automatically generated with medium confidence

Figure 7 INSERT Queried

# **Retrieve Information using SQL Queries**

## **MUST**

### Record information for each pet, including its type / species, approximate age (DoB not always known), comments.

A picture containing text, screenshot, font, line

Description automatically generated

Figure 8 Pet Table

The CREATE and INSERT scripts themselves allows you to record information for each pet, the **petTypeID** sets its species by referencing the *tblPetTypes* and both **petDOB** and **petComments** allow for the date of birth and further comment but aren’t compulsory, an example of the ‘INSERT’ is on the previous page.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 9 Pet Table Queried

1. A screenshot of a computer code

   Description automatically generated with medium confidenceEnable one owner to be shown for multiple pets.

This is achieved by creating a 1-to-many relationship between *tblOwners* and *tblPets*, in the pets table we have an **ownerID** that is a foreign key referencing the owner’s table.

This allows each pet to be associated with one owner while each owner can be associated with multiple pets.

Figure 10 Owner & Pet Tables

A screenshot of a computer

Description automatically generated with medium confidence

Figure 11 INSERT 2nd Pet into Pet Table

### Record staff information, identifying whether they are VETs or VET NURSEs.

A screenshot of a computer code

Description automatically generated with medium confidence

Figure 12 Staff and Staff Type Tables

This is handled by the *tblStaff* and *tblStaffType* tables where you can INSERT or record staff information, the staff table handles ID’s, first and second names etc the **staffTypeID** foreign key in the staff table references the primary key of the staff type table, altogether this allows you to record staff information as well as identify and categorise staff by their type.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 13 Staff & Staff Type JOIN

### Record appointment information, showing pet and staff information as well as date and time. Allow space for outcome of appointment to be recorded.

A picture containing text, screenshot, font

Description automatically generatedTo record appointment information, we have the appointments table where you can record or INSERT appointment information. The **staffID** and **petID** are both foreign keys referencing their own tables allowing you to associate a staff member and an animal for each appointment.

Figure 14 Appointment Table

You also have the ‘DATE’ and ‘TIME’ columns for storing the date and time information as well as columns for the outcome of the treatment, invoice number and payment status.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 15 Appointment Table Queried

## **SHOULD**

### Show all appointments for today (including home visits) for a given staff member. Include time, pet, owner, staff, and treatment codes and cost.

A screenshot of a computer

Description automatically generated with medium confidenceThe ‘SELECT’ clause specifies the required columns I want to retrieve from the different tables and the ‘FROM’ clause selects the main table.

Figure 16 Appointment Query

I used multiple ‘JOIN’ clauses to combine the rows from these tables based on the foreign keys from the appointments table, doing this we can join multiple tables together based on a related column/key.

Finally, the ‘WHERE’ clause will filter the results to only show appointments on the 13/11/2022 and where the staff ID is equal to ‘1’.

A picture containing text, screenshot, font, line

Description automatically generated

Figure 17 Appointment Query Execution

### Record payments made, including the invoice information, amount paid, method of payment etc. A payment can only be applied to one invoice (see Case Study). Generate a unique receipt number for each payment.

A screenshot of a computer

Description automatically generated with low confidenceTo record payments made including the required fields we can simply ‘INSERT’ into the *tlbPayments* table which has the fields already created.

The **invoiceNumber** foreign key in the payments table and **invoiceNumber** in the appointments table being ‘UNIQUE’ allows each payment to be applied to one invoice.

Figure 18 INSERT into Payments Table

A screenshot of a computer

Description automatically generated with low confidence

Figure 19 INSERT Multiple Payments

The auto-generated **paymentID** does allow for multiple payments to the same invoice but is recorded as a separate payment.

### Show staff information, showing their line managers. Include staff who do not have line managers.

In my ‘CREATE’ statement we added a **lineManagerID** to the staff table but never gave it a description in the staff type table;

A screenshot of a computer

Description automatically generated with medium confidence

Ideally I wanted **MANAGER** to be **staffTypeID** of 3 which is what I assumed would happen using ‘IDENTITY(1,1)’ however it turns out that when using this if a row is deleted or an insertion fails for whatever reason the identity value continues incrementing and does not ignore the fails or deletions but still guarantees uniqueness – to avoid this I’d have to alter or drop the table all together and remove the identity clause.

Figure 20 INSERT New Staff Type

A screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidenceAfter that we can insert a new staff member to manage the staff and then add the new member of staff as Gurkirans manager;

Figure 22 UPDATE Gurkirans Manager to New Line Manager

Figure 21 INSERT New Line Manager

Now we specify the columns required, using aliases for the same table is useful when you want to compare data from the same table – we are essentially creating a staff table and a line manager table from the original staff table.

