DESIGNING DATABASE SYSTEM FOR A LIBRARY

REPORT

BY

CHINAZU CHRISTIAN INYAMA

Table of Contents

1.0 Introduction

This report is a guide to creating a database system for a library that will store information on members, loan history, overdue fine repayments, and the library catalogue. The goal is to design a system that meets the client's needs and avoids data loss during system downtime. The report explains the design decisions and justifies them, including the creation of stored procedures, user-defined functions, views, and triggers. The report explains the benefits of the database system, including streamlined operations for the library. It summarizes the findings and recommendations for the database system design and suggests ways to improve efficiency and reliability. Overall, the report provides a thorough overview of the database system design process, including client requirements and design decisions, and recommends improvements to enhance the system.

PART 1

2.0 Database Design Decisions and Implementations

Before designing my database into 3NF form, I made some of the following assumptions

- Each member can have only one address associated with them.

- Members must have a username and password to log in to the system.
- Each member must have a unique email address associated with their account.
- Telephone numbers are not required for members and can be left blank.
- A member's membership start date is automatically set to the current date.
- A member's membership end date can be left blank if the membership is still active.
- Each item in the catalogue must have an associated item type.
- The status of a catalogue item can only be one of the following: "On Loan", "Overdue", "Available", or "Lost/Removed".
- Each loan must have a due date.
- The days overdue field in the loans table is nullable and will only be populated if the item is returned late.
- Overdue fines must be associated with a specific member.
- Each overdue fine must have an amount repaid field.
- The outstanding balance field in the overdue fines table is nullable and will only be populated if the member has not fully repaid their fine.
- Repayments must be associated with a specific member and overdue fine.
- Each repayment must have a repayment date, amount, and repayment method.

2.1 Here are the steps I took to design the Library_DesignDB to 3NF form:

I designed the database to (3NF) to eliminate data redundancy and improve efficiency.

I started implementing my database design by identifying the main entities in the system: Addresses, Members, ItemTypes , CatalogueItems, Loans, Overdue Fines, and Repayments. For each entity. I removed the functional, partial and transitive dependencies

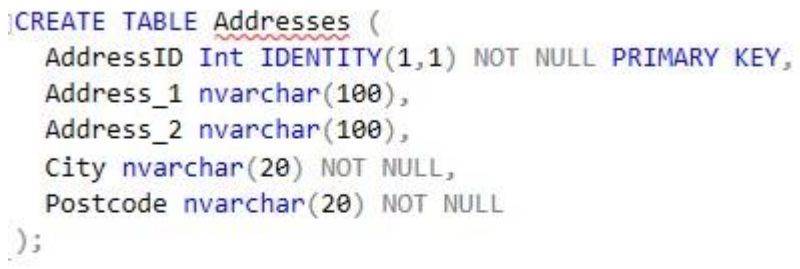
Once I had identified the entities and attributes, removed dependencies and normalized my design to 3NF, I created the tables using T-SQL statements in Microsoft SQL Server Management Studio I used primary keys and foreign key constraints to enforce referential integrity between the tables and also created my Entity Relationship Diagram (ERD) as seen in Figure 1 below.

I created my database and used the USE command to switch to the database called "library_designDB" so that any following SQL commands will be executed within this database. And I used "GO" to indicate the end of a batch of my USE statement.

2.3 Tables Created Using T-SQL Statements

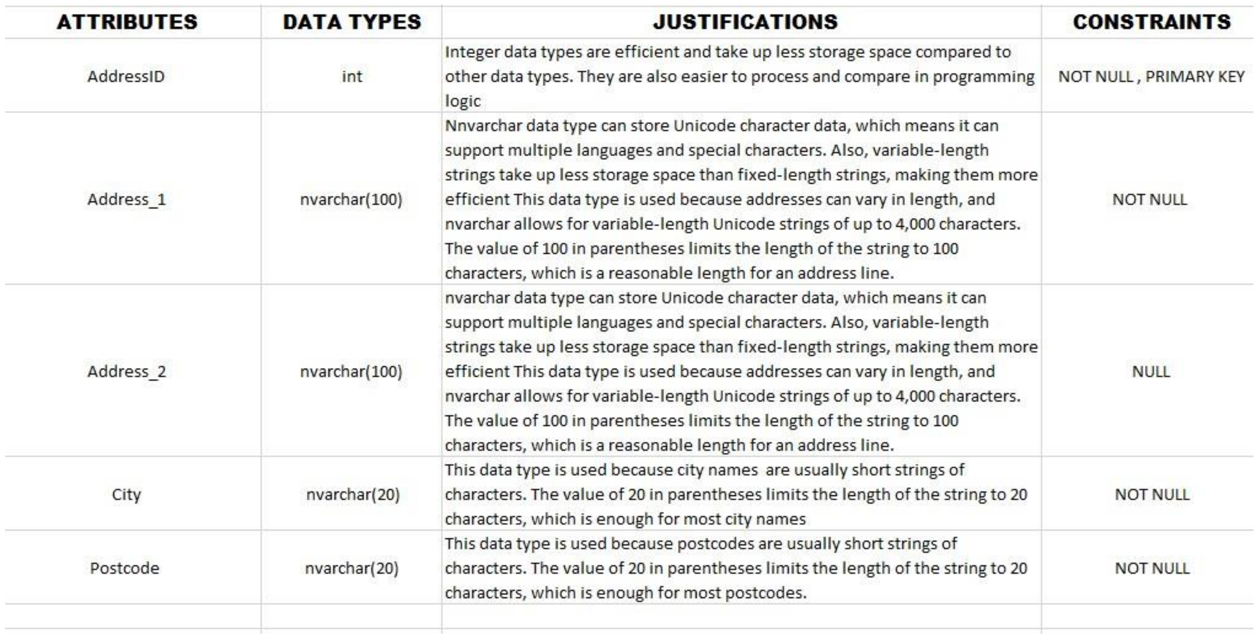
2.3.1 Addresses Table

The Addresses table stores the addresses of library members. Each address has a unique AddressID generated automatically using the IDENTITY property. The table has four columns: Address_1, Address_2, City, and Postcode, where Address_1 and Address_2 are the member's address lines, City is the city where the member lives, and Postcode is the member's postcode.



```
CREATE TABLE Addresses (
  AddressID Int IDENTITY(1,1) NOT NULL PRIMARY KEY,
  Address_1 nvarchar(100),
  Address_2 nvarchar(100),
  City nvarchar(20) NOT NULL,
  Postcode nvarchar(20) NOT NULL
);
```

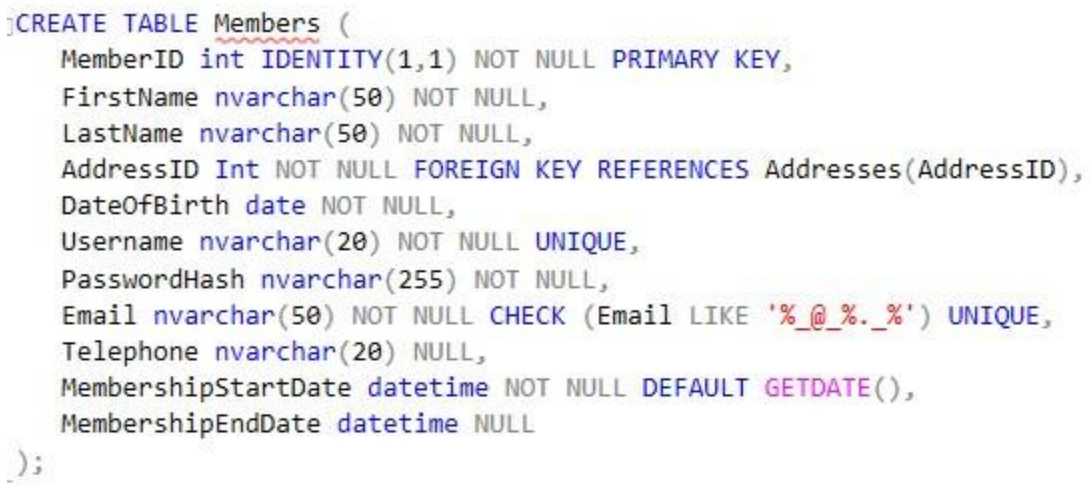
Screenshot of Justification of the data types used



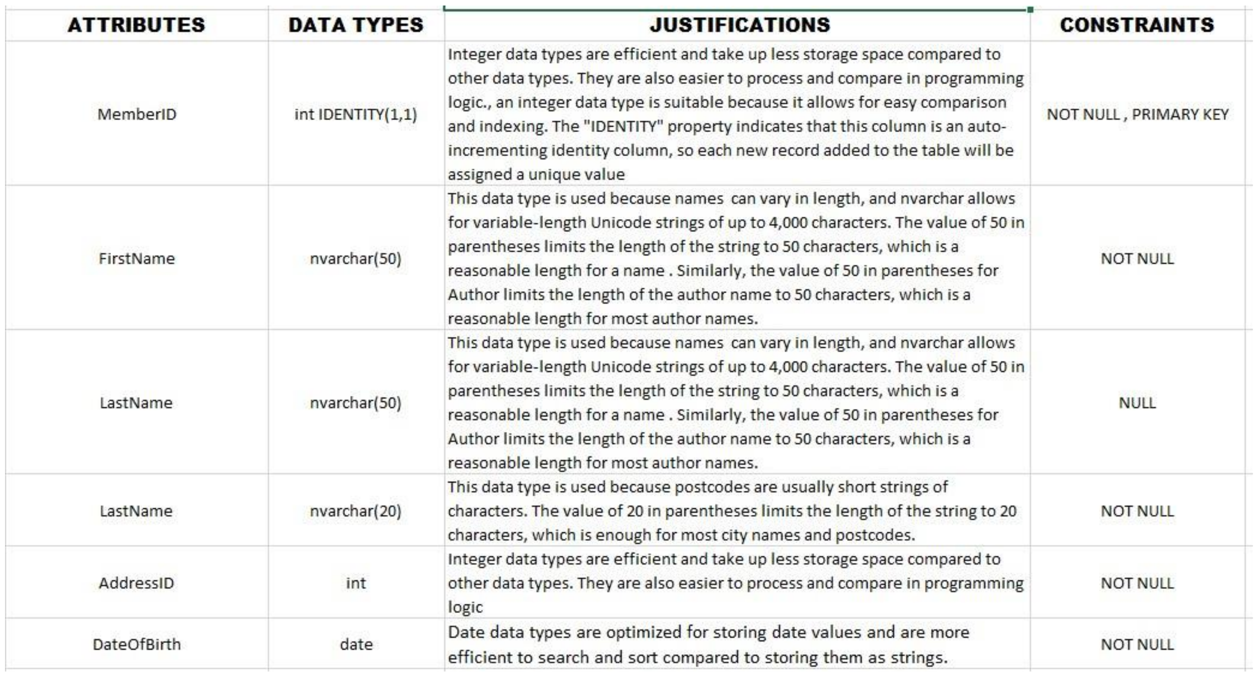
ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
AddressID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL , PRIMARY KEY
Address_1	nvarchar(100)	Nnvarchar data type can store Unicode character data, which means it can support multiple languages and special characters. Also, variable-length strings take up less storage space than fixed-length strings, making them more efficient This data type is used because addresses can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 100 in parentheses limits the length of the string to 100 characters, which is a reasonable length for an address line.	NOT NULL
Address_2	nvarchar(100)	nvarchar data type can store Unicode character data, which means it can support multiple languages and special characters. Also, variable-length strings take up less storage space than fixed-length strings, making them more efficient This data type is used because addresses can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 100 in parentheses limits the length of the string to 100 characters, which is a reasonable length for an address line.	NULL
City	nvarchar(20)	This data type is used because city names are usually short strings of characters. The value of 20 in parentheses limits the length of the string to 20 characters, which is enough for most city names	NOT NULL
Postcode	nvarchar(20)	This data type is used because postcodes are usually short strings of characters. The value of 20 in parentheses limits the length of the string to 20 characters, which is enough for most postcodes.	NOT NULL

