Table of Contents

Task 1 Report

Introduction

This task involves analyzing clinical trial datasets and combining the results with a list of pharmaceutical companies. The goal is to extract pertinent information from clinical trial datasets, such as the sponsor, status, start and end dates, study type, and conditions and interventions investigated. The clinical trial data will be correlated with a list of pharmaceutical companies to uncover any relationships. The final submission will include results from the data release in 2021. This task requires proficiency in data manipulation, merging, and analysis, and the capacity to manage extensive datasets. This task requires the usage of Databricks and Tableau for completion.

Part 3

1.0 Identifying Entities

The first step to designing the database is to identify the various entities in the database. The following entities have been identified as necessary for the database system:

- Members: The Members entity stores personal information about library members. The Members entity will also track the date the membership ended.

- Fines: The Fines entity is responsible for keeping track of the fines owed by members.

- Repayments: The Repayments entity records information about each repayment made by a member for a fine.

- Catalogue: The Catalogue entity stores information about the library's resources.

- Loans: The Loans entity stores information about the borrowing history of members. If an item is overdue, an overdue fee needs to be calculated at a rate of 10p per day.

1.1 Normalization

The next process after the identification of the entities is to normalize the database. Normalization is, in relational database design, the process of organizing data to minimize redundancy [1]. (“Concept of Normalization - theintactone”) (“Concept of Normalization - theintactone”) It involves dividing larger tables into smaller, more manageable ones, with the aim of improving data integrity and making data maintenance more efficient.

The client’s requirement is to normalize the database into the 3rd Normal form:

- 1st Normal Form: The initial design of the database included the following entities: Members, Fines, Repayments, Catalogue, and Loans. After examining the attributes of each entity, I determined that the database was already in 1NF as each entity contained atomic values and there were no repeating groups.

    - The Members
    - The Fines
    - The Repayments
    - The Catalogue
    - The Loans

    These attributes were already atomic, and no further normalization was necessary.

    In conclusion, the database is already in 1NF as each entity contains atomic values and no repeating groups. This means that the database meets the requirements of the first normal form and ensures that data is stored consistently and efficiently, making it easier to maintain and query.

- 2nd Normal Form: The initial design of the database was already in first normal form as each entity contained atomic values and there were no repeating groups. However, some of the entities were not fully normalized to the second normal form as they contained attributes that were partially dependent on the primary key.

    To address this, the entities were split further.

This new design ensures that each entity is fully dependent on the primary key and that there are no partial dependencies. Each entity now represents a single concept and there are no redundant attributes.

In summary, the updated design in second normal form consists of the following entities:

- Members
- Fines
- Repayments
- Items
- Item Copies
- Loans

By normalizing the database to 2NF, I have improved its structure, eliminated redundancy, and ensured that data is stored efficiently and accurately. This will make it easier to maintain and query the database in the future.

- 3rd Normal Form: Finally, I normalized the database to 3NF to eliminate transitive dependencies. This involved creating new entities and further splitting some of the existing entities to ensure that each entity contained only non-transitive attributes.

  The resulting database design in 3NF consists of the following entities: Member, Member Login, Member Address, Member Category, Item, Item Copy, Loan, and Fine Repayment. Each entity has been designed to store only attributes that are directly related to the primary key and avoid any data redundancy.

  By normalizing the database to 3NF, I have ensured that the database is more efficient, scalable, and maintainable. It will reduce the potential for data anomalies, such as update anomalies, insertion anomalies, and deletion anomalies, which could occur when data is not properly organized. Overall, the 3NF design will help to improve the accuracy and reliability of the library database system.

  These are the tables created from the normalization in the 3rd Normal form and their attributes:

  - Addresses store the addresses of library members
  - Member Addresses, this table exists because I considered the library having members who will have multiple addresses(home, office, billing, shipping, etc.), this table will serve as a link between the member and the address table.
  - Member table stores information about each library member. Which also helps with member sign-up.
  - Member_Login table enables the client to keep track of the login audit of each member.
  - Member_Category table is a lookup table for the different categories of library membership.
  - Item_Type_Lookup table is a lookup table for the different types of items in the library's collection.

- Payment_Method_Lookup table is a lookup table for the different payment methods accepted by the library for repayment of overdue fines.
- Current_Status_Lookup table is a lookup table for the different statuses of an item copy in the library's collection.
- Item table stores information about each item in the library's collection.
- ItemCopy table stores information about each copy of an item in the library's collection. It will have an itemID foreign column which will reference the item table which contains more information about the item.
- The loan table tracks when a member borrows a copy of an item from the library.
- The fine_Repayment table tracks the repayment of loan fines by members. The reason why this table is key is that it not only enables members to repay their loan fine at once but allows them to repay it more than once.

1.2 Entity Relationship Diagram

The various entities and their attributes will be represented in the Entity relationship diagram and the relationships between the entities will be discussed as well.