We use the ‘FROM’ clause to specify the main table which is the staff table.

Next, we *‘LEFT JOIN’ tblStaff* to itself known as a “self-join”, this will return all the rows from the left table (S1 or staff table) and the matched rows from the right table (S2 or line manager table).

Figure 23 Staff Table Self-Join Query

And finally, the condition for the join is that the **lineManagerID** in the staff/S1 table equals a **staffID** in the manager/S2 table – meaning the member of staff in S2 is the line manager of the member of staff in S1.

Note: a ‘LEFT JOIN’ is used to return all members of staff whether they have a line manager or not, if there is no manager then the result will be ‘NULL’. *(SQL Server Tutorial*. 2021, *SQL joins*. Nd.)

### Calculate the total owed by each owner (whether invoices are overdue or not).

A screenshot of a computer

Description automatically generated with medium confidence

We ‘SELECT’ the necessary columns to be selected and then we use the ‘SUM’ aggregate function to calculate the total owed by each owner and assign it as a new column called **“totalOwed”**.

We once again use the ‘LEFT JOIN’ to join multiple tables together based on common keys, I used a ‘LEFT JOIN’ so I could return all rows from the owners table.

Finally, we use the ‘WHERE’ clause to filter the results based on the condition that the **paidStatus** is ‘N’ and then we use the ‘GROUP BY’ clause to group the results by the owners’ details.

Figure 24 Total Owed Query

To test this, we have Trevor with **ownerID** of ‘1’ who happens to have a paid appointment from the first appointment of the day which was a test with the treatment code of ‘T47’ and the cost of £33.

A screenshot of a computer

Description automatically generated with medium confidenceWe can update the appointment table to return this **paidStatus** to ‘N’:

Figure 25 Total Owed Test

We can then re-query the owners table to calculate the total owed by each owner and we should expect Trevor’s total owed to increase from 150 to 183.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 26 Total Owed Test Result

## **COULD**

### Identify any unpaid invoices dated more than 28 days ago, showing the name and address of the responsible owner. Identify the owners that owe the most.

A screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedLet’s start by creating some more data for the database, we will create two new owners with two new pets and two unpaid appointments from at least 28 days ago and then we can start querying it.

Figure 28 INSERT Test Pets

Figure 27 INSERT Test Owners

Figure 29 INSERT Test Appointments

A screenshot of a computer

Description automatically generated with medium confidence

Almost identical to the query from question 8 with a couple of exceptions, firstly we don’t use a ‘LEFT-JOIN’ as we’re not concerned about returning ALL owners just the ones with unpaid invoices over 28 days old and we also added an extra filter in the ‘WHERE’ clause using the ‘DATEADD’ function which filters the results to only the results that are -28 days from 13/11/2022.

*(Dateadd (transact-SQL) - SQL server,* 2022)

Figure 30 Query Test Appointments

A screenshot of a computer

Description automatically generated with medium confidence

Figure 31 Owners That Owe the Most

Again, very similar queries except with the additions of ‘TOP 3 WITH TIES’ and the ‘ORDER BY’ clause, top 3 will like the name suggest return the top 3 results of the **TotalOwed** which includes ties as you can see from the image above - the order of the top 3 itself is determined by the ‘ORDER BY TotalOwed DESC’ clause. *(SQL Server select top by practical examples*, 2020)

### Calculate the total amount owed to each surgery (consider how Home visits will be included).

A screenshot of a computer

Description automatically generated with medium confidence

Following the same general pattern we can select the **surgeryDetails** and the SUM of the **treatmentCost** as TotalOwed with our primary table being the surgeries table.

Next, we join the **surgeryID**’s from the surgery and appointment tables as well as the **treatmentCodes** from the treatment and appointment tables, ‘WHERE’ the paidStatus is equal to ‘N’ and finally we ‘GROUP BY’ the **surgeryDetails** itself rather than an owner.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 32 Query Amount Owed to Each Surgery

To test this, most appointments are in the Worthing Surgery or **surgeryID** of ‘1’ except one appointment at the Brighton Surgery, lets change this one appointment to the Worthing Surgery.

Figure 33 Appointment Table

A screenshot of a computer

Description automatically generated with medium confidence

Once changed we can then re-query the appointment table to check the script.

Figure 34 UPDATE Appointment Table

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 35 Re-Query Amount Owed to Each Surgery

As you can see the £45 that was owed to the Brighton Surgery has now been transferred over to the Worthing Surgery.

### Find the average income generated by each staff member per month (ignore unpaid invoices).

Firstly, lets insert some data into the appointments table for my two staff members for the previous 2 months and update the current appointment tables **paidStatus**’s to ‘Y’ or *paid*.

Note: We will leave one appointment as ‘N’ or *unpaid* for testing purposes.