2.3.2 Members Table

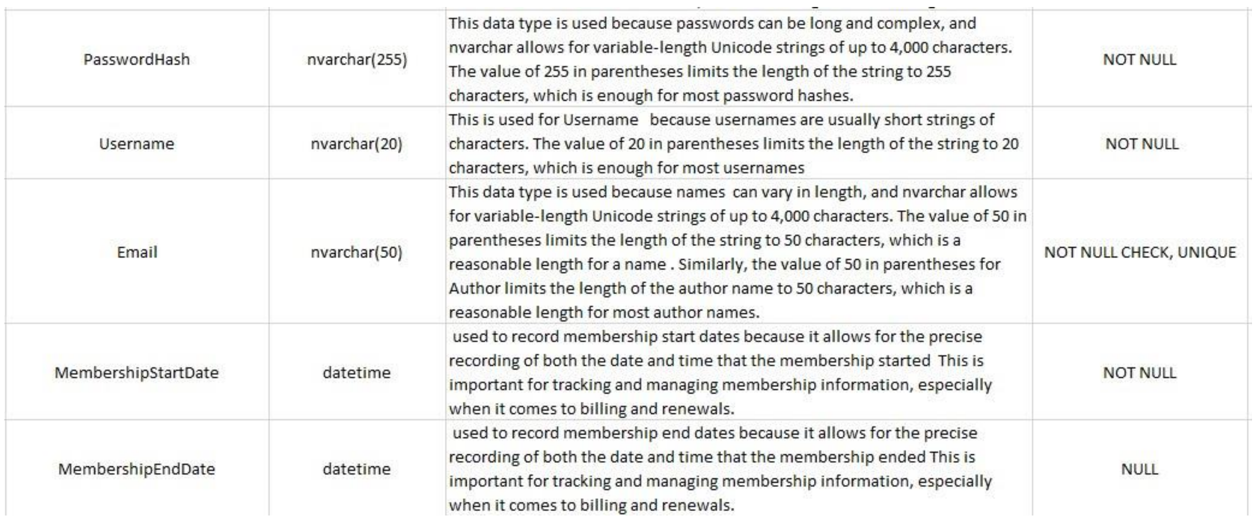
The Members table stores data about library members, including their personal information, membership start and end dates, contact details, and their addresses. The table has ten columns: MemberID, FirstName, LastName, AddressID, DateOfBirth, Username, PasswordHash, Email, Telephone, MembershipStartDate, and MembershipEndDate. The MemberID column is the primary key, and it is generated automatically using the IDENTITY property. The AddressID column is a foreign key that references the AddressID column in the Addresses table. The Username and Email columns have UNIQUE constraints to ensure that each member has a unique username and email address.



```
CREATE TABLE Members (
    MemberID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
    FirstName nvarchar(50) NOT NULL,
    LastName nvarchar(50) NOT NULL,
    AddressID Int NOT NULL FOREIGN KEY REFERENCES Addresses(AddressID),
    DateOfBirth date NOT NULL,
    Username nvarchar(20) NOT NULL UNIQUE,
    PasswordHash nvarchar(255) NOT NULL,
    Email nvarchar(50) NOT NULL CHECK (Email LIKE '%_@_%._%') UNIQUE,
    Telephone nvarchar(20) NULL,
    MembershipStartDate datetime NOT NULL DEFAULT GETDATE(),
    MembershipEndDate datetime NULL
);
```



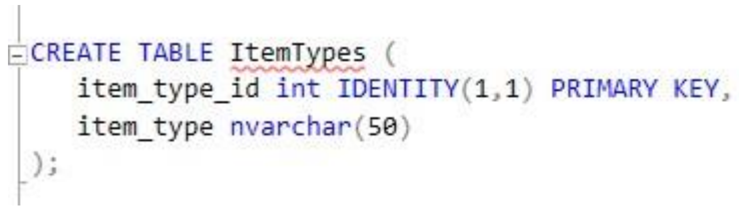
ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
MemberID	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL , PRIMARY KEY
FirstName	nvarchar(50)	This data type is used because names can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NOT NULL
LastName	nvarchar(50)	This data type is used because names can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NULL
LastName	nvarchar(20)	This data type is used because postcodes are usually short strings of characters. The value of 20 in parentheses limits the length of the string to 20 characters, which is enough for most city names and postcodes.	NOT NULL
AddressID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL
DateOfBirth	date	Date data types are optimized for storing date values and are more efficient to search and sort compared to storing them as strings.	NOT NULL



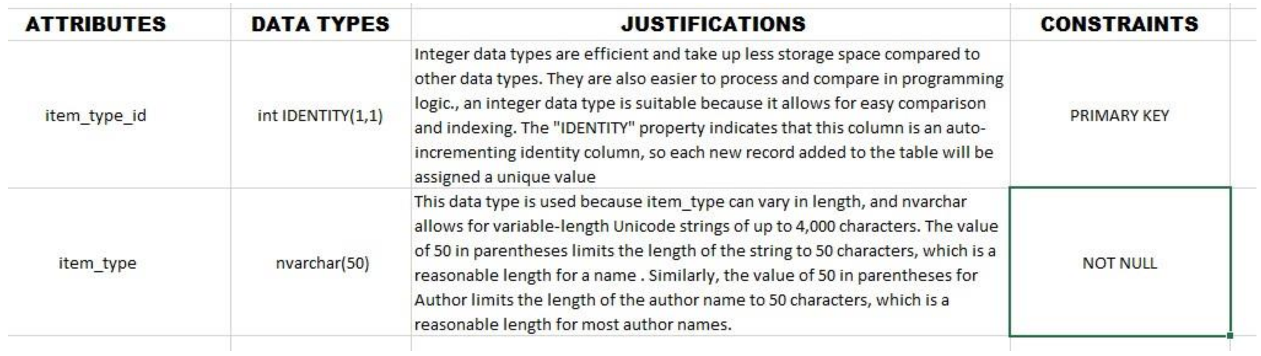
PasswordHash	nvarchar(255)	This data type is used because passwords can be long and complex, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 255 in parentheses limits the length of the string to 255 characters, which is enough for most password hashes.	NOT NULL
Username	nvarchar(20)	This is used for Username because usernames are usually short strings of characters. The value of 20 in parentheses limits the length of the string to 20 characters, which is enough for most usernames	NOT NULL
Email	nvarchar(50)	This data type is used because names can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NOT NULL CHECK, UNIQUE
MembershipStartDate	datetime	used to record membership start dates because it allows for the precise recording of both the date and time that the membership started This is important for tracking and managing membership information, especially when it comes to billing and renewals.	NOT NULL
MembershipEndDate	datetime	used to record membership end dates because it allows for the precise recording of both the date and time that the membership ended This is important for tracking and managing membership information, especially when it comes to billing and renewals.	NULL

2.3.3 ItemTypes Table

The ItemTypes table stores data about the types of items available in the library, such as books, CDs, DVDs, etc. The table has two columns: item_type_id and item_type, where item_type_id is the primary key generated automatically using the IDENTITY property, and item_type is the name of the item type.



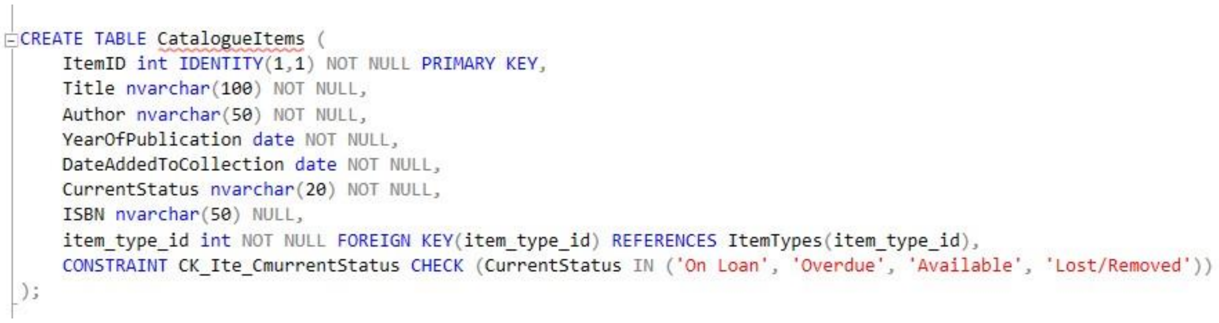
```
CREATE TABLE ItemTypes (
    item_type_id int IDENTITY(1,1) PRIMARY KEY,
    item_type nvarchar(50)
);
```



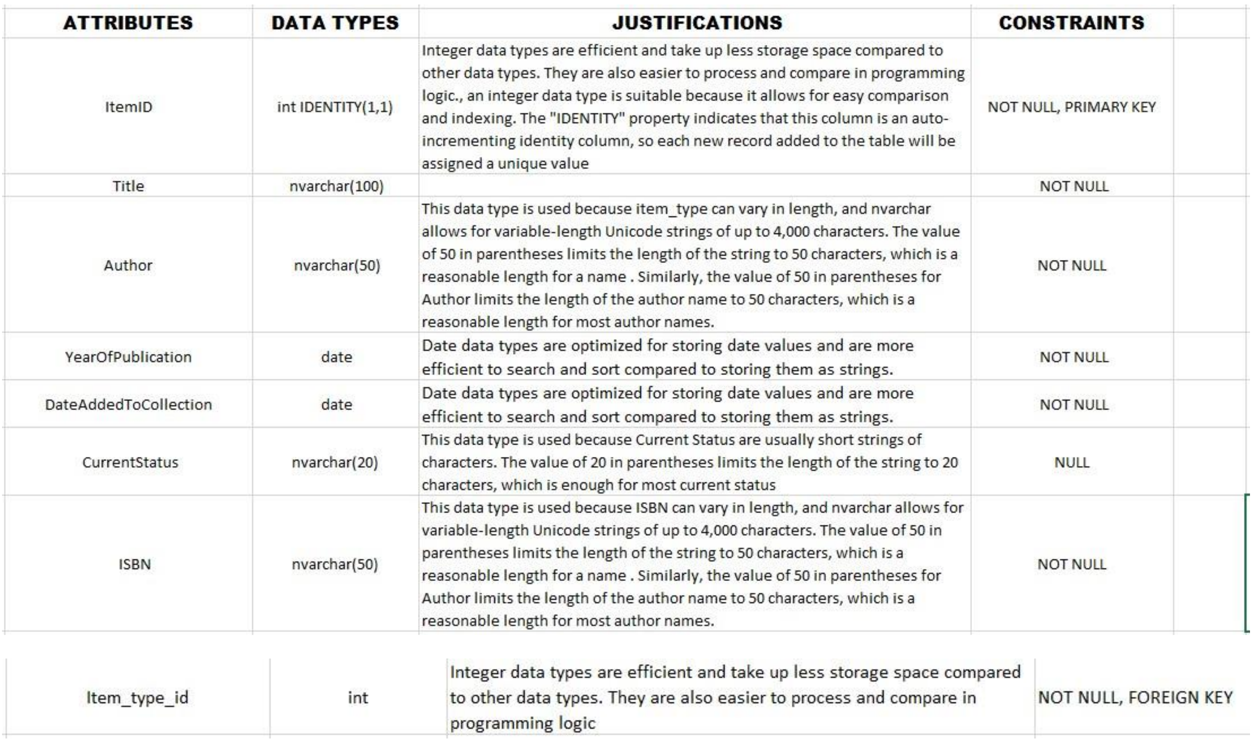
ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
item_type_id	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	PRIMARY KEY
item_type	nvarchar(50)	This data type is used because item_type can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NOT NULL

2.3.4 CatalogueItems Table

The CatalogueItems table stores data about the library items, including their titles, authors, publication year, date added to the collection, current status, ISBN, and item type. The table has nine columns: ItemID, Title, Author, YearOfPublication, DateAddedToCollection, CurrentStatus, ISBN, item_type_id, and the CHECK constraint ensures that the CurrentStatus column only contains specific values: On Loan, Overdue, Available, and Lost/Removed. The ItemID column is the primary key generated automatically using the IDENTITY property, and the item_type_id column is a foreign key that references the item_type_id column in the ItemTypes table.



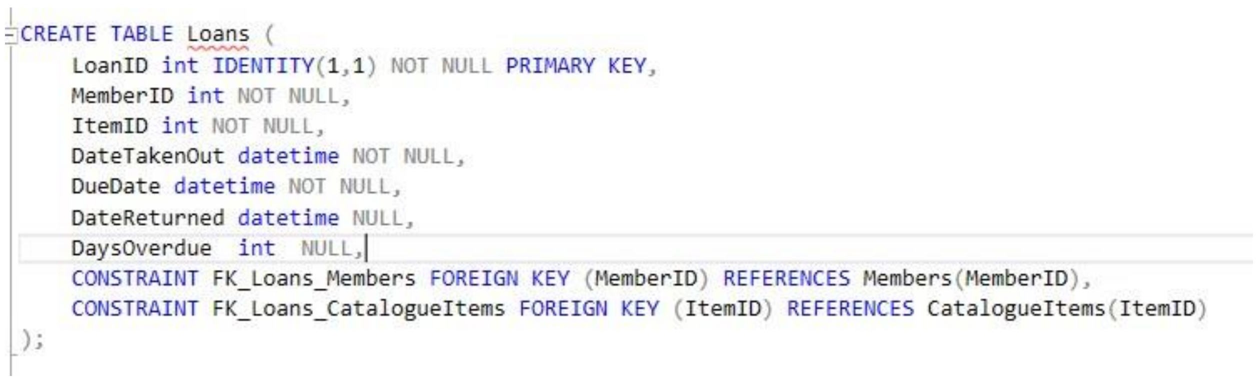
```
CREATE TABLE CatalogueItems (
    ItemID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
    Title nvarchar(100) NOT NULL,
    Author nvarchar(50) NOT NULL,
    YearOfPublication date NOT NULL,
    DateAddedToCollection date NOT NULL,
    CurrentStatus nvarchar(20) NOT NULL,
    ISBN nvarchar(50) NULL,
    item_type_id int NOT NULL FOREIGN KEY(item_type_id) REFERENCES ItemTypes(item_type_id),
    CONSTRAINT CK_Ite_CmurrentStatus CHECK (CurrentStatus IN ('On Loan', 'Overdue', 'Available', 'Lost/Removed'))
);
```