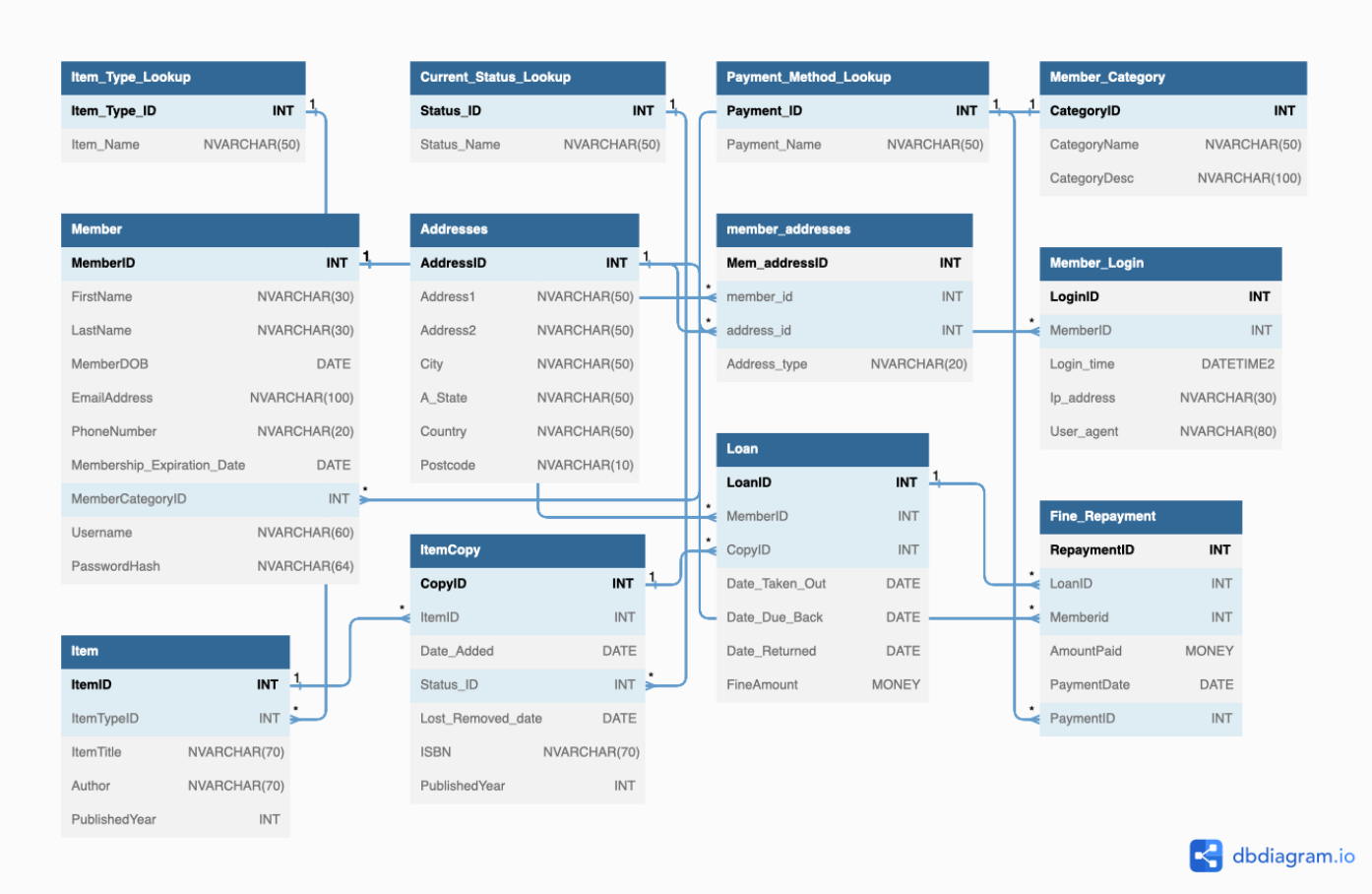

Figure 1.0 - Entity Relationship Diagram. Image generated from www.dbdiagram.io

The primary keys for each table are highlighted in bold black while the foreign keys have a light blue background. Based on the entity diagram and the normalization, the relationship between the tables is as follows:

- Member_Category to Member: one-to-many (one member category can have multiple members, but each member can only belong to one category).
- Member to Addresses: many-to-many (a member can have multiple addresses, and an address can belong to multiple members)
- Member to Member_Login: one-to-many (one member can have multiple login records, but each login record can only belong to one member)
- Item_Type_Lookup to Item: one-to-many (one item type can have multiple items, but each item can only belong to one type)
- Item to ItemCopy: one-to-many (one item can have multiple copies, but each copy can only belong to one item)
- Current_Status_Lookup to ItemCopy: many-to-one (multiple item copies can have the same status, but each copy can only have one status)
- The loan and fine_repayment tables have a one-to-many relationship. This is because a loan can have multiple fine repayments associated with it, but each fine repayment is associated with only one loan.

1.3 Database Schema in T-SQL

1.3.1 Creating the Database

Given the client didn't give a specific name for the database in the requirement, I will call the database "**ClientLibraryDB**". Here is the T-SQL script to create the database.

```
1  /* The following code creates a new database called ClientLibraryDB */
2
3  CREATE DATABASE ClientLibraryDB;
```

Figure 1.1 – database creation T-SQL script. Image generated from www.snappify.com

1.3.2 Creating tables.

1.3.2.1 Creating the Look-up Tables

I will be creating some lookup tables which enable other tables to function properly, by using a lookup table, the system can avoid duplicating item type names in multiple tables and ensure data consistency. Here are the tables and the codes:

- **Item Type Lookup table:** This table stores the name of different item types, currently the client has four item types(Book, Journal, DVD or Other Media), the good thing about this table is that if the member decides to create another item type, it can be added to this table and used by the system.
- **Current Status Lookup table:** This table will store the current status(On Loan, Overdue, Available or Lost/Removed) in the client's requirement and future status that might be created by the client.
- **Payment Type Lookup table:** This table will store the payment type (cash, card) in the client's requirement and a future payment type that might be created by the client.
- **Member Category table:** This table will store the various categories a member could belong to.

```
/* lookup table for the item type */
CREATE TABLE Item_Type_Lookup
(
-- auto-incrementing primary key for each item type
Item_Type_ID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
Item_Name NVARCHAR(50) NOT NULL -- name of the item type, maximum length of 50 characters
);

/* lookup table for the Status Name */
CREATE TABLE Current_Status_Lookup
(
-- Unique ID for each status which will be Autogenerated
Status_ID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
-- Status name like On Loan, Overdue, Available or Lost/Removed
Status_Name NVARCHAR(50) NOT NULL
);

/* lookup table for the payment method */
CREATE TABLE Payment_Method_Lookup
(
-- Unique ID for each payment which will be Autogenerated
Payment_ID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
-- Payment name like cash or card or any future name
Payment_Name NVARCHAR(50) NOT NULL
);

/*  Creating the Member Category Lookup table */
CREATE TABLE Member_Category
(
CategoryID INT IDENTITY(1,1) NOT NULL PRIMARY KEY, -- auto-incrementing primary key for each category
CategoryName NVARCHAR(50) NOT NULL, -- Category Name
CategoryDesc NVARCHAR(100) NULL -- Category Description
);

```

Figure 1.2 – Lookup tables T-SQL script. Image generated from www.snappify.com

1.3.2.2 Creating the Member tables

The T-SQL script below is for creating the member tables and other tables associated with a member, the tables in the script are below:

- **Member table:** This is the table that will hold information about members and all the client specifications were considered when designing this table up to the sign-up and login specifications.
- **Addresses table:** This table will contain the addresses belonging to different members. This table can store multiple addresses belonging to a single member.
- **Member Address table:** This table is created to link members to their addresses. This table allows for a many-to-many relationship between members and addresses, as a member can have multiple addresses and an address can be associated with multiple members.
- **Member Login table:** This table is crucial for audit purposes. It keeps track of every time a member logs in.

```sql
/* Creating the Member table */
CREATE TABLE Member
(
MemberID INT IDENTITY(1,1) NOT NULL PRIMARY KEY, -- auto-incrementing primary key for each member
FirstName NVARCHAR(30) NOT NULL, -- first name of member, maximum length of 30 characters
LastName NVARCHAR(30) NOT NULL, -- last name of member, maximum length of 30 characters
MemberDOB DATE NOT NULL, -- date of birth of member
EmailAddress NVARCHAR(100) UNIQUE, -- email address of member, unique constraint
PhoneNumber NVARCHAR(20), -- phone number of member, maximum length of 20 characters
Membership_Expiration_Date DATE, -- date when member's membership expires
MemberCategoryID INT, -- foreign key reference to Member_Category table(optional)
Username NVARCHAR(60) UNIQUE NOT NULL, -- username of member, maximum length of 60 characters
PasswordHash NVARCHAR(64), -- this column will contain hashed characters, maximum length of 64 characters
 -- foreign key constraint for MemberCategoryID column
CONSTRAINT ck_fk_mem_memcat FOREIGN KEY (MemberCategoryID) REFERENCES Member_Category (CategoryID)
);

/* Creating the Addresses which will hold different address */
CREATE TABLE Addresses
(
AddressID INT IDENTITY(1,1)  NOT NULL PRIMARY KEY, -- Unique ID for each address
Address1 NVARCHAR(50) NOT NULL, -- Address line 1 of the member
Address2 NVARCHAR(50) NULL, -- Address line 2 of the member
City NVARCHAR(50) NULL, -- City of the member
A_State NVARCHAR(50) NULL, -- state of the member
Country NVARCHAR(50) NULL, -- Country of the member
Postcode NVARCHAR(10) NOT NULL -- Postcode of the member
);


/* Creating the member_addresses table to link members to their addresses */
CREATE TABLE member_addresses
(
Mem_addressID INT IDENTITY(1,1)  NOT NULL PRIMARY KEY,
member_id INT, -- foreign key field that references the ID field in the members table
address_id INT, -- foreign key field that references the ID field in the addresses table
Address_type NVARCHAR(20) NOT NULL, -- The type of address(Home,office,Billing,Shipping)
-- create the foreign key relationship to the members table
FOREIGN KEY (member_id) REFERENCES Member(MemberID),
-- create the foreign key relationship to the addresses table
FOREIGN KEY (address_id) REFERENCES Addresses(AddressID)
);


/*  Code to create the Member Login table */
CREATE TABLE Member_Login
(
LoginID INT IDENTITY(1,1) NOT NULL PRIMARY KEY, -- auto-incrementing primary key for each login record
MemberID INT NOT NULL, -- foreign key reference to the Member table
-- date and time of login, default value is current date and time
Ip_address NVARCHAR(30) NOT NULL, -- IP address of device used for login
Login_time DATETIME2 NOT NULL DEFAULT (GETDATE()),
User_agent NVARCHAR(80) NOT NULL, -- user agent string of device used for login
-- foreign key constraint for MemberID column
CONSTRAINT ck_fk_memberLog_mid FOREIGN KEY (MemberID) REFERENCES Member (MemberID)
);

```

Figure 1.3 – Member tables T-SQL script. Image generated from www.snappify.com

1.3.2.3 Creating the Item table

```
/* Creating the Item Table */
CREATE TABLE Item
(
ItemID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,  -- Autogenerated ItemID
ItemTypeID INT NOT NULL,  -- Foreign key to the ItemTypeLookup table to indicate the type of item
ItemTitle NVARCHAR(70) NOT NULL,  -- Title of the item
Author NVARCHAR(70) NOT NULL,  -- Author of the item
PublishedYear INT NOT NULL,  -- Original published date of the item
-- Check constraint to ensure that ItemTypeID is a valid ItemType_ID in ItemTypeLookup table
CONSTRAINT ckfkitemitemtype FOREIGN KEY (ItemTypeID) REFERENCES Item_Type_Lookup(Item_Type_ID)
);
```

Figure 1.4 – Item table creation T-SQL script. Image generated from www.snappify.com

1.3.2.4 Creating the Item Copy Table

```
/* Creating the ItemCopy Table */
CREATE TABLE ItemCopy
(
-- Unique ID for each item copy which will be Autogenerated
CopyID INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
ItemID INT NOT NULL,  -- ID of the item that this copy belongs to
Date_Added DATE NOT NULL,   -- Date on which this copy was added to the library
Status_ID INT Default 1, -- ID of the current status of the item copy. Default Status(available)
Lost_Removed_date DATE, -- Date on which this copy was lost or removed from the library
ISBN NVARCHAR(70), -- ISBN number when applicable
PublishedYear INT,  -- Year the item was published(if different)
-- Foreign key to the Current_Status_Lookup table for data integrity
CONSTRAINT ck_item_status FOREIGN KEY (Status_ID)  REFERENCES Current_Status_Lookup(Status_ID),
-- Foreign key to the Item table to ensure that only valid item IDs are inserted into this column
CONSTRAINT ck_fk_itemcopy_itemID  FOREIGN KEY (ItemID) REFERENCES Item(ItemID)
);
```

Figure 1.5 – Item copy table creation T-SQL script. Image generated from www.snappify.com

1.3.2.5 Creating the Loan table.

```
/* Creating the Loan Table */
CREATE TABLE Loan
(
LoanID INT IDENTITY(1,1) NOT NULL PRIMARY KEY, -- Unique ID for each Loan which will be Autogenerated
MemberID INT NOT NULL, -- Identifier for the member who borrowed the item
CopyID INT NOT NULL, -- Identifier for the item copy that was borrowed
Date_Taken_Out DATE NOT NULL, -- Date on which the item was borrowed
Date_Due_Back DATE NOT NULL, -- Due date for returning the item
Date_Returned DATE, -- Date on which the item was returned (nullable)
FineAmount MONEY, -- Amount of fine charged for late return (nullable)
-- Foreign key constraint to ensure that MemberID references a valid MemberID in the Member table
CONSTRAINT ck_loan_memberID FOREIGN KEY (MemberID) REFERENCES Member (MemberID),
-- Foreign key constraint to ensure that CopyID references a valid CopyID in the ItemCopy table
CONSTRAINT ck_loan_item_copyid FOREIGN KEY (CopyID) REFERENCES ItemCopy (CopyID)
);
```

Figure 1.6 – Item Loan table creation T-SQL script. Image generated from www.snappify.com

1.3.2.6 Creating the Fine Repayment Table

```sql
/* Creating the Repayment Table  */
CREATE TABLE Fine_Repayment
(
RepaymentID INT IDENTITY(1,1) NOT NULL PRIMARY KEY, -- Unique ID for each repayment which will be Autogenerated
LoanID INT NOT NULL, -- Identifier for the loan that the fine repayment is associated with
Memberid INT NOT NULL, -- Identifier for the member who made the fine repayment
AmountPaid MONEY NOT NULL,  -- Amount of the fine that was paid
PaymentDate DATE NOT NULL, -- Date on which the fine was paid
PaymentID INT NOT NULL, -- Identifier for the payment method that was used to make the fine repayment
-- Foreign key constraint to ensure that LoanID references a valid LoanID in the Loan table
CONSTRAINT chkfineIdreplace FOREIGN KEY (LoanID)  REFERENCES Loan (LoanID),
-- Foreign key constraint to ensure that PaymentID references a valid PaymentID in the PaymentMethodLookup table
CONSTRAINT chkpaymentidreplace FOREIGN KEY (PaymentID)  REFERENCES Payment_Method_Lookup(Payment_ID),
-- Foreign key constraint to ensure that Memberid references a valid MemberID in the Member table
CONSTRAINT chkmemidreplace FOREIGN KEY (Memberid) REFERENCES Member (MemberID)
);
```