A screenshot of a computer

Description automatically generated

Figure 36 INSERT New Test Data

A screenshot of a computer

Description automatically generated with medium confidenceIn September Sarah (**staffID** = 2) had 3 appointments with the treatment codes of T40, T45 and T47 with their respective costs being 100, 45 and 33 – (100+45+33)/3 = 59.33\*.

Figure 37 Query Average Income Per Month Per Employee

In November Sarah had 2 appointments with the treatment codes of T46 and T47 with their respective costs being 30 and 33 – (30+33)/2 = 31.5.

In September Gurkiran (**staffID** = 1) had 3 appointments with the treatment codes of T40, T47 and T47 with their respective costs being 100, 33 and 33 – (100+33+33)/3 = 55.33\*.

In November Gurkiran had 5 appointments with the treatment codes of T47, T45, T45, T40 and T48 with their respective costs being 33, 45, 45, 100 and 150 but we have to take into account 1 of the appointments **paidStatus** is set to ‘N’ or *unpaid* which should not return in the SQL query,that appointment had the treatment code of T47 – (45+45+100+150)/4 = 85.

Note: If you want the total income per month per employee simply replace the **“AVG(T.treatmentCost) As AverageIncomePerMonth”** to **“SUM(T.treatmentCost) As TotalIncomePerMonth**.

# **Implementation of DBMS Functionality**

## **WOULD BE NICE**

### Create a stored procedure to input new pet information, including associating them with an existing or new owner (given the owner’s name).

A screenshot of a computer

Description automatically generated

Figure 38 Creating a Stored Procedure

To create a stored procedure, we navigate to the Stored Procedure directory and right click on the “New Stored Procedure” option, it will automatically generate a template procedure in a new window for you.

To begin with we create a new procedure, name it, and then define the parameters the stored procedure will take.

‘AS BEGIN’ is the start of the stored procedures block of code where we declare **@ownerID** as an INT, we then check if owner already exists by comparing our **@ownerID**, **@ownerFName** and **@ownerLName** to their counterparts in the owner’s table – if owner exists, the **ownerID** is stored in the **@ownerID** variable and we skip to the final ‘INSERT’ and add the new pet to the **@ownerID**.

The following block of code runs if the **@ownerID** is *NULL* or doesn’t exist and simply inserts a new owner into the owner table and using the ‘SCOPE\_IDENTITY()’ function we can retrieve the last identity value that was inserted (as it is auto-generated) within the current scope and assign it to the **@ownerID** variable - once the owner is added we can proceed to the final ‘INSERT’ and add the new pet as well.

(Assaf, W.D. no date *Create a stored procedure - SQL server.*

Medewar, S. and Thul, B. 2010 *How to use scope\_identity to retrieve the last ID that was inserted.*

VanMSFT. 2023 *Scope\_identity (transact-SQL) - SQL server)*

To test our new stored procedure, we can use the ‘EXEC’ command followed by the stored procedures name to execute it with the required parameters and then we can check the owners and pets table to see if they have been updated.

A screenshot of a computer

Description automatically generated

Figure 39 Testing a Stored Procedure

### Create a trigger that will check how much a given owner owes when they make an appointment. If there are any unpaid invoices more than 28 days old, do not make the booking.

A screenshot of a computer code

Description automatically generated with medium confidence

Figure 40 Creating a Trigger

To begin with ‘CREATE TRIGGER’ followed by the name, the table you want the trigger to execute on, and declare it to execute ‘AFTER INSERT’.

Again ‘AS BEGIN’ starts the block of code for the trigger, we declare 2 INT variables - **@ownerID** and **@petID** and then we assign these variables to the newly inserted **ownerID** and **petID.**

Next we check if the owner has any unpaid appointments that are -28 days from the current date of the database (13/11/2022), this is very similar to the script from question 9 with the the difference of ‘SELECT 1’ which will select any unpaid appointment and run the ROLLBACK.

If an unpaid appointment exists the ‘ROLLBACK TRANSACTION’ begins and ‘THROWS’ an exception/error message.

*(SQL server,* no date. *Create trigger (transact-SQL), Gullezn, G. 2012, Trigger SQL server (see if a hotelroom is already booked))*

A screenshot of a computer

Description automatically generated with medium confidenceNow to test the trigger we can set 2 appointments that are 28 days or older to the **paidStatus** of ‘N’ or *unpaid* and then we can try to insert these pets into new appointments, and we can expect to see the error message.

Figure 41 Testing the Trigger

1. Create an app capable of running the queries.

I created a GUI which would have the ability to add, remove, update, and search like a basic CRUD application, unfortunately I believe firewall issues stopped me from connecting to the database.

A screen shot of a computer program

Description automatically generated with medium confidenceTo begin with I imported all required statements to allow JavaFX to work with SQL Server.