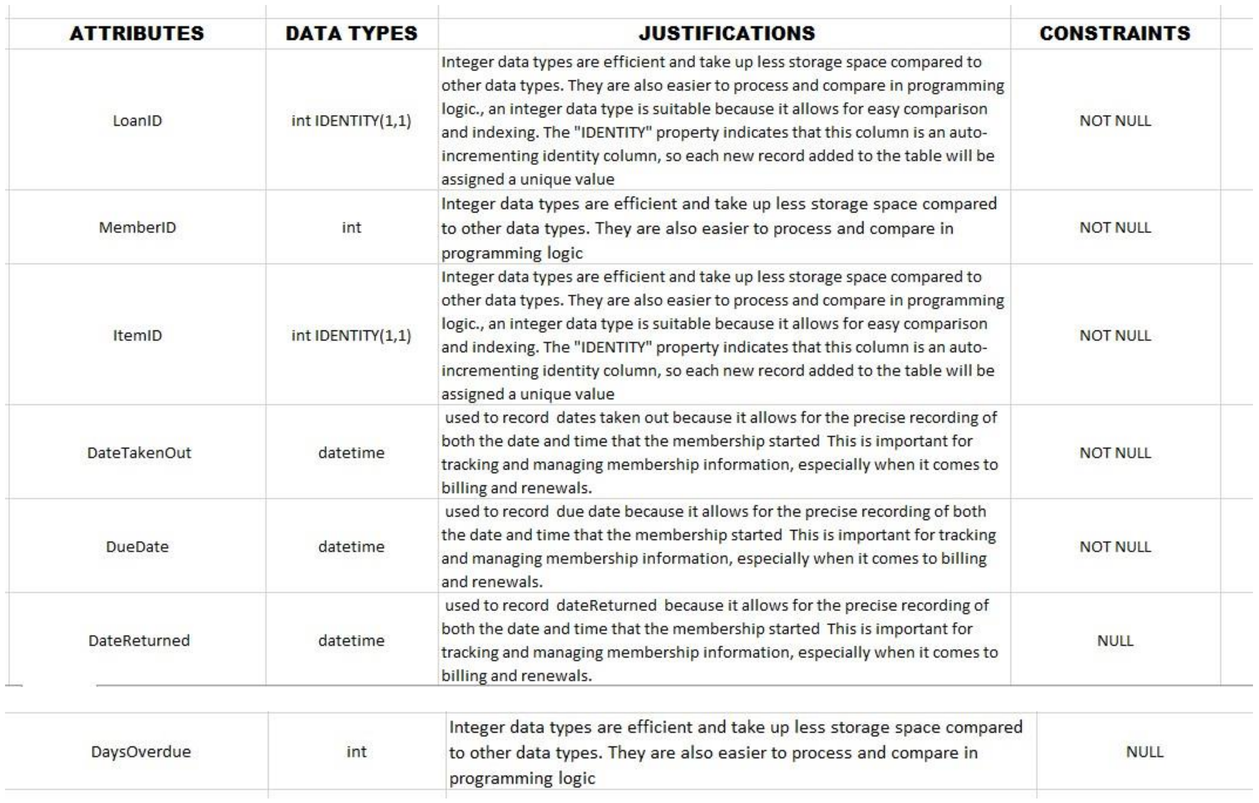
ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
ItemID	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL, PRIMARY KEY
Title	nvarchar(100)		NOT NULL
Author	nvarchar(50)	This data type is used because item_type can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NOT NULL
YearOfPublication	date	Date data types are optimized for storing date values and are more efficient to search and sort compared to storing them as strings.	NOT NULL
DateAddedToCollection	date	Date data types are optimized for storing date values and are more efficient to search and sort compared to storing them as strings.	NOT NULL
CurrentStatus	nvarchar(20)	This data type is used because Current Status are usually short strings of characters. The value of 20 in parentheses limits the length of the string to 20 characters, which is enough for most current status	NULL
ISBN	nvarchar(50)	This data type is used because ISBN can vary in length, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 50 in parentheses limits the length of the string to 50 characters, which is a reasonable length for a name . Similarly, the value of 50 in parentheses for Author limits the length of the author name to 50 characters, which is a reasonable length for most author names.	NOT NULL
Item_type_id	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL, FOREIGN KEY

2.3.5 Loans Table

respectively. The Loans table stores data about the loans made by library members, including the date the item was taken out, the due date, the date returned, and the days overdue. The table has seven columns: LoanID, MemberID, ItemID, DateTakenOut, DueDate, DateReturned, and DaysOverdue. The LoanID column is the primary key generated automatically using the IDENTITY property, and the MemberID and ItemID columns are foreign keys that reference the MemberID and ItemID columns in the Members and CatalogueItems tables,



```sql
CREATE TABLE Loans (
    LoanID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
    MemberID int NOT NULL,
    ItemID int NOT NULL,
    DateTakenOut datetime NOT NULL,
    DueDate datetime NOT NULL,
    DateReturned datetime NULL,
    DaysOverdue  int  NULL,
    CONSTRAINT FK_Loans_Members FOREIGN KEY (MemberID) REFERENCES Members(MemberID),
    CONSTRAINT FK_Loans_CatalogueItems FOREIGN KEY (ItemID) REFERENCES CatalogueItems(ItemID)
);
```

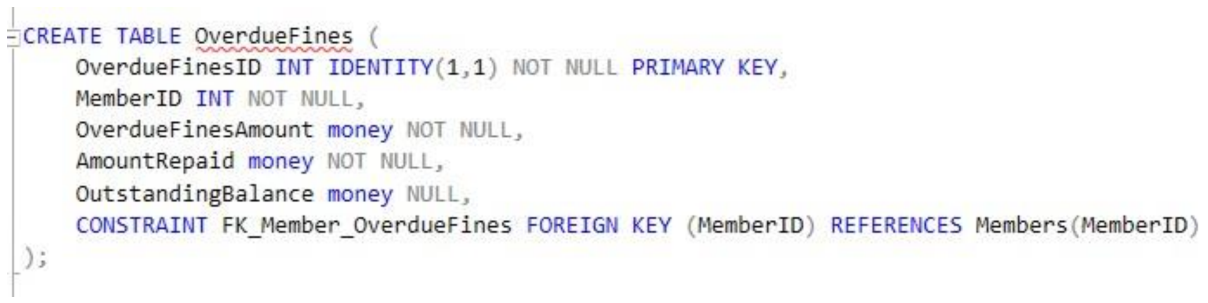


ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
LoanID	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL
MemberID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL
ItemID	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL
DateTakenOut	datetime	used to record dates taken out because it allows for the precise recording of both the date and time that the membership started This is important for tracking and managing membership information, especially when it comes to billing and renewals.	NOT NULL
DueDate	datetime	used to record due date because it allows for the precise recording of both the date and time that the membership started This is important for tracking and managing membership information, especially when it comes to billing and renewals.	NOT NULL
DateReturned	datetime	used to record dateReturned because it allows for the precise recording of both the date and time that the membership started This is important for tracking and managing membership information, especially when it comes to billing and renewals.	NULL
DaysOverdue	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NULL

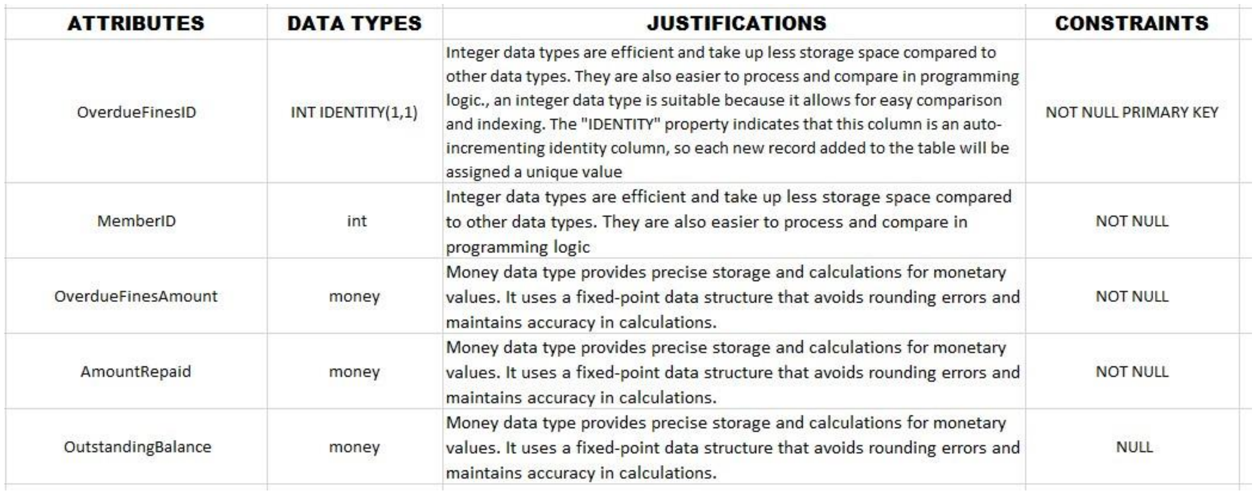
2.3.6 OverdueFines Table

The OverdueFines table stores data about the overdue fines imposed on library members, including the member ID, overdue fines amount, amount repaid, and outstanding balance. The table has four columns:

OverdueFinesID, MemberID, OverdueFinesAmount. The T-SQL statement and justification of the data types I used is seen below



```
CREATE TABLE OverdueFines (
    OverdueFinesID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
    MemberID INT NOT NULL,
    OverdueFinesAmount money NOT NULL,
    AmountRepaid money NOT NULL,
    OutstandingBalance money NULL,
    CONSTRAINT FK_Member_OverdueFines FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);
```



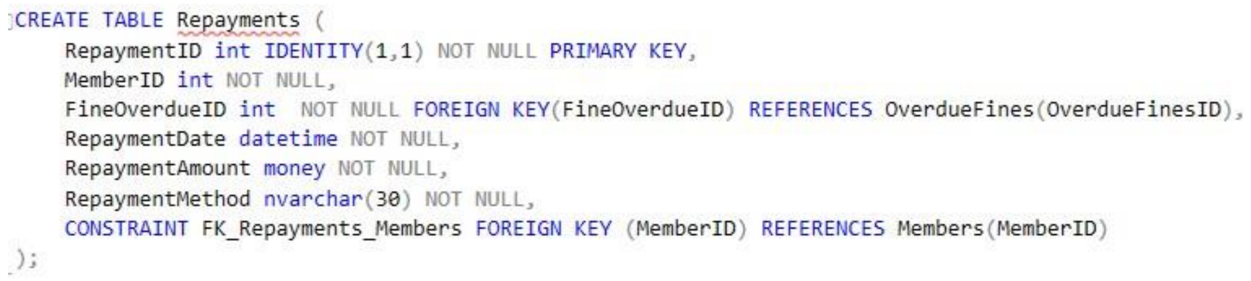
ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
OverdueFinesID	INT IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL PRIMARY KEY
MemberID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL
OverdueFinesAmount	money	Money data type provides precise storage and calculations for monetary values. It uses a fixed-point data structure that avoids rounding errors and maintains accuracy in calculations.	NOT NULL
AmountRepaid	money	Money data type provides precise storage and calculations for monetary values. It uses a fixed-point data structure that avoids rounding errors and maintains accuracy in calculations.	NOT NULL
OutstandingBalance	money	Money data type provides precise storage and calculations for monetary values. It uses a fixed-point data structure that avoids rounding errors and maintains accuracy in calculations.	NULL

2.3.7 Repayments table

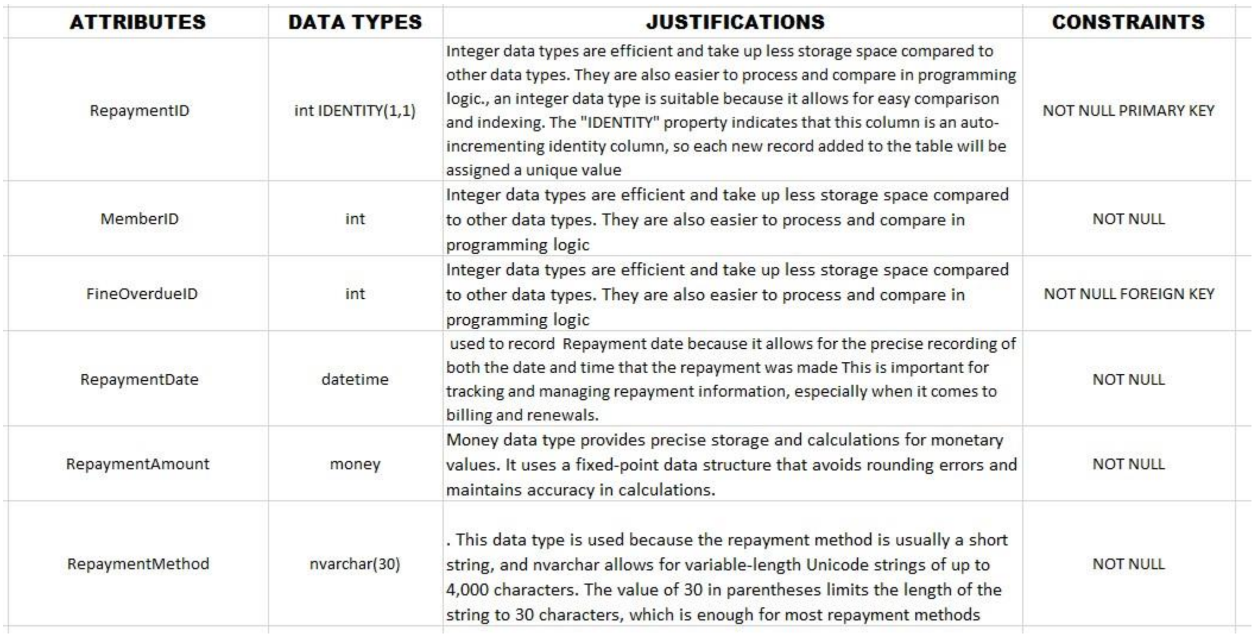
The Repayments table stores information about repayments made by members for overdue fines. Each repayment has a unique

RepaymentID and is associated with a member and an overdue fine.