Figure 1.7 – Fine Repayment table creation T-SQL script. Image generated from www.snappify.com

1.3.3 Data Types Justification

These are some of the data types used in the database design:

Data Type	Justification	Column
INT	INT is a fixed-length data type that uses 4 bytes of storage, making it useful for data storage and retrieval. When performing numeric calculations in SQL Server queries and aggregations, it is faster than larger data types like BIGINT.	This data type was used by the Primary and Foreign key columns, given they are all autogenerated and all started from 1.
NVARCHAR	One of the best features of NVARCHAR is its ability to store multi-languages. Given the client's requirement, there's a possibility of storing multi-language data.	Columns like Item titles, Member names are likely to have multi-language characters which make NVARCHAR the perfect data type for them and other variable-length characters in this schema
DATE	The date datatype is the datatype in T-SQL which is perfect for storing date values that don't have a time component.	MemberDOB which is the member's date of birth is the first column I used this data type for.
DATETIME2	This date data type has a time component, unlike the Date datatype. I used datetime2 instead of just datetime because datetime2 provides greater precision and a more comprehensive range of values.	The login time on the Member login table used this data type because I want the library to be able to capture both the date and time a member log in.
MONEY	This data type is used for storing monetary value. Using the MONEY data type ensures that the values stored in the column will be	FineAmount and AmountPaid are two columns that made use of this data type because

	accurate to four decimal places, which is important for financial calculations	money values will be stored in this column

Part 2

2. (A) Search Catalogue with character String

One of the client requirements is to be able to search the catalogue for matching character strings by title. Results should be sorted with the most recent publication date first. Which will allow them to query the catalogue looking for a specific item.

```
 /* View that search item catalogue by matching character */

 Go

CREATE PROCEDURE SP_SearchItemCatalogueByTitle
    @searchTerm NVARCHAR(50)
AS
BEGIN
    SELECT i.ItemTitle 'Item Title',i.Author 'Item Author',
            -- I'm picking the item copy published date first
            -- incase it's different from the original one
            ISNULL(c.PublishedYear,i.PublishedYear) 'Published Year',
            l.Item_Name 'Item Name'
    FROM Itemcopy c -- I'm returning the copies of the item
    JOIN  Item i
    on c.itemID = i.itemID -- joining item to itemcopy
    JOIN Item_Type_Lookup l
    ON i.ItemTypeID = l.Item_Type_ID --joining the item lookup table for the item type name
    WHERE i.ItemTitle LIKE '%' + @searchTerm + '%' -- search for the character match
    ORDER BY  ISNULL(c.PublishedYear,i.PublishedYear) DESC; -- ordering it by the most recent published year
END;

-- testing the stored procedure
EXEC SP_SearchItemCatalogueByTitle 'O';

GO
```

Figure 2.1 – Item search by title stored procedure T-SQL script. Image generated from www.snappify.com

This stored procedure accepts one parameter, @searchTerm, which is the search term the client wishes to use to find matching items. The LIKE operator with % wildcards is used to find any items with the search term in their title. For the published date, if an item is reprinted later, the new copies may have a different published date than the original copies, so, I'm sorting it by the item copy date first before the original item copy.

Results Messages

	Item Title	Item Author	Published Year	Item Name
1	Sleeping Dog Lies	George Orwell	2002	Book
2	Sleeping Dog Lies	George Orwell	2001	Book
3	The Godfather	Mario Puzo	1969	DVD
4	The Godfather	Mario Puzo	1969	DVD
5	The Godfather	Mario Puzo	1969	DVD
6	The Godfather	Mario Puzo	1969	DVD
7	The Godfather	Mario Puzo	1969	DVD
8	To Kill a Mockingbird	Harper Lee	1960	Book
9	To Kill a Mockingbird	Harper Lee	1960	Book
10	To Kill a Mockingbird	Harper Lee	1960	Book
11	To Kill a Mockingbird	Harper Lee	1960	Book
12	To Kill a Mockingbird	Harper Lee	1960	Book

Figure 2.1.1 – Item search by title stored procedure result set.

From the result set, ‘Sleeping dog lies’ has two items published on different dates.

2. (B) Items on Loan View

The next task is to return a complete list of all items currently on loan which have a due date of fewer than five days from the current date (i.e., the system date when the query is run).

```sql
/* item on loan due in less that five days */
GO

CREATE VIEW VW_GetFullItemListDueSoon
AS
     SELECT
            CONCAT(m.FirstName,' ',m.LastName) 'Member Full Name',
            i.ItemTitle 'Item Title',i.Author 'Item Author',i.PublishedYear 'Published Year',
            t.Item_Name 'Item Name',l.Date_Taken_Out 'Loan Date', l.Date_Due_Back 'Loan Date Due',
            DATEDIFF(day,GETDATE(),Date_Due_Back) AS 'Days Left To Return(From Today)'

    FROM Loan l
    JOIN ItemCopy c
    ON l.CopyID = c.CopyID -- joining the loan and itemcopy table
    JOIN item i
    ON i.ItemID = c.ItemID -- joining the item and itemcopy table
    JOIN Item_Type_Lookup t
    ON t.Item_Type_ID = i.ItemTypeID --joining the item lookup table for the item type name
    JOIN member m
    ON m.memberID = l.MemberID -- joining the loan to the member table
    WHERE c.Status_ID   =
        -- using subquery to find the status id of "on loan" items
        (SELECT Status_ID FROM Current_Status_Lookup WHERE Status_Name = 'On Loan')
    AND DATEDIFF(day,GETDATE(),Date_Due_Back) < 5
    AND DATEDIFF(day,GETDATE(),Date_Due_Back) ⩾ 0 ; -- to avoid longer negative days

GO
-- to test the view and see the full list, the following code is used
SELECT *
FROM VW_GetFullItemListDueSoon;
```

Figure 2.3 – Items on Loan view T-SQL script. Image generated from www.snappify.com

The view VW_GetFullItemListDueSoon joins the tables Loan, ItemCopy, item, Item Type Lookup, and member to return a list of items that are currently on loan and have a due date that is less than five days from today, filtered for items that are still on loan (by the status ID) and the due date that is less than five days from the current date(system Date).

Results Messages

	Member…	Item Title	Item Author	Published Year	Item Name	Loan Date	Loan Date Due	Days Left To…
1	Dr NathnAr	To Kill a Mock…	Harper Lee	1960	Book	2023-01-10	2023-03-26	1
2	Dr NathnAr	Gideon Spies	Ben Hayta	2015	Book	2023-03-10	2023-03-28	3

Figure 2.3.1 – Items on Loan view result set.