As well as declaring “con” a **Connection** object and the ‘start’ method for the application.

Next the **connectDatabase** method was a template method pulled from SQL documentation that is meant to first load the Microsoft SQL Server drivers using ‘Class.forName’.

After we define the connection URLs, I was using localdb with a server name of ‘local’ with the database called ‘BetsPetsVet’

Finally, we attempt a connection and output to console on successful connection.

getConnection is the getter method that returns the con Connection object.

Figure 42 SQL Imports

A screenshot of a computer program

Description automatically generated with low confidence

Figure 43 connectDatabase Method

A screen shot of a computer

Description automatically generated with medium confidenceUsing SceneBuilder I set up the user input fields and declare a ListView for output display and then referenced them in the FXML file and my controller class.

Figure 44 GUI Inputs

A screenshot of a computer

Description automatically generated

Figure 45 FXML Inputs

A screenshot of a computer program

Description automatically generated with medium confidenceFinally, to try and put it all together I added an EventListener to my Search button that would run my ‘handleSearch’ method when clicking it.

Figure 46 handleSearch Method

The problem I have is when I launch the application the first thing it tries to do before even opening the application window is to connect to the database, this is the error I get when launching.

Figure 47 Connection Errors

I tried enabling TCP/IP through the SQL Server Configuration Manager and set the Dynamic port to zero with the same error.

A screenshot of a computer

Description automatically generated

Figure 48 Attempt to Open SQL Server Ports

I’ve tried disabling my Firewall as well as Windows Defender Firewall to no success, I’ve also tried opening TCP/IP port 1433 (the default port for SQL Server) through registry keys which doesn’t seem to help.

# **Business Justification**

The implemented features, procedures and triggers in the database will improve the operational efficiency of ‘Bets Pets Vets’ which will keep its clients happy as well as meeting and maintaining business objectives.

To begin with the stored procedure for adding new pet information and then associating it with an existing or new owner provides a smooth and efficient way of updating and recording records. It ensures that any new pet/owner information is recorded accurately and reduces the chance of human error, potential data loss and re-enforces referential integrity.

The trigger for checking unpaid invoices before making an appointment also reduces the chance of human error by not booking appointments when money is due, for the business this will help sustain cash flow and reduce debts to clients.

Potential improvements

* Create a trigger to remind clients a day or 2 prior to appointments by SMS or email – this could help avoid missing appointments and re-scheduling but would need to integrate some form of communication.
* Create a trigger to remind clients of overdue bills, again by SMS/email or call – this could be automated to be sent out if no payment has been made after 2 weeks of appointment.
* Creating an inventory/medicine table to keep track of stock and supplies, this would include item name, quantity, expiration date and supplier information – could also create a reorder trigger on low supplied items either with automatic re-order or notifications.
* Create all new stored procedures and triggers to manage and track inventory i.e., adding new items etc.
* Report and Analytics feature on both stock, staff, and treatments, potentially add success rates to treatments, staff proficiency, how fast stock goes etc.

In conclusion, the features implemented in this database will not only greatly increase the efficiency of administrative tasks at the vet but also help avoid the risk of human error while also improving customer service quality.

# References

*Assaf, W.D. (no date) Create a stored procedure - SQL server, Create a Stored Procedure - SQL Server | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-ver16.*

*Dateadd (transact-SQL) - SQL server (2022) SQL Server | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/sql/t-sql/functions/dateadd-transact-sql?view=sql-server-ver16.*

*Gullezn, G. (2012) Trigger SQL server (see if a hotelroom is already booked), Stack Overflow. Available at: https://stackoverflow.com/questions/35014987/trigger-sql-server-see-if-a-hotelroom-is-already-booked.*

*Medewar, S. and Thul, B. (2010) How to use scope\_identity to retrieve the last ID that was inserted, Stack Overflow. Available at: https://stackoverflow.com/questions/8986278/how-to-use-scope-identity-to-retrieve-the-last-id-that-was-inserted.*

*SQL server (no date) Create trigger (transact-SQL) - SQL Server | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16.*

*SQL Server select top by practical examples (2020) SQL Server Tutorial. Available at: https://www.sqlservertutorial.net/sql-server-basics/sql-server-select-top/.*

*SQL Server Tutorial. (2021) SQL Server Self join by practical examples Available at: https://www.sqlservertutorial.net/sql-server-basics/sql-server-self-join/.*

*SQL joins. (No date) Available at: https://www.w3schools.com/sql/sql\_join.asp.*

*VanMSFT (2023) Scope\_identity (transact-SQL) - SQL server, SQL Server | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/sql/t-sql/functions/scope-identity-transact-sql?redirectedfrom=MSDN&view=sql-server-ver16.*