Other attributes include the repayment date, amount, and method. The Repayments table contains foreign key constraints to the Members and Overdue Fines tables.

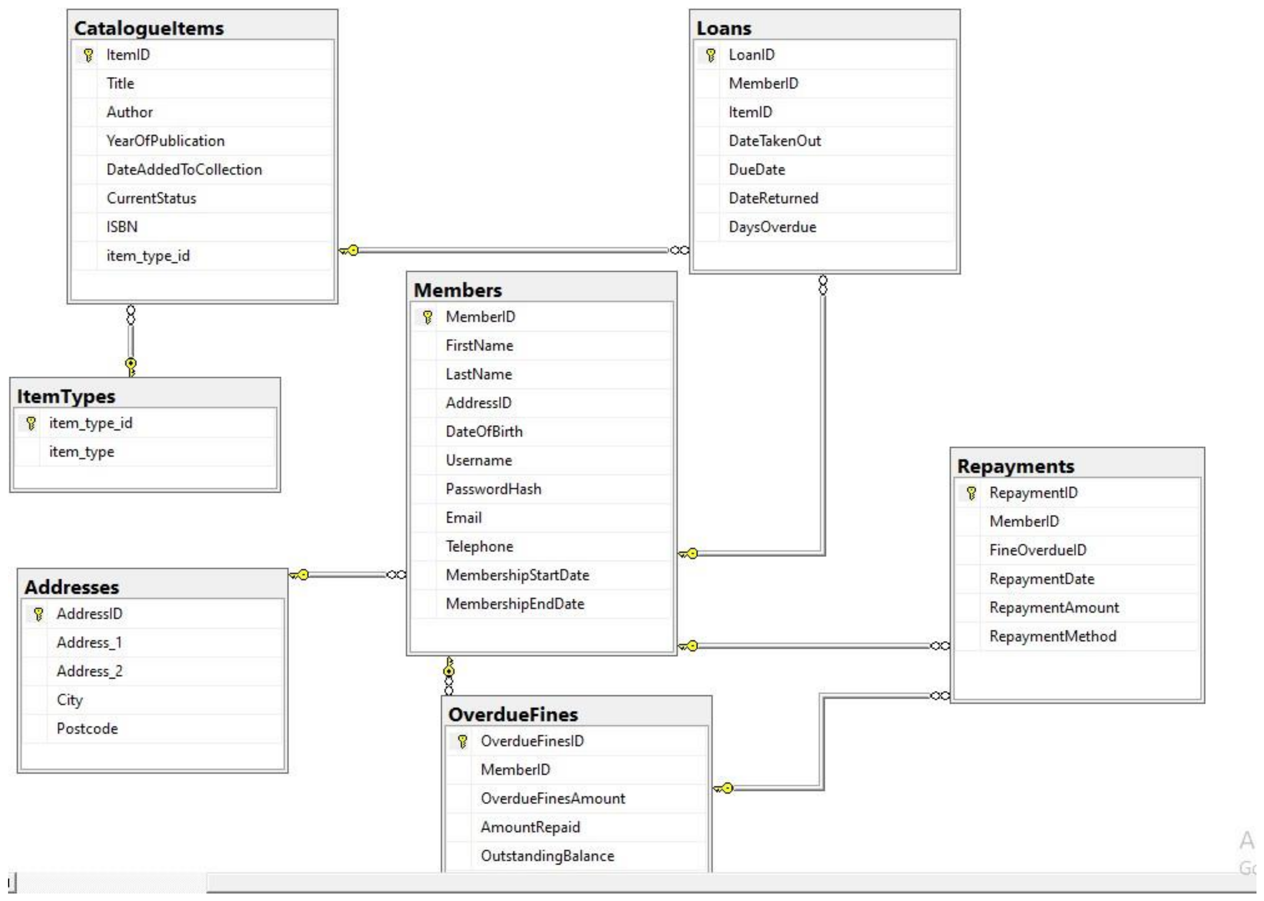


```
CREATE TABLE Repayments (
    RepaymentID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
    MemberID int NOT NULL,
    FineOverdueID int  NOT NULL FOREIGN KEY(FineOverdueID) REFERENCES OverdueFines(OverdueFinesID),
    RepaymentDate datetime NOT NULL,
    RepaymentAmount money NOT NULL,
    RepaymentMethod nvarchar(30) NOT NULL,
    CONSTRAINT FK_Repayments_Members FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);
```



ATTRIBUTES	DATA TYPES	JUSTIFICATIONS	CONSTRAINTS
RepaymentID	int IDENTITY(1,1)	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic., an integer data type is suitable because it allows for easy comparison and indexing. The "IDENTITY" property indicates that this column is an auto-incrementing identity column, so each new record added to the table will be assigned a unique value	NOT NULL PRIMARY KEY
MemberID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL
FineOverdueID	int	Integer data types are efficient and take up less storage space compared to other data types. They are also easier to process and compare in programming logic	NOT NULL FOREIGN KEY
RepaymentDate	datetime	used to record Repayment date because it allows for the precise recording of both the date and time that the repayment was made This is important for tracking and managing repayment information, especially when it comes to billing and renewals.	NOT NULL
RepaymentAmount	money	Money data type provides precise storage and calculations for monetary values. It uses a fixed-point data structure that avoids rounding errors and maintains accuracy in calculations.	NOT NULL
RepaymentMethod	nvarchar(30)	. This data type is used because the repayment method is usually a short string, and nvarchar allows for variable-length Unicode strings of up to 4,000 characters. The value of 30 in parentheses limits the length of the string to 30 characters, which is enough for most repayment methods	NOT NULL

2.4 Figure 1: Entity Relationship Diagram (ERD) for the Library_DesignDB



In the above ER diagram, there are several tables that are related to each other through various one-to-many and many-to-one relationships.

Members table: has a one-to-many relationship with the Addresses table because each member can have only one address, but multiple members can have the same address.

Addresses table: has a one-to-many relationship with the Members table because each address can be associated with multiple members.

ItemTypes table: has a one-to-many relationship with the CatalogueItems table because each item type can be associated with multiple catalogue items, but each catalogue item can have only one item type.

CatalogueItems table: It has a one-to-many relationship with the Loans table because each catalogue item can be associated with multiple loans, but each loan can only be associated with one catalogue item.

It also has a many-to-one relationship with the ItemTypes table.

Loans table: has a many-to-one relationship with both the Members table and the CatalogueItems table because each loan can be associated with one member and one catalogue item.

OverdueFines table: It has a many-to-one relationship with the Members table because each overdue fine can be associated with one member.

Repayments table: has a many-to-one relationship with both the Members table and the OverdueFines table because each repayment can be associated with one member and one overdue fine.

2.5 PRIMARY AND FOREIGN KEY COLUMNS

In relational databases, primary keys and foreign keys are important constraints that maintain data integrity and table relationships. A primary key is a unique identifier for each record in a table, which can be a single column or a combination of columns, and its value cannot be null or duplicated. It is used to identify a record and create relationships with other tables. On the other hand, a foreign key is a column or a set of columns that refers to the primary key of another table, and it ensures referential integrity by preventing records from being added to a table unless the foreign key exists in the referenced table. When a record is deleted from the referenced table, all the related records in the table containing the foreign key are either deleted or updated accordingly.

2.6 Justification for using primary and foreign keys in the tables:

2.6.1 Data Integrity: Primary keys and foreign keys ensure data integrity by preventing duplication, ensuring uniqueness, and maintaining relationships between tables. By using primary keys, each record can be uniquely identified, and foreign keys ensure that only valid values are entered into related tables, preventing inconsistencies.

2.6.2 Referential Integrity: Foreign keys ensure that the relationships between tables are maintained, and the data is consistent. By enforcing referential integrity, it is ensured that data is not orphaned, and any changes made to the referenced table are reflected in the referencing table.

2.6.3 Data Consistency: Primary keys and foreign keys help maintain data consistency by ensuring that only valid data is entered into the tables, and there are no inconsistencies between related tables.

Efficiency: By using primary keys and foreign keys, it becomes easier and more efficient to query the database and retrieve data from multiple tables. This is because the relationships between tables are clearly defined and can be easily navigated.

The primary and foreign key columns I used in the tables are as follows:

Addresses

Primary Key: AddressID (Int)

Members

Primary Key: MemberID (Int)

Foreign Key: AddressID (Int), references Addresses(AddressID)

ItemTypes

Primary Key: item_type_id (Int)

CatalogueItems

Primary Key: ItemID (Int)

Foreign Key: item_type_id (Int), references ItemTypes(item_type_id)

Loans

Primary Key: LoanID (Int)

Foreign Key: MemberID (Int), references Members(MemberID)

Foreign Key: ItemID (Int), references CatalogueItems(ItemID)

OverdueFines

Primary Key: OverdueFinesID (Int)

Foreign Key: MemberID (Int), references Members(MemberID)

Repayments

Primary Key: RepaymentID (Int)

Foreign Key: MemberID (Int), references Members(MemberID)

Foreign Key: FineOverdueID (Int), references
OverdueFines(OverdueFinesID)

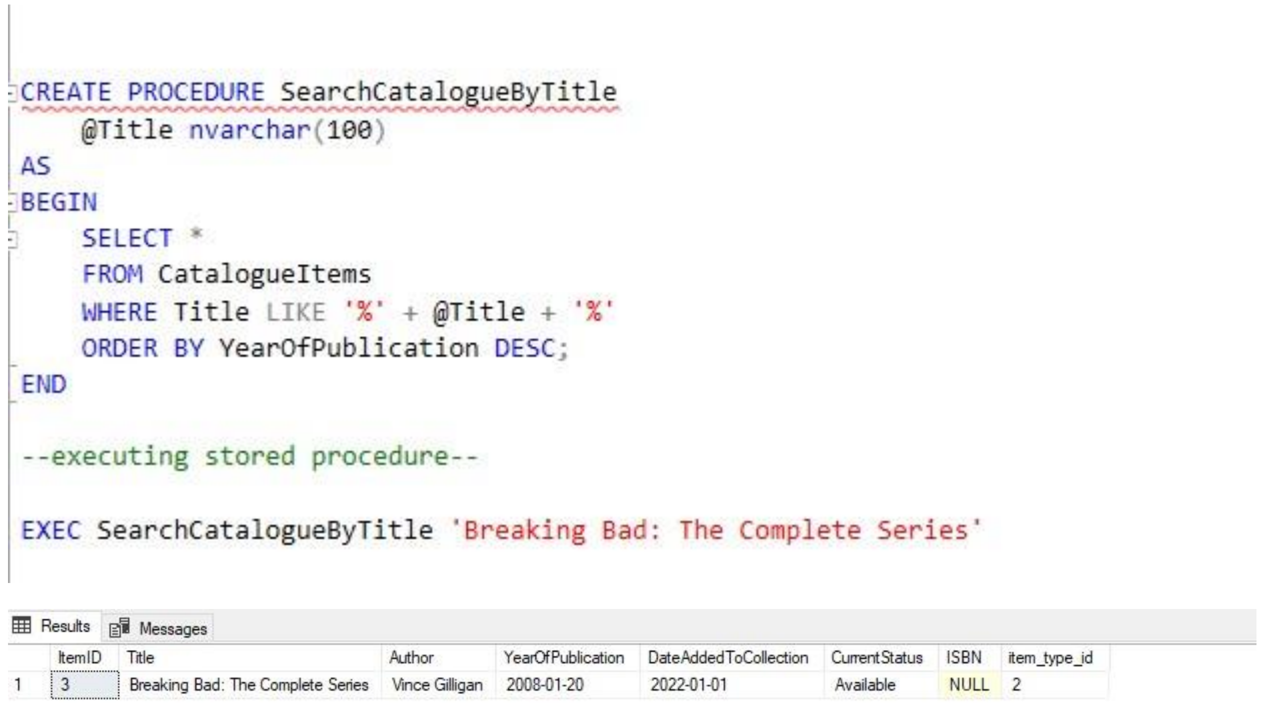
Overall, I used these primary keys and foreign keys is crucial for maintaining data integrity, referential integrity, data consistency, and query efficiency in relational databases.

PART 2

3.1

2 A

Query to Search the catalogue for matching character strings by title.



```
CREATE PROCEDURE SearchCatalogueByTitle
    @Title nvarchar(100)
AS
BEGIN
    SELECT *
    FROM CatalogueItems
    WHERE Title LIKE '%' + @Title + '%'
    ORDER BY YearOfPublication DESC;
END

--executing stored procedure--

EXEC SearchCatalogueByTitle 'Breaking Bad: The Complete Series'
```

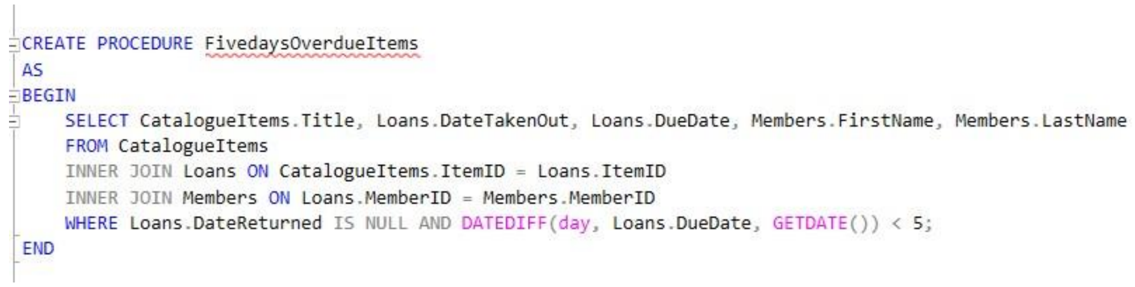
	ItemID	Title	Author	YearOfPublication	DateAddedToCollection	CurrentStatus	ISBN	item_type_id
1	3	Breaking Bad: The Complete Series	Vince Gilligan	2008-01-20	2022-01-01	Available	NULL	2

In this code I created a stored procedure named
"SearchCatalogueByTitle" that takes a title parameter and returns all items from the CatalogueItems table whose title matches the provided string . I performed the search by using the LIKE operator with the "%" wildcard characters on both sides of the search string to match any titles that contain the search string. The items are ordered by their year of publication in descending order. I use the

EXEC statement at the end to call the stored procedure with the title "Breaking Bad: The Complete Series".