From the result set, the first item was borrowed and due back 26th of March and this result was run on the 25th of March which is why it shows that it's due in 1 day. The same goes for the second item which expires in 3 days.

2. (C) Inserting Member Stored Procedure

```sql
/* Stored procedure to insert into member information */

GO

CREATE PROCEDURE SP_InsertMember
    -- various parameter passed to the stored procedure for insertion
    @FirstName NVARCHAR(30),@LastName NVARCHAR(30),@MemberDOB DATE,
    @EmailAddress NVARCHAR(100) = NULL, @PhoneNumber NVARCHAR(20) = NULL,
    @Membership_Expiration_Date DATE = NULL, @MemberCategoryID INT = NULL,
    @Username NVARCHAR(60), @PasswordH NVARCHAR(64), @Address1 NVARCHAR(50),
    @Address2 NVARCHAR(50) = NULL, @City NVARCHAR(50)= NULL,
    @A_State NVARCHAR(50)= NULL,  @Country NVARCHAR(50)= NULL,
    @Postcode NVARCHAR(10),  @Address_type NVARCHAR(20)
AS
BEGIN

     /* When SET NOCOUNT is turned on, SQL Server does not send the message
     indicating the number of rows affected by a Transact-SQL statement to the
     client, which can reduce network traffic and improve the performance of querie */
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION; -- start of transaction

        DECLARE @AddressID INT;
        DECLARE @MemberID INT;
        -- for Database security, I'm using Hasbytes to store the password passed to the procedure
        DECLARE @hashedPassword NVARCHAR(100) = CONVERT(NVARCHAR(100), HASHBYTES('SHA2_256', @PasswordH), 2)

        -- Insert address
        INSERT INTO Addresses (Address1, Address2, City, A_State, Country, Postcode)
        VALUES (@Address1, @Address2, @City, @A_State, @Country, @Postcode);

        -- Get the ID of the inserted address
        SET @AddressID = SCOPE_IDENTITY();

        -- Insert member
        INSERT INTO Member (FirstName, LastName, MemberDOB, EmailAddress, PhoneNumber,
                            Membership_Expiration_Date, MemberCategoryID, Username, PasswordHash)
        VALUES (@FirstName, @LastName, @MemberDOB, @EmailAddress, @PhoneNumber, @Membership_Expiration_Date,
                            @MemberCategoryID, @Username, @hashedPassword);

        -- Get the ID of the inserted member
        SET @MemberID = SCOPE_IDENTITY();

        -- Insert member's address
        INSERT INTO member_addresses (member_id, address_id, Address_type)
        VALUES (@MemberID, @AddressID, @Address_type);

        -- view the various records inserted
        -- Viewing the member's table after Insertion
        SELECT * FROM member WHERE MemberID = @MemberID; -- Member table
        SELECT * FROM addresses WHERE AddressID = @AddressID; -- address table
        SELECT * FROM member_addresses WHERE member_id = @MemberID; -- member address

        COMMIT TRANSACTION; -- commit the transaction
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION; -- rollback transaction if it fails
        THROW;
    END CATCH;

END;
```

Figure 2.4 – Inserting a new Member T-SQL script. Image generated from www.snappify.com

In the insert stored procedure, I've added a BEGIN TRY and BEGIN CATCH block. If an error occurs during the transaction, the ROLLBACK TRANSACTION statement undoes any database changes and the THROW statement throws an error message. If no errors occur, the COMMIT

TRANSACTION statement will save the database changes. In this manner, transactions can be used to restore the database to its previous state in the event of a system failure, ensuring data integrity and allowing for recovery.

Overall, this stored procedure helps to ensure that the library system can accurately and securely add new members.

```
-- Testing the stored Procedure
EXEC InsertMemberT @FirstName = 'Nathan',@LastName = 'Topping',@MemberDOB = '1987-10-10',
    @EmailAddress = 'nathan@salforduni.com',@PhoneNumber = '0382823323',  @MemberCategoryID = 1,
    @Username = 'nattop',@PasswordH = 'hashedpassword*1',@Address1 = '123 Main St',
    @Address2 = NULL,@City = 'Salford',@A_State = 'Manchester',@Country = 'UK',@Postcode = 'M50KT',
    @Address_type = 'Home';
```

Figure 2.5 – Testing inserting a new Member T-SQL script. Image generated from www.snappify.com

2. (D) Updating Member Stored Procedure

```
/* Updating a member information */
CREATE PROCEDURE SP_UpdateMember
    @memberID INT,
    @firstName VARCHAR(50),
    @lastName VARCHAR(50),
    @email VARCHAR(50),
    @phoneNumber VARCHAR(20)
AS
BEGIN
    SET NOCOUNT ON; -- good for query performance
    BEGIN TRY
        BEGIN TRANSACTION;
        UPDATE Member
        SET FirstName = @firstName,LastName = @lastName,EmailAddress = @email,
            PhoneNumber = @phoneNumber
        WHERE MemberID = @memberID;
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION; --
        THROW;
    END CATCH
END;
```

Figure 2.6 – Updating Member information T-SQL script. Image generated from www.snappify.com

3. (Loan History View)

The library wants to be able 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.

```sql
/* View showing the previous and current loan history */
GO

CREATE VIEW VW_LoanHistory
AS SELECT m.FirstName + ' ' + m.LastName AS 'Member Name',
       i.ItemTitle AS 'Item Title',
       l.Date_Taken_Out AS 'Borrowed Date',
       l.Date_Due_Back AS 'Due Date',
       l.Date_Returned AS 'Returned Date',

       CASE
         WHEN l.Date_Returned IS NOT NULL AND l.Date_Returned > l.Date_Due_Back THEN
              DATEDIFF(DAY, l.Date_Due_Back, l.Date_Returned) * 0.1 -- 10p per overdue day
         ELSE 0
       END AS 'Loan Associated Fine Amount', -- each day due attracts 10p

       CASE  WHEN l.Date_Returned IS NULL AND l.FineAmount > 0 THEN 'Current Loan'
            -- I'm using this logic to clasify a loan as previous since it has been returned
             WHEN   l.Date_Returned IS NOT NULL
                AND l.Date_Due_Back ⩽  GETDATE()
                AND l.FineAmount = 0 THEN 'Previous Loan'

             ELSE 'OTHER'
        END AS 'Loan Status'

        -- Getting the sum of all repayment for this loan on the fine_repayment table
        , sum(isnull(f.AmountPaid,0)) 'Total Fine Repaid'
FROM Loan l
JOIN ItemCopy c ON l.CopyID = c.CopyID
JOIN Item i ON i.ItemID = c.ItemID
JOIN Member m ON m.MemberID = l.MemberID
LEFT JOIN Fine_Repayment f ON l.LoanID = f.LoanID
GROUP BY
m.FirstName + ' ' + m.LastName ,
       i.ItemTitle , l.Date_Taken_Out , l.Date_Due_Back , l.Date_Returned  ;

 GO
```