2 B

Query to return a full list of all items currently on loan which have a due date of less than five days from the current date

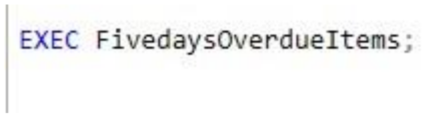


```
CREATE PROCEDURE FivedaysOverdueItems
AS
BEGIN
    SELECT CatalogueItems.Title, Loans.DateTakenOut, Loans.DueDate, Members.FirstName, Members.LastName
    FROM CatalogueItems
    INNER JOIN Loans ON CatalogueItems.ItemID = Loans.ItemID
    INNER JOIN Members ON Loans.MemberID = Members.MemberID
    WHERE Loans.DateReturned IS NULL AND DATEDIFF(day, Loans.DueDate, GETDATE()) < 5;
END
```

I used this stored procedure to retrieve a list of items that are five days overdue. It joins the CatalogueItems, Loans, and Members tables and filters the results by checking that the item has not been returned (i.e., the DateReturned is NULL) and that the difference between the due date and the current date is less than five days.

The output includes the title of the overdue item, the date it was taken out, the due date, and the first and last names of the member who borrowed it.

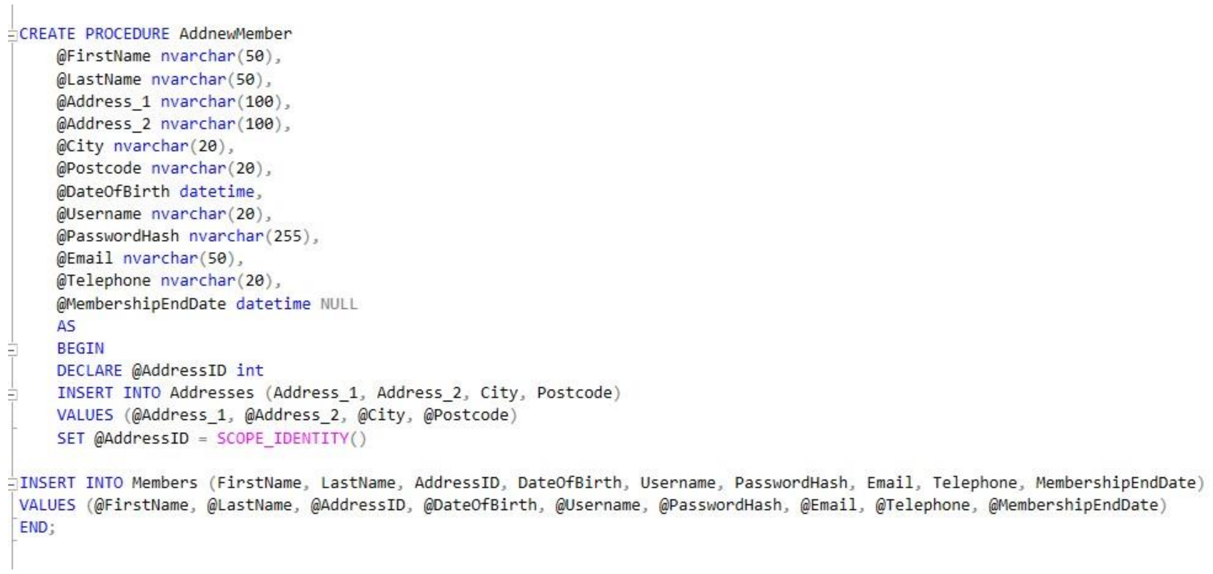
I executed this procedure by using the command below



```
EXEC FivedaysOverdueItems;
```

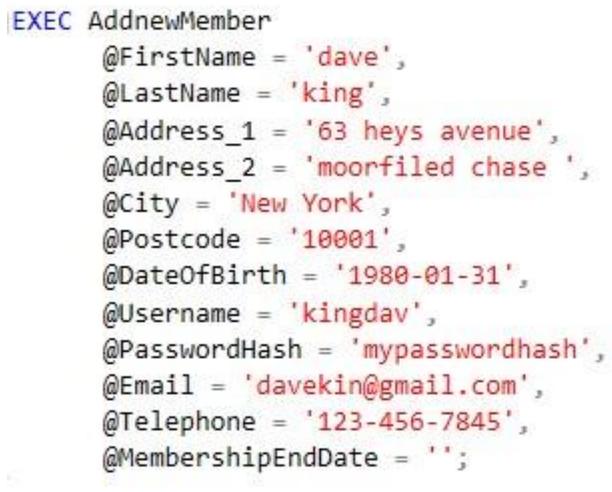
2 C

Insert a new member into the database



```
CREATE PROCEDURE AddnewMember
    @FirstName nvarchar(50),
    @LastName nvarchar(50),
    @Address_1 nvarchar(100),
    @Address_2 nvarchar(100),
    @City nvarchar(20),
    @Postcode nvarchar(20),
    @DateOfBirth datetime,
    @Username nvarchar(20),
    @PasswordHash nvarchar(255),
    @Email nvarchar(50),
    @Telephone nvarchar(20),
    @MembershipEndDate datetime NULL
    AS
    BEGIN
    DECLARE @AddressID int
    INSERT INTO Addresses (Address_1, Address_2, City, Postcode)
    VALUES (@Address_1, @Address_2, @City, @Postcode)
    SET @AddressID = SCOPE_IDENTITY()

INSERT INTO Members (FirstName, LastName, AddressID, DateOfBirth, Username, PasswordHash, Email, Telephone, MembershipEndDate)
VALUES (@FirstName, @LastName, @AddressID, @DateOfBirth, @Username, @PasswordHash, @Email, @Telephone, @MembershipEndDate)
END;
```

This code creates a stored procedure to add a new member to the library database. It takes in several input parameters The stored procedure inserts the address details into the Addresses table and retrieves the new AddressID. It then inserts the member's details into the Members table, using the retrieved AddressID to link the member to their address.

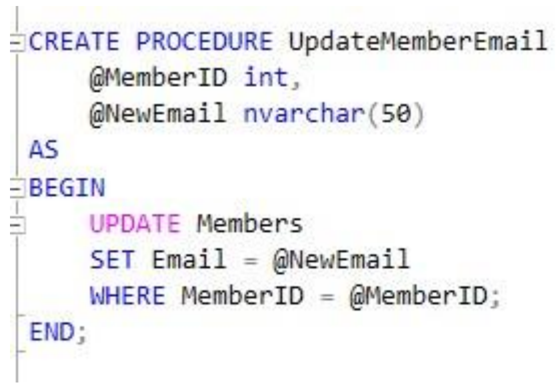


```
EXEC AddnewMember
        @FirstName = 'dave',
        @LastName = 'king',
        @Address_1 = '63 heys avenue',
        @Address_2 = 'moorfiled chase ',
        @City = 'New York',
        @Postcode = '10001',
        @DateOfBirth = '1980-01-31',
        @Username = 'kingdav',
        @PasswordHash = 'mypasswordhash',
        @Email = 'davekin@gmail.com',
        @Telephone = '123-456-7845',
        @MembershipEndDate = '';
```

I used the EXEC command to execute the stored procedure called "AddnewMember" and insert a new member into the "Members" table in a database.

2 D

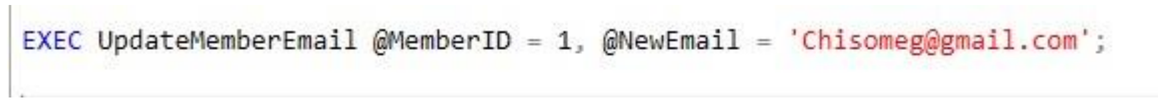
Query to update the details of an existing member



```
CREATE PROCEDURE UpdateMemberEmail
    @MemberID int,
    @NewEmail nvarchar(50)
AS
BEGIN
    UPDATE Members
    SET Email = @NewEmail
    WHERE MemberID = @MemberID;
END;
```

This stored procedure takes in two parameters: the MemberID of the member whose email is to be updated, and the new email address, NewEmail. It then updates the Email column in the Members table for the specified member with the new email address.

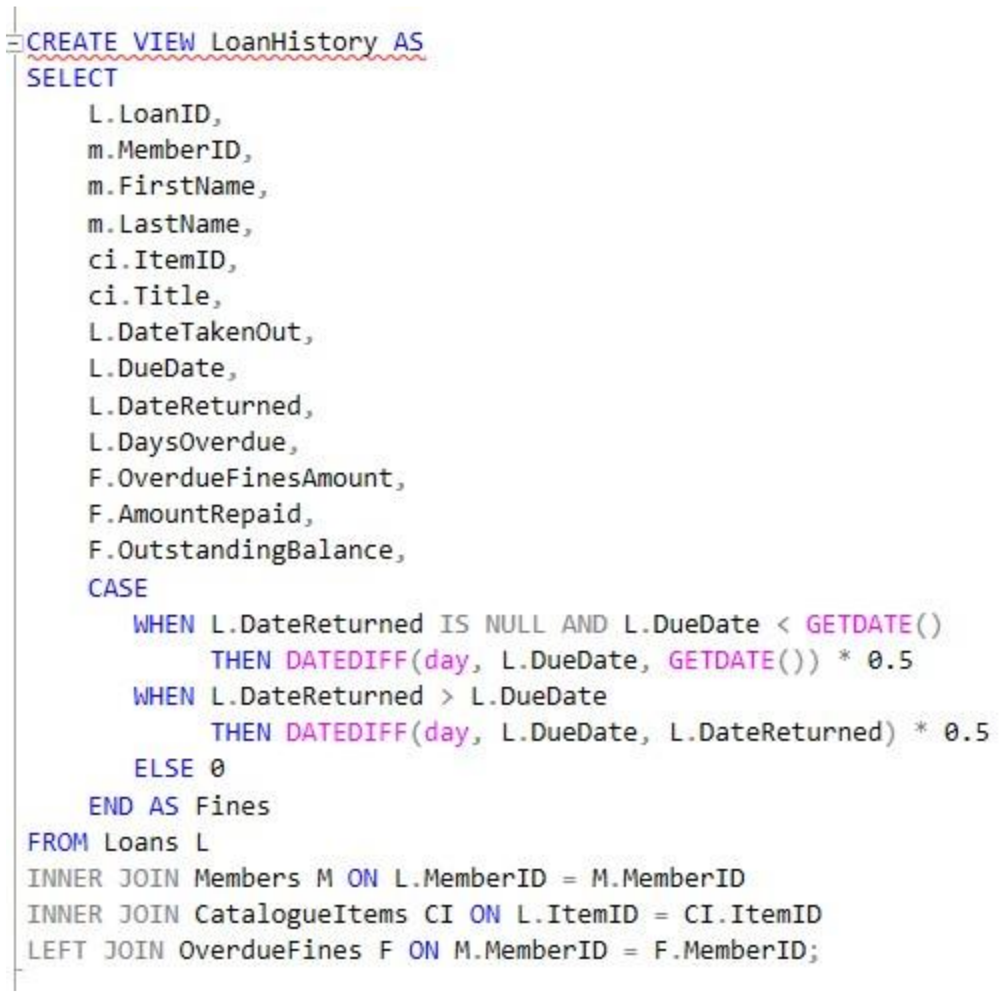
To call this stored procedure and update the email address for a member with a specific MemberID, I used the code below



```
EXEC UpdateMemberEmail @MemberID = 1, @NewEmail = 'Chisomeg@gmail.com';
```

3.2 QUESTION 3

Query to view the loan history, showing all previous and current loans, and including details of the item borrowed, borrowed date, due date and any associated fines for each loan.

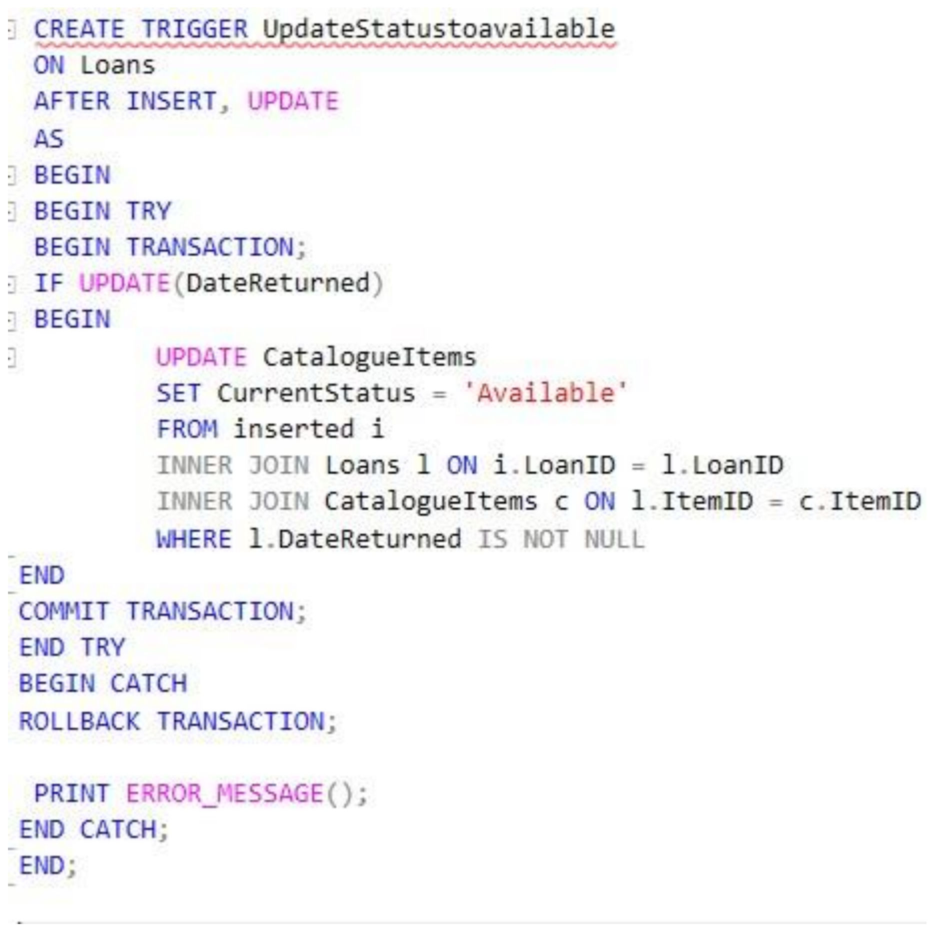


```
CREATE VIEW LoanHistory AS
SELECT
    L.LoanID,
    m.MemberID,
    m.FirstName,
    m.LastName,
    ci.ItemID,
    ci.Title,
    L.DateTakenOut,
    L.DueDate,
    L.DateReturned,
    L.DaysOverdue,
    F.OverdueFinesAmount,
    F.AmountRepaid,
    F.OutstandingBalance,
    CASE
       WHEN L.DateReturned IS NULL AND L.DueDate < GETDATE()
            THEN DATEDIFF(day, L.DueDate, GETDATE()) * 0.5
       WHEN L.DateReturned > L.DueDate
            THEN DATEDIFF(day, L.DueDate, L.DateReturned) * 0.5
       ELSE 0
    END AS Fines
FROM Loans L
INNER JOIN Members M ON L.MemberID = M.MemberID
INNER JOIN CatalogueItems CI ON L.ItemID = CI.ItemID
LEFT JOIN OverdueFines F ON M.MemberID = F.MemberID;
```

The code creates a "LoanHistory" view using data from different tables to show information about library loans, such as member name, item title, loan date, due date, return date, and fines. Tables like Loans, Members, CatalogueItems, and OverdueFines are joined using JOIN clauses, including a left join to show data from OverdueFines even if there's no matching record. The view calculates fines using a CASE statement based on the number of days overdue or late. The LoanHistory view is a useful tool for library staff to access loan and fine information for all members in one place.

3.3 QUESTION 4

Query to Create a trigger so that the current status of an item automatically updates to Available when the book is returned



```
CREATE TRIGGER UpdateStatustoavailable
 ON Loans
 AFTER INSERT, UPDATE
 AS
 BEGIN
 BEGIN TRY
 BEGIN TRANSACTION;
 IF UPDATE(DateReturned)
 BEGIN
         UPDATE CatalogueItems
         SET CurrentStatus = 'Available'
         FROM inserted i
         INNER JOIN Loans l ON i.LoanID = l.LoanID
         INNER JOIN CatalogueItems c ON l.ItemID = c.ItemID
         WHERE l.DateReturned IS NOT NULL
END
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
ROLLBACK TRANSACTION;

 PRINT ERROR_MESSAGE();
END CATCH;
END;
```

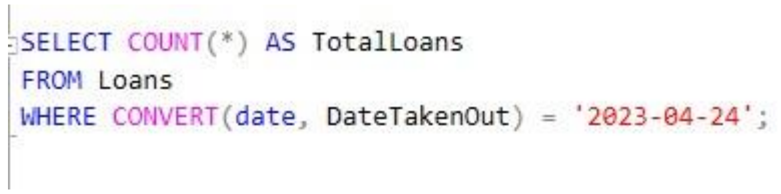
I created this trigger called "UpdateStatustoavailable" for the "Loans" table. It fires after an insert or update operation is performed on the "Loans" table. The trigger checks if the "DateReturned" column was updated, and if so, updates the "CurrentStatus" column in the "CatalogueItems" table to "Available" for the corresponding item ID in the "Loans" table.

The trigger starts with a TRY-CATCH block to handle any errors that may occur during the transaction. If the "DateReturned" column was updated, the trigger executes an update statement to set the "CurrentStatus" column to "Available" for the appropriate items in the "CatalogueItems" table.

Finally, the transaction is committed if there are no errors, or rolled back if an error occurs. Any error messages are printed to the console using the ERROR_MESSAGE() function.

3.4

Query to provide a function, view, or SELECT query which allows the library to identify the total number of loans made on a specified date.

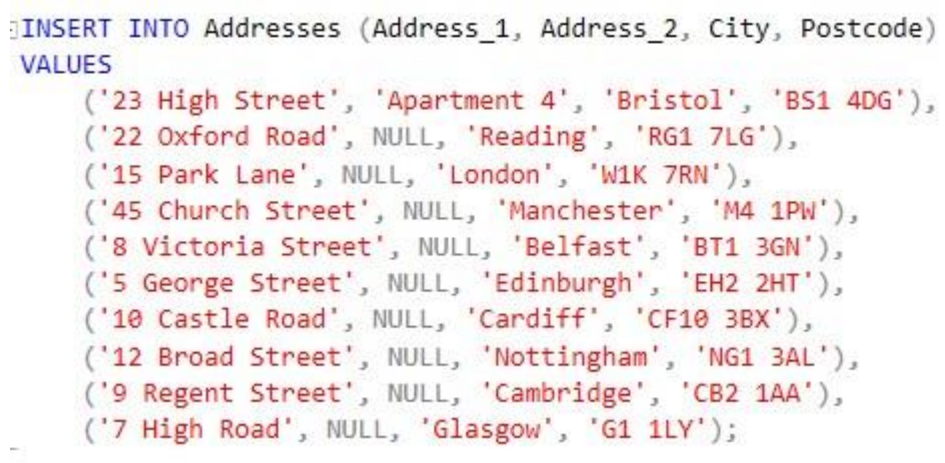


```
SELECT COUNT(*) AS TotalLoans
FROM Loans
WHERE CONVERT(date, DateTakenOut) = '2023-04-24';
```

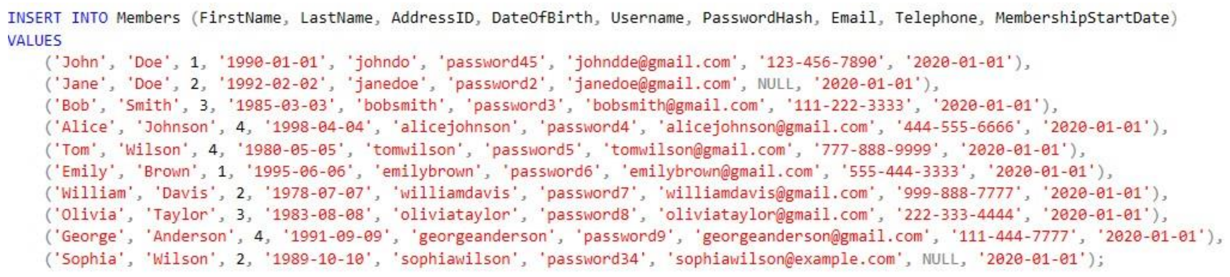
This query returns the total number of loans made on April 24, 2023 (assuming that is the date being specified). I used the CONVERT function to extract only the date portion of the DateTakenOut column and compare it to the specified date. Then I used the count function to specify the number of matching rows. The alias Total Loans is given to the resulting count.

3.5 QUESTION 6

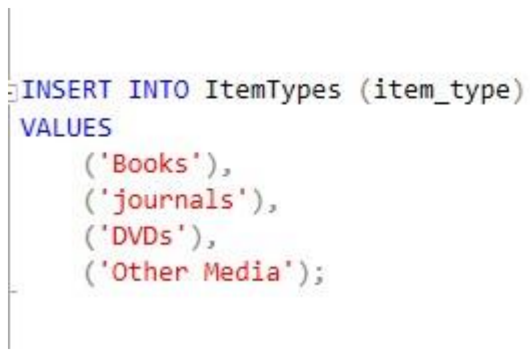
I populated all the tables with sample data using the INSERT INTO statement as seen below



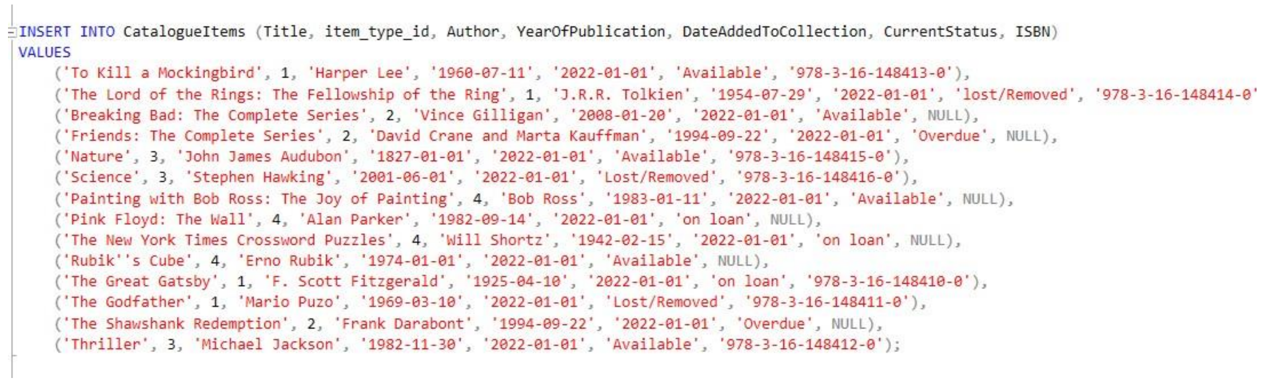
```
INSERT INTO Addresses (Address_1, Address_2, City, Postcode)
VALUES
    ('23 High Street', 'Apartment 4', 'Bristol', 'BS1 4DG'),
    ('22 Oxford Road', NULL, 'Reading', 'RG1 7LG'),
    ('15 Park Lane', NULL, 'London', 'W1K 7RN'),
    ('45 Church Street', NULL, 'Manchester', 'M4 1PW'),
    ('8 Victoria Street', NULL, 'Belfast', 'BT1 3GN'),
    ('5 George Street', NULL, 'Edinburgh', 'EH2 2HT'),
    ('10 Castle Road', NULL, 'Cardiff', 'CF10 3BX'),
    ('12 Broad Street', NULL, 'Nottingham', 'NG1 3AL'),
    ('9 Regent Street', NULL, 'Cambridge', 'CB2 1AA'),
    ('7 High Road', NULL, 'Glasgow', 'G1 1LY');
```



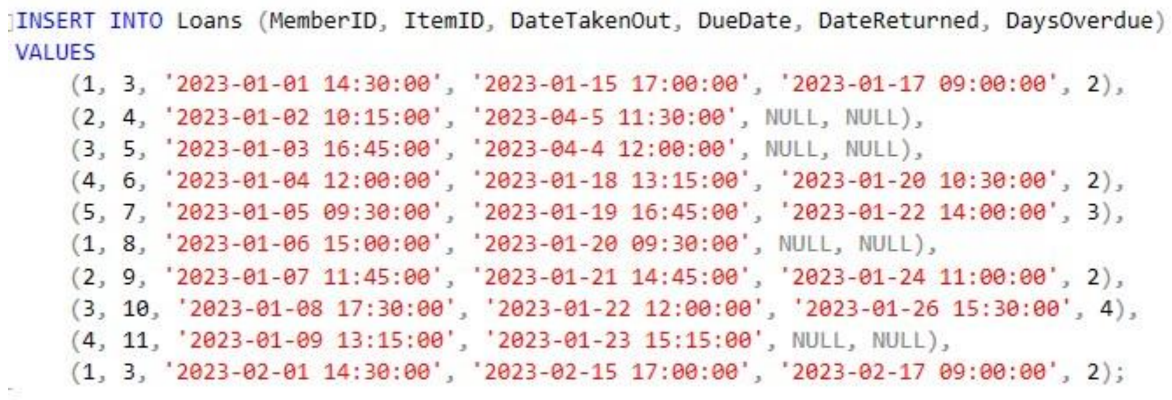
```sql
INSERT INTO Members (FirstName, LastName, AddressID, DateOfBirth, Username, PasswordHash, Email, Telephone, MembershipStartDate)
VALUES
    ('John', 'Doe', 1, '1990-01-01', 'johndo', 'password45', 'johndde@gmail.com', '123-456-7890', '2020-01-01'),
    ('Jane', 'Doe', 2, '1992-02-02', 'janedoe', 'password2', 'janedoe@gmail.com', NULL, '2020-01-01'),
    ('Bob', 'Smith', 3, '1985-03-03', 'bobsmith', 'password3', 'bobsmith@gmail.com', '111-222-3333', '2020-01-01'),
    ('Alice', 'Johnson', 4, '1998-04-04', 'alicejohnson', 'password4', 'alicejohnson@gmail.com', '444-555-6666', '2020-01-01'),
    ('Tom', 'Wilson', 4, '1980-05-05', 'tomwilson', 'password5', 'tomwilson@gmail.com', '777-888-9999', '2020-01-01'),
    ('Emily', 'Brown', 1, '1995-06-06', 'emilybrown', 'password6', 'emilybrown@gmail.com', '555-444-3333', '2020-01-01'),
    ('William', 'Davis', 2, '1978-07-07', 'williamdavis', 'password7', 'williamdavis@gmail.com', '999-888-7777', '2020-01-01'),
    ('Olivia', 'Taylor', 3, '1983-08-08', 'oliviataylor', 'password8', 'oliviataylor@gmail.com', '222-333-4444', '2020-01-01'),
    ('George', 'Anderson', 4, '1991-09-09', 'georgeanderson', 'password9', 'georgeanderson@gmail.com', '111-444-7777', '2020-01-01'),
    ('Sophia', 'Wilson', 2, '1989-10-10', 'sophiawilson', 'password34', 'sophiawilson@example.com', NULL, '2020-01-01');
```