Figure 2.6 – Loan History View T-SQL script. Image generated from www.snappify.com

The "LoanHistory" view contains a complete record of all loans made by members, combining data from several tables such as Loan, ItemCopy, Item, Member, and Fine Repayment. On this view, I used an inner join and a left join. The left join is used because it is possible that a payment history for a specific loan is not available. For the classification of the loan as a previous one, I'm using an assumption that once an item has been returned and the whole fine has been paid and the due date is less than the current date, then it's a previous loan. This can be adjusted based on another assumption

Results | Messages

Membe…	Item Title	Borrowed Date	Due Date	Returned Date	Loan Associated Fine Amount	Loan Status	Total Fine
Dr NathnAr	Gideon Spies	2023-03-10	2023-03-28	*NULL*	0.0	Current Loan	0.00
Dr NathnAr	The Godfather	2023-01-12	2023-03-01	2023-02-28	0.0	Previous Loan	0.00
Dr NathnAr	The Godfather	2023-03-10	2023-03-20	*NULL*	0.0	Current Loan	0.00
Dr NathnAr	The Godfather	2023-03-10	2023-04-12	*NULL*	0.0	Current Loan	0.00
Dr NathnAr	To Kill a Mo…	2023-01-10	2023-03-26	*NULL*	0.0	Current Loan	0.00
Ken Angel	Gideon Spies	2023-02-18	2023-03-18	*NULL*	0.0	Current Loan	0.00
Ken Angel	Gideon Spies	2023-02-28	2023-03-20	*NULL*	0.0	Current Loan	0.00
Ken Angel	Gideon Spies	2023-03-10	2023-03-14	2023-03-16	0.2	Previous Loan	0.00
Ken Angel	To Kill a Mo…	2023-02-10	2023-03-14	*NULL*	0.0	Current Loan	0.00
Ken Angel	To Kill a Mo…	2023-03-01	2023-03-30	*NULL*	0.0	Current Loan	0.00

Figure 2.6 – Loan History View result set.

From the result set, the loan on the second row is a previous loan but was also returned before the due date which is why it didn't attract a fine unlike the Gideon Spies loan that was returned 2 days after the due date, and it attracted a fine of 20pence which is 10pence per day. The rest are current loans.

4. Trigger (Update Item Status)

```sql
/*4. Create triggers so that the current status of an item automatically updates to Overdue if the item has not
 beenreturned by the due date, and so that the current status updates to Available when the book is returned.

/* Trigger to update item status to available or overdue    */

GO

CREATE TRIGGER TRG_UpdateItemStatus
ON Loan
AFTER INSERT, UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- Update status to Available when the book is returned
        UPDATE ItemCopy
        SET Status_ID =
              -- Available status
             (SELECT Status_ID from  Current_Status_Lookup WHERE Status_Name = 'Available')
        FROM inserted i
        INNER JOIN Loan l ON i.LoanID = l.LoanID
        INNER JOIN ItemCopy c ON l.CopyID = c.CopyID
        WHERE l.Date_Returned IS NOT NULL
        AND c.Status_ID IN
                 -- Only update if status is On Loan or Overdue
                (SELECT Status_ID from  Current_Status_Lookup WHERE Status_Name IN ('On Loan','Overdue'));

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- If an error occurs, roll back the transaction and print an error message
        ROLLBACK TRANSACTION;
        PRINT ERROR_MESSAGE();
    END CATCH;
END
```

Figure 2.6 – Update Item status trigger T-SQL script. Image generated from www.snappify.com

This trigger will activate following an insert or update on the Loan table once an item has been returned, it will change the status of the corresponding ItemCopy record. If the item has been returned, the status will be changed to "Available," (with available ID from the status table) only if the current status is Loan" or "Overdue". This trigger runs as a single transaction, and any errors are caught and rolled back.

4.1 Testing the Trigger

Results Messages

	CopyID	ItemTitle	Status_Name
1	16	Sleeping Dog Lies	On Loan

Figure 2.7 – Item status BEFORE the update on the Loan table

Results Messages

	CopyID	ItemTitle	Status_Name
1	16	Sleeping Dog Lies	Available

Figure 2.8 – Item status AFTER the update on the Loan table

5. Function(Total Number of loans on a given date)

```sql
/* Function to return the number of loans on a specific date */

GO

CREATE FUNCTION Func_GetLoansOnDate (@loanDate DATE)
RETURNS INT
AS
BEGIN
    DECLARE @numLoans INT;

    SELECT @numLoans = COUNT(*) --count the number of rows
    FROM Loan
    -- loan date that matches the parameter passed
    WHERE CONVERT(DATE, Date_Taken_Out) = @loanDate;
    RETURN @numLoans; -- returns the count
END

GO


-- testing the function
SELECT dbo.Func_GetLoansOnDate('2022-01-01') AS NumLoans;
```

Figure 2.9 – Loan count Function T-SQL script. Image generated from www.snappify.com

6. Inserting into the various tables

```sql
/* The Following insert helped me in the testing of my
SELECT queries, user-defined functions, stored procedures, and triggers */

-- Inserting records into Item Lookup table
INSERT INTO Item_Type_Lookup (Item_Name)
VALUES ('Book'), ('DVD'),('CD'),('Other Media');

-- Inserting records into status Lookup table
INSERT INTO Current_Status_Lookup (Status_Name)
VALUES ('Available'),('On Loan'),('Overdue'),('Lost/Removed') ;

-- Inserting records into Payment_Method Lookup table
INSERT INTO Payment_Method_Lookup (Payment_Name) VALUES ('Cash'),('Card');

-- Inserting records into Member_Category Lookup table
INSERT INTO Member_Category (CategoryName, CategoryDesc)
VALUES ('Basic', 'Basic membership category')
,('Premium', 'Premium membership category');


-- Insert two new member table

EXEC InsertMember @FirstName = 'Ken',@LastName = 'Angel',@MemberDOB = '1967-10-10',
    @EmailAddress = 'ken.angel@magic.com',@PhoneNumber = '078495484',  @MemberCategoryID = 1,
    @Username = 'ken.angel',@PasswordH = 'Kennyangel*1',@Address1 = 'Helside Daytun',
    @Address2 = NULL,@City = 'Dayton',@A_State = 'Utah',@Country = 'USA',@Postcode = '99032',
    @Address_type = 'Home';

EXEC InsertMember @FirstName = 'Dr',@LastName = 'NathnAr',@MemberDOB = '1983-05-15',
    @EmailAddress = 'drnatsa@salforduni.com',@PhoneNumber = '07495984895',  @MemberCategoryID = 2,
    @Username = 'dr.nathan',@PasswordH = 'NathyNat*1',@Address1 = 'SSEE Building, Salford Crescent',
    @Address2 = NULL,@City = 'Salford',@A_State = 'Manchester',@Country = 'UK',@Postcode = 'M54WT',
    @Address_type = 'Office';


-- Inserting records into member login audit

INSERT INTO Member_Login(MemberID, Ip_address, User_agent)
VALUES (1, '192.168.1.100', 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'),
     (2, '192.168.1.101', 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) '),
      (1, '192.168.1.110', 'Andriod'),
     (2, '192.168.1.121', 'Iphone');

-- Inserting records into  item table
INSERT INTO Item (ItemTypeID, ItemTitle, Author, PublishedYear)
VALUES (1, 'To Kill a Mockingbird', 'Harper Lee', 1960),
(1, 'Sleeping Dog Lies', 'George Orwell', 1949),
(2, 'The Godfather', 'Mario Puzo', 1969)
,(3, 'Graduation', 'Kanye West', 2008);

-- Inserting records into itemcopy
INSERT INTO ItemCopy (ItemID, Date_Added, Status_ID, Lost_Removed_date, ISBN,PublishedYear)
VALUES  (2, '2022-01-01', 3, NULL, NULL,NULL),
        (1, '2023-01-01', 3, NULL, NULL,NULL),
        (3, '2023-01-01', 3, NULL, NULL,NULL),
        (2, '2022-01-01', 3, NULL, NULL,2001),
        (1, '2023-01-01', 3, NULL, NULL,NULL),
        (3, '2023-01-01', 3, NULL, NULL,NULL)   ; -- available


-- Inserting records into itemcopy
INSERT INTO ItemCopy (ItemID, Date_Added, Status_ID, Lost_Removed_date, ISBN,PublishedYear)
VALUES  (2, '2022-01-11', 2, NULL, NULL,NULL),
        (1, '2023-01-02', 2, NULL, NULL,NULL),
        (3, '2023-01-03', 2, NULL, NULL,NULL),
        (2, '2022-01-04', 2, NULL, NULL,2002),
        (1, '2023-01-05', 2, NULL, NULL,NULL),
        (3, '2023-01-06', 2, NULL, NULL,NULL) ; -- On loan


-- inserting into itemcopy
INSERT INTO ItemCopy (ItemID, Date_Added, Status_ID, Lost_Removed_date, ISBN,PublishedYear)
VALUES (1, '2020-01-02', 4, '2023-02-20', '984393sjdhdjs',1960),
        (2, '2022-01-12', 4, '2023-01-10', '48787437834hdh',1949),
        (3, '2023-01-22', 4, '2023-03-10', '094394kjfd',1969);
        -- lost(update the ISBN) -- lost(update the ISBN)

 -- -- Inserting into  loan table

INSERT INTO Loan (MemberID, CopyID, Date_Taken_Out, Date_Due_Back, Date_Returned, FineAmount)
VALUES (1, 10, '2023-02-28', '2023-03-10', NULL, NULL), --overdue
         (1, 11, '2023-02-18', '2023-03-08', NULL, NULL), -- overdue
          (1, 12, '2023-03-10', '2023-03-14', NULL, NULL), -- due in 4 days
          (1, 13, '2023-02-10', '2023-03-14', NULL, NULL), -- due in 3 days
          (2, 14, '2023-01-10', '2023-03-12', NULL, NULL), -- due in 1 days
          (2, 15, '2023-02-10', '2023-04-12', NULL, NULL), -- next month
          (2, 16, '2023-03-10', '2023-03-13', NULL, NULL), -- due 2 days
          (2, 17, '2023-01-10', '2023-02-13', NULL, NULL), -- overdue
           (2, 17, '2023-01-12', '2023-03-01', NULL, NULL); -- overdue

-- Inserting into fine repayment table
INSERT INTO Fine_Repayment(loanID,Memberid,AmountPaid,PaymentDate,PaymentID)
           VALUES (2,1,0.14,'2023-02-12',1) ;
```

Figure 2.10 – Inserting into the various tables. Image generated from www.snappify.com

7. Other objects I created

7.1 View

```sql
/*
This view display all the items that are currently overdue, along with the member who
borrowed them and the amount of the overdue fine. This view selects information from the
loan, member, item, itemcopy, and Item_Type_Lookup tables to show details of overdue items.
The overdue fee calculation assumes a rate of 10p per day(which is based on clients requirement),
 and the view only includes items that are currently on loan and overdue.
 */

GO

CREATE VIEW VW_OverDueLoanItems

AS

SELECT

            CONCAT(m.FirstName,' ',m.LastName) 'Member Full Name',
            i.ItemTitle 'Item Title',
            i.Author 'Item Author',
            i.PublishedYear 'Published Year',
            t.Item_Name 'Item Name',
            l.Date_Taken_Out 'Loan Date',
            l.Date_Due_Back 'Loan Date Due',
    CASE
        -- item is overdue and item is not returned
        WHEN l.Date_Returned IS NULL AND l.Date_Due_Back < GETDATE() THEN
            DATEDIFF(day, l.Date_Due_Back, GETDATE())  -- number of days overdue
        ELSE
            0
    END AS 'Days Overdue',
    CASE
        WHEN l.Date_Returned IS NULL AND l.Date_Due_Back < GETDATE() THEN
            DATEDIFF(day, l.Date_Due_Back, GETDATE()) * 0.1  -- overdue charge times 10p per day
        ELSE
            0
    END AS 'Overdue Fee'

FROM Loan l
    JOIN ItemCopy c
    ON l.CopyID = c.CopyID -- joining the loan and itemcopy table
    JOIN item i
    ON i.ItemID = c.ItemID -- joining the item and itemcopy table
    JOIN Item_Type_Lookup t
    ON t.Item_Type_ID = i.ItemTypeID --joining the item lookup table for the item type name
    JOIN member m
    ON m.memberID = l.MemberID -- joining the loan to the member table

WHERE
    l.date_returned IS NULL  --item not returned
    AND l.Date_Due_Back < GETDATE(); -- return date is less than today
GO

-- viewing the OverDueLoanItems view
SELECT * FROM dbo.VW_OverDueLoanItems;
```