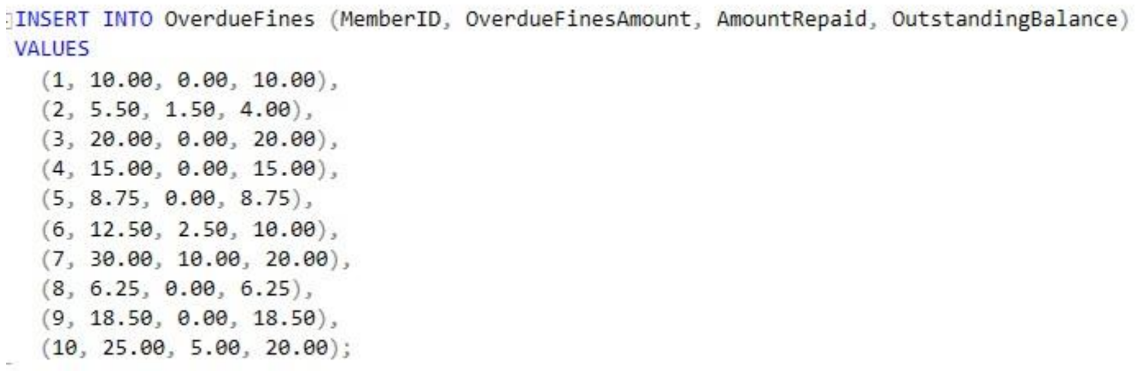
```sql
INSERT INTO ItemTypes (item_type)
VALUES
    ('Books'),
    ('journals'),
    ('DVDs'),
    ('Other Media');
```



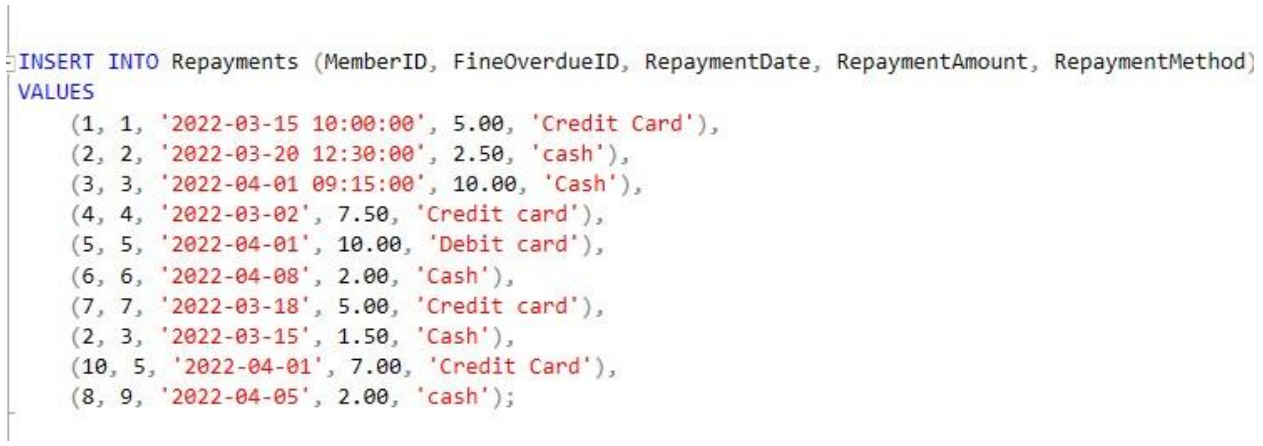
```sql
INSERT INTO CatalogueItems (Title, item_type_id, Author, YearOfPublication, DateAddedToCollection, CurrentStatus, ISBN)
VALUES
    ('To Kill a Mockingbird', 1, 'Harper Lee', '1960-07-11', '2022-01-01', 'Available', '978-3-16-148413-0'),
    ('The Lord of the Rings: The Fellowship of the Ring', 1, 'J.R.R. Tolkien', '1954-07-29', '2022-01-01', 'lost/Removed', '978-3-16-148414-0'
    ('Breaking Bad: The Complete Series', 2, 'Vince Gilligan', '2008-01-20', '2022-01-01', 'Available', NULL),
    ('Friends: The Complete Series', 2, 'David Crane and Marta Kauffman', '1994-09-22', '2022-01-01', 'Overdue', NULL),
    ('Nature', 3, 'John James Audubon', '1827-01-01', '2022-01-01', 'Available', '978-3-16-148415-0'),
    ('Science', 3, 'Stephen Hawking', '2001-06-01', '2022-01-01', 'Lost/Removed', '978-3-16-148416-0'),
    ('Painting with Bob Ross: The Joy of Painting', 4, 'Bob Ross', '1983-01-11', '2022-01-01', 'Available', NULL),
    ('Pink Floyd: The Wall', 4, 'Alan Parker', '1982-09-14', '2022-01-01', 'on loan', NULL),
    ('The New York Times Crossword Puzzles', 4, 'Will Shortz', '1942-02-15', '2022-01-01', 'on loan', NULL),
    ('Rubik''s Cube', 4, 'Erno Rubik', '1974-01-01', '2022-01-01', 'Available', NULL),
    ('The Great Gatsby', 1, 'F. Scott Fitzgerald', '1925-04-10', '2022-01-01', 'on loan', '978-3-16-148410-0'),
    ('The Godfather', 1, 'Mario Puzo', '1969-03-10', '2022-01-01', 'Lost/Removed', '978-3-16-148411-0'),
    ('The Shawshank Redemption', 2, 'Frank Darabont', '1994-09-22', '2022-01-01', 'Overdue', NULL),
    ('Thriller', 3, 'Michael Jackson', '1982-11-30', '2022-01-01', 'Available', '978-3-16-148412-0');
```



```sql
INSERT INTO Loans (MemberID, ItemID, DateTakenOut, DueDate, DateReturned, DaysOverdue)
VALUES
    (1, 3, '2023-01-01 14:30:00', '2023-01-15 17:00:00', '2023-01-17 09:00:00', 2),
    (2, 4, '2023-01-02 10:15:00', '2023-04-5 11:30:00', NULL, NULL),
    (3, 5, '2023-01-03 16:45:00', '2023-04-4 12:00:00', NULL, NULL),
    (4, 6, '2023-01-04 12:00:00', '2023-01-18 13:15:00', '2023-01-20 10:30:00', 2),
    (5, 7, '2023-01-05 09:30:00', '2023-01-19 16:45:00', '2023-01-22 14:00:00', 3),
    (1, 8, '2023-01-06 15:00:00', '2023-01-20 09:30:00', NULL, NULL),
    (2, 9, '2023-01-07 11:45:00', '2023-01-21 14:45:00', '2023-01-24 11:00:00', 2),
    (3, 10, '2023-01-08 17:30:00', '2023-01-22 12:00:00', '2023-01-26 15:30:00', 4),
    (4, 11, '2023-01-09 13:15:00', '2023-01-23 15:15:00', NULL, NULL),
    (1, 3, '2023-02-01 14:30:00', '2023-02-15 17:00:00', '2023-02-17 09:00:00', 2);
```



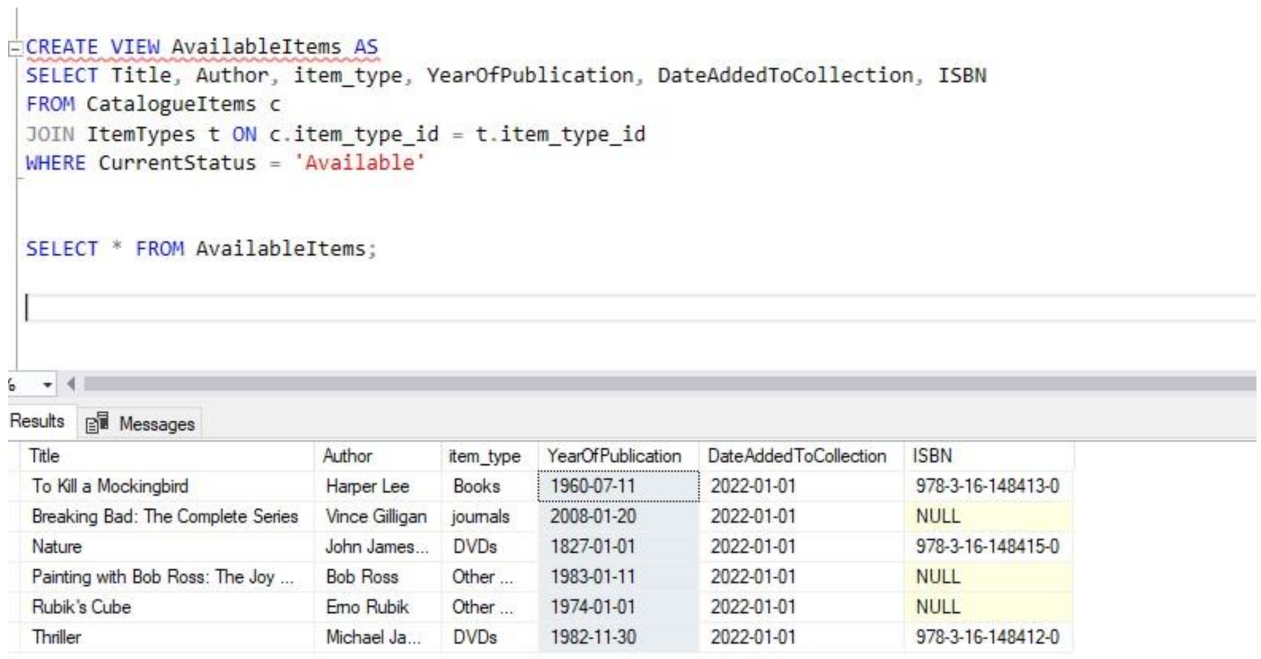
```
INSERT INTO OverdueFines (MemberID, OverdueFinesAmount, AmountRepaid, OutstandingBalance)
VALUES
  (1, 10.00, 0.00, 10.00),
  (2, 5.50, 1.50, 4.00),
  (3, 20.00, 0.00, 20.00),
  (4, 15.00, 0.00, 15.00),
  (5, 8.75, 0.00, 8.75),
  (6, 12.50, 2.50, 10.00),
  (7, 30.00, 10.00, 20.00),
  (8, 6.25, 0.00, 6.25),
  (9, 18.50, 0.00, 18.50),
  (10, 25.00, 5.00, 20.00);
```



```
INSERT INTO Repayments (MemberID, FineOverdueID, RepaymentDate, RepaymentAmount, RepaymentMethod)
VALUES
    (1, 1, '2022-03-15 10:00:00', 5.00, 'Credit Card'),
    (2, 2, '2022-03-20 12:30:00', 2.50, 'cash'),
    (3, 3, '2022-04-01 09:15:00', 10.00, 'Cash'),
    (4, 4, '2022-03-02', 7.50, 'Credit card'),
    (5, 5, '2022-04-01', 10.00, 'Debit card'),
    (6, 6, '2022-04-08', 2.00, 'Cash'),
    (7, 7, '2022-03-18', 5.00, 'Credit card'),
    (2, 3, '2022-03-15', 1.50, 'Cash'),
    (10, 5, '2022-04-01', 7.00, 'Credit Card'),
    (8, 9, '2022-04-05', 2.00, 'cash');
```

3.5 QUESTION 7i

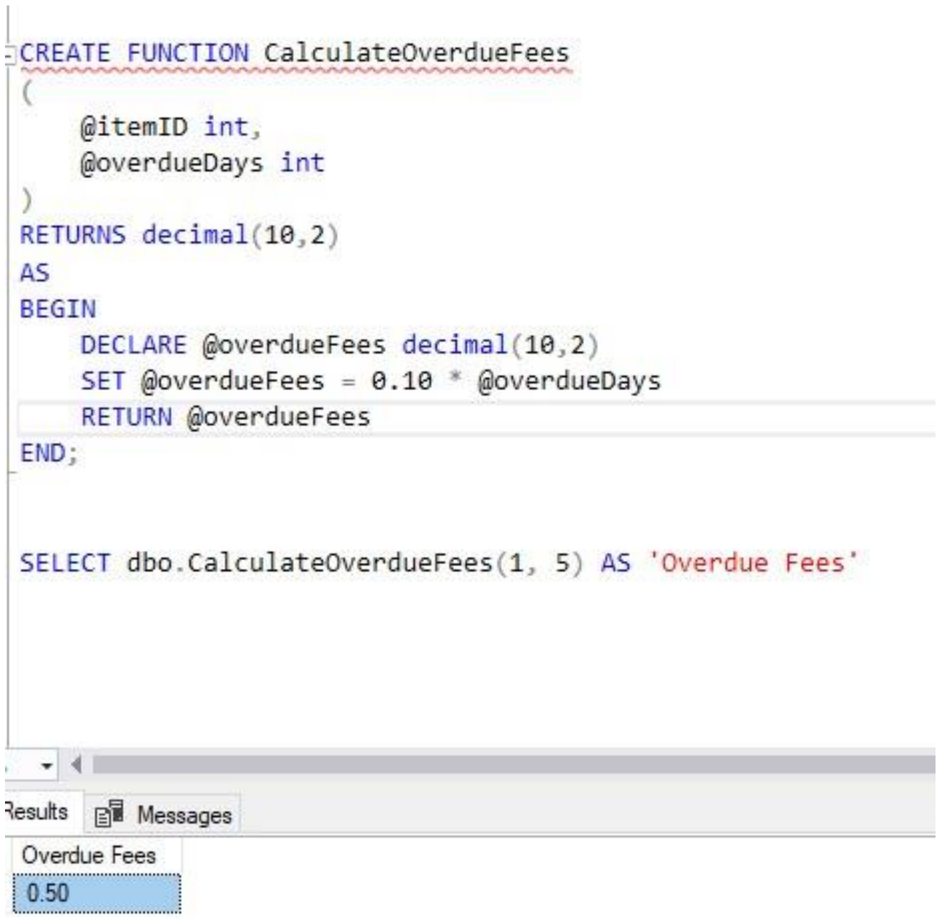
I created this view to show all available items in the catalogue



```sql
CREATE VIEW AvailableItems AS
SELECT Title, Author, item_type, YearOfPublication, DateAddedToCollection, ISBN
FROM CatalogueItems c
JOIN ItemTypes t ON c.item_type_id = t.item_type_id
WHERE CurrentStatus = 'Available'


SELECT * FROM AvailableItems;
```

Title	Author	item_type	YearOfPublication	DateAddedToCollection	ISBN
To Kill a Mockingbird	Harper Lee	Books	1960-07-11	2022-01-01	978-3-16-148413-0
Breaking Bad: The Complete Series	Vince Gilligan	journals	2008-01-20	2022-01-01	NULL
Nature	John James...	DVDs	1827-01-01	2022-01-01	978-3-16-148415-0
Painting with Bob Ross: The Joy ...	Bob Ross	Other ...	1983-01-11	2022-01-01	NULL
Rubik's Cube	Emo Rubik	Other ...	1974-01-01	2022-01-01	NULL
Thriller	Michael Ja...	DVDs	1982-11-30	2022-01-01	978-3-16-148412-0

I created a Function to calculate overdue fees:



```
CREATE FUNCTION CalculateOverdueFees
(
    @itemID int,
    @overdueDays int
)
RETURNS decimal(10,2)
AS
BEGIN
    DECLARE @overdueFees decimal(10,2)
    SET @overdueFees = 0.10 * @overdueDays
    RETURN @overdueFees
END;


SELECT dbo.CalculateOverdueFees(1, 5) AS 'Overdue Fees'
```

```
Overdue Fees
0.50
```

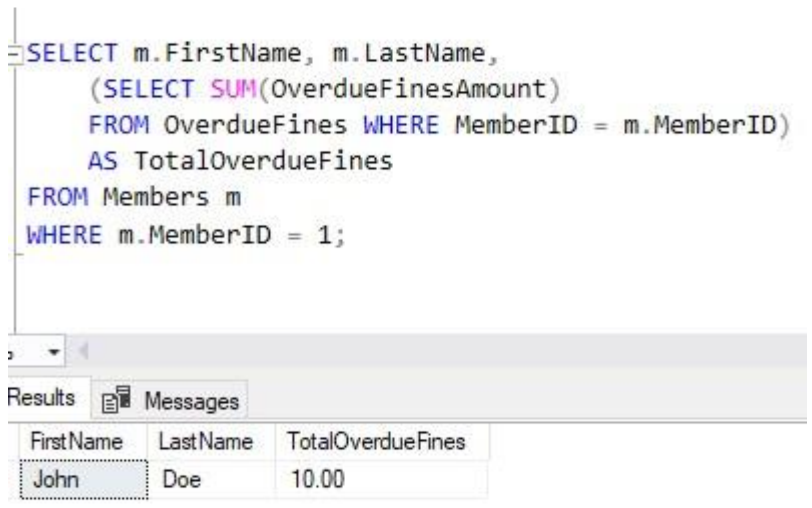
This user-defined function called "CalculateOverdueFees" takes two input parameters, "itemID" and "overdueDays", and returns the calculated overdue fees for the item that is overdue. The overdue fees are calculated based on a fixed rate of 10p per day

After creating the function, the query then executes the function for a specific item (itemID = 1) that is overdue for 5 days. The returned result is the calculated overdue fees for this item, which is displayed as "Overdue Fees".

QUESTION 7III

sub query that selects the first name and last name of a member with

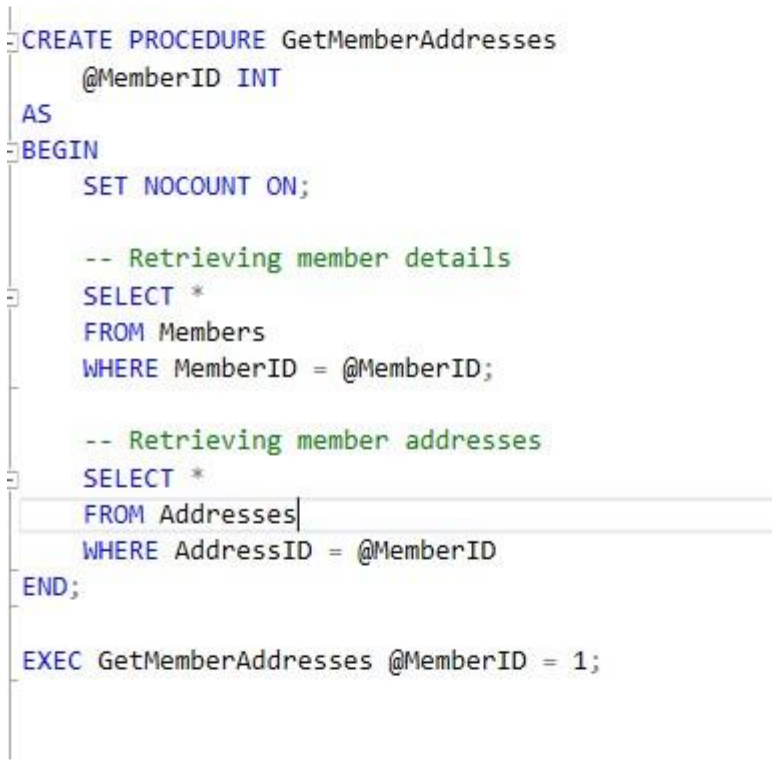
MemberID = 1 and calculates their total overdue fines.



```
SELECT m.FirstName, m.LastName,
    (SELECT SUM(OverdueFinesAmount)
    FROM OverdueFines WHERE MemberID = m.MemberID)
    AS TotalOverdueFines
FROM Members m
WHERE m.MemberID = 1;
```

Results | Messages

FirstName	LastName	TotalOverdueFines
John	Doe	10.00



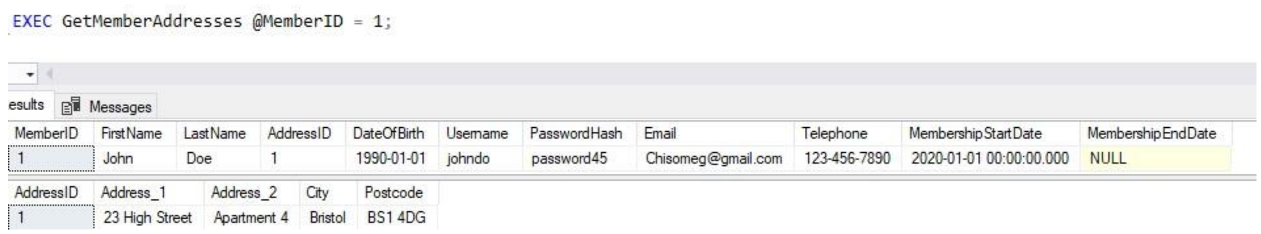
```
CREATE PROCEDURE GetMemberAddresses
    @MemberID INT
AS
BEGIN
    SET NOCOUNT ON;

    -- Retrieving member details
    SELECT *
    FROM Members
    WHERE MemberID = @MemberID;

    -- Retrieving member addresses
    SELECT *
    FROM Addresses
    WHERE AddressID = @MemberID
END;

EXEC GetMemberAddresses @MemberID = 1;
```

I used this stored procedure to retrieve the details of a member and their addresses based on a given MemberID.

The stored procedure takes a single input parameter, which is the MemberID of the member to retrieve. It then retrieves the member details from the Members table using a SELECT statement that filters by the input MemberID.

The procedure also retrieves the member addresses from the Addresses table using a SELECT statement that filters by the input MemberID.

Finally, the procedure returns the results of both SELECT statements as a single dataset.



```
EXEC GetMemberAddresses @MemberID = 1;
```

esults | Messages

MemberID	FirstName	LastName	AddressID	DateOfBirth	Usemame	PasswordHash	Email	Telephone	MembershipStartDate	MembershipEndDate
1	John	Doe	1	1990-01-01	johndo	password45	Chisomeg@gmail.com	123-456-7890	2020-01-01 00:00:00.000	NULL

AddressID	Address_1	Address_2	City	Postcode
1	23 High Street	Apartment 4	Bristol	BS1 4DG

To execute the stored procedure, I used the EXEC command followed by the procedure name and the input parameter @MemberID set to the desired value, in this case, 1.

4.0 ADVICE AND GUIDANCE

4.1 Data Integrity and Concurrency

To ensure data integrity and concurrency for my client’s library database, I would recommend the following:

Use transactions: Transactions are used to ensure that multiple database operations happen atomically. For example, when a book is loaned out, there is a need to ensure that the book's status is updated to "On Loan" and the loan details are recorded in the Loans table. By using a transaction, we can ensure that these two operations happen atomically, and if one fails, the other is rolled back.

Use constraints: I will recommend that my client use Constraints such as NOT NULL, UNIQUE, CHECK, PRIMARY and FOREIGN KEY to ensure data integrity. For example, the AddressID in the Members table is a

FOREIGN KEY that references the AddressID in the Addresses table.

This ensures that every member has a valid address associated with them.

Use appropriate data types: I will recommend that my client choose appropriate data types for each column to ensure data integrity. For example, the Telephone column in the Members table is defined as nvarchar(20) to allow for international phone numbers.

Use appropriate indexing: Indexes can be used to ensure data integrity and improve performance. For example, I will recommend an addition of an index to the Username column in the Members table to ensure that each username is unique, which will improve the speed of searches and prevent duplicate usernames.

4.2 Database Security

Database security: is a critical aspect of any system that stores sensitive and valuable data. To ensure the security of a database, I would recommend the following to my clients.

Use strong passwords: Passwords should be strong and not easily guessable. The PasswordHash column in the Members table should be stored using a strong hashing algorithm such as bcrypt.

Grant Minimal Privileges: Suppose my client has a member who needs to view the details of loans but doesn't need to modify them. In this case, my client could grant them SELECT permission on the Loans table.

4.3 Database Backup and Recovery

Regular backups are crucial to avoid data loss and enable data retrieval in case of system failure, mistakes, or cyber attacks. The Library_DesignDB tables hold important data, such as member information, loans, fines, and repayments, and losing or corrupting any of these tables could have severe consequences for the library.

To ensure data recovery, my recommendation is to perform full backups, differential backups, and transaction log backups. Full backups capture the entire database, while differential backups only capture changed data since the last full backup. Transaction log backups record all transactions since the last log backup for point-in-time recovery.

For optimal data recovery, I suggest performing full backups weekly, differential backups daily, and transaction log backups hourly. It is essential to store backups securely in an offsite location to prevent data loss due to physical damage to the servers or data center. Regular testing of backups will ensure their reliability for data recovery.

5.0 CONCLUSION

This is a database solution for a library management system with seven tables: Addresses, Members, ItemTypes, CatalogueItems, Loans, OverdueFines, and Repayments. Members' personal information is in the Members table with addresses in the Addresses table. ItemTypes and CatalogueItems tables hold information about the library's items. Loans, OverdueFines, and Repayments tables track borrowed items, overdue fines, and repayments made.

The database offers critical functions for library management, including tracking of library items, managing membership accounts, overdue fines, and loans while ensuring data integrity through foreign keys and check constraints. As a data scientist, I would suggest utilizing data analysis techniques to recognize patterns in borrowing behavior and identify popular items. Employing data visualization tools to create interactive reports could help library staff make data-driven decisions to enhance library services.