Figure 2.11 – Viewing Overdue Loan Items. Image generated from www.snappify.com

7.2 Stored Procedure

7.2.1 Update Fine Amount Daily Stored Procedure

```sql
CREATE PROCEDURE SP_DailyUpdateLoanFineAmount
AS
BEGIN
    BEGIN TRY
        DECLARE @CurrentDate DATE = GETDATE(); -- the current day
        DECLARE @LoanID INT, @DueDate DATE, @FineAmount MONEY,@returnDate DATE;

        -- the records are stored in a cursor for easy looping
        DECLARE LoandCursor CURSOR FOR
            SELECT LoanID, Date_Due_Back, FineAmount,Date_Returned
            FROM Loan ;
        OPEN LoandCursor; -- opening the cursor
        FETCH NEXT FROM LoandCursor INTO @LoanID, @DueDate, @FineAmount,@returnDate;

        WHILE @@FETCH_STATUS = 0
        BEGIN
            IF @DueDate < @CurrentDate AND @returnDate IS NULL  -- Item not returned
            BEGIN
                -- calculate the number of day overdue and multiply by 10p
                SET @FineAmount = DATEDIFF(DAY, @DueDate, @CurrentDate) * 0.10;
                -- check how much has been repaid for this particular loan
                -- from the fine repayment table and take it from the fine amount
                UPDATE Loan SET FineAmount
                            = @FineAmount - (SELECT ISNULL(SUM(AmountPaid),0)
                                                FROM Fine_Repayment WHERE loanID = @LoanID)
                WHERE LoanID = @LoanID;
            END

            IF @DueDate < @returnDate AND @returnDate IS NOT NULL  -- Item is returned
            BEGIN
                -- calculate the number of day overdue and multiply by 10p
                SET @FineAmount = DATEDIFF(DAY, @DueDate, @returnDate) * 0.10;
                -- check how much has been repaid for this particular loan
                -- from the fine repayment table and take it from the fine amount
                UPDATE Loan SET FineAmount
                            = @FineAmount - (SELECT ISNULL(SUM(AmountPaid),0)
                                                FROM Fine_Repayment WHERE loanID = @LoanID)
                WHERE LoanID = @LoanID;
            END

             IF @DueDate > @returnDate AND @returnDate IS NOT NULL  -- Item is returned
            BEGIN
                -- item was returned earlier than the due date
                UPDATE Loan SET FineAmount
                            = 0
                WHERE LoanID = @LoanID;
            END

            FETCH NEXT FROM LoandCursor INTO @LoanID, @DueDate, @FineAmount,@returnDate; -- fetch next record
        END

        CLOSE LoandCursor;
        DEALLOCATE LoandCursor;
    END TRY
    BEGIN CATCH
        -- Handle any errors that occurred during the execution of the stored procedure.
        SELECT ERROR_MESSAGE() AS ErrorMessage;
    END CATCH
END;
```

Figure 2.12 – Daily Update of Fine Amount. Image generated from www.snappify.com

One of the best features of SSMS is that it enables the client to schedule a “SQL Server Agent Job” to run this stored procedure daily. This can be set to run before the start of business so that all loan fines can be updated. This stored procedure takes note of repayment that has been made on the loan which was indicated in the script comments. If all the fine has been paid and the item returned, the amount will always be 0.

Results Messages

	LoanID	MemberID	CopyID	Date_Taken_Out	Date_Due_Back	Date_Returned	FineAmount
1	1	1	7	2023-02-28	2023-03-10	NULL	0.34
2	2	1	7	2023-02-18	2023-03-08	NULL	0.60
3	3	1	7	2023-03-10	2023-03-14	NULL	0.00
4	4	1	8	2023-02-10	2023-03-14	NULL	NULL
5	5	2	8	2023-01-10	2023-03-12	NULL	0.00
6	6	2	9	2023-02-10	2023-03-13	NULL	NULL
7	7	2	10	2023-03-10	2023-03-15	NULL	NULL
8	8	1	11	2023-03-01	2023-03-18	NULL	0.00
9	9	2	12	2023-01-10	2023-02-13	NULL	NULL
10	10	2	12	2023-01-12	2023-03-01	NULL	NULL

	RepaymentID	LoanID	Memberid	AmountPaid	PaymentDate	PaymentID
1	1	1	1	0.06	2023-03-14	2

Figure 2.13 – Before running the stored procedure.(LoanID 1 is 0.34p)

Results Messages

	LoanID	MemberID	CopyID	Date_Taken_Out	Date_Due_Back	Date_Returned	FineAmount
1	1	1	7	2023-02-28	2023-03-10	NULL	0.28
2	2	1	7	2023-02-18	2023-03-08	NULL	0.60
3	3	1	7	2023-03-10	2023-03-14	NULL	0.00
4	4	1	8	2023-02-10	2023-03-14	NULL	NULL
5	5	2	8	2023-01-10	2023-03-12	NULL	0.20
6	6	2	9	2023-02-10	2023-03-13	NULL	0.10
7	7	2	10	2023-03-10	2023-03-15	NULL	NULL
8	8	1	11	2023-03-01	2023-03-18	NULL	0.00
9	9	2	12	2023-01-10	2023-02-13	NULL	2.90
1…	10	2	12	2023-01-12	2023-03-01	NULL	1.30

Figure 2.14 – After running the stored procedure. (LoanID 1 is 0.28p after 0.06p was paid)

7.2.2 Update Lost/Removed Item Stored Procedure

```
/* 7.2.2
 this Procedure updates the item status to lost/removed
and updates the ISBN if the item is a book */

GO
CREATE PROCEDURE UpdateItemCopyStatusAndISBN
    @IN_CopyID INT,
    @lost_remove_date DATE,
    @IN_ISBN NVARCHAR(20) = NULL
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- update the status
        UPDATE ItemCopy
        SET Status_ID =
        -- set statusId to Lost/Removed
        (SELECT Status_ID from  Current_Status_Lookup WHERE Status_Name = 'Lost/Removed'),
        -- set lost_remove_date to
         Lost_Removed_date = @lost_remove_date
        WHERE CopyID = @IN_CopyID;

        -- check to see if the item copy is a book
        IF EXISTS (SELECT * FROM ItemCopy ic JOIN Item i ON ic.ItemID = i.ItemID
                    WHERE ic.CopyID = @IN_CopyID
                    AND i.ItemTypeID = (select Item_Type_ID from Item_Type_Lookup
                    where Item_Name = 'Book') )
        BEGIN
             -- because it's a book, update the ISBN column
            UPDATE ItemCopy
            SET ISBN = @IN_ISBN
            WHERE CopyID =  @IN_CopyID;
        END

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END
GO
```

Figure 2.15 – Update lost/removed item stored procedure. Image generated from www.snappify.com

One of the client's requirements is to update the date an item is lost/removed and update the ISBN if the item is a book. This is what this stored procedure is going to achieve. The first update part of the code updates the status to lost/removed and the date it is lost/removed which will be supplied

to the stored procedure while the second part checks if the item is a book and then updates the ISBN. Passing the ISBN to the stored procedure is an option in case the item is not a book.

7.3 User-Defined Function

```sql
/*
This function takes in a loan ID and calculates the number of days the loan is overdue by
subtracting the due date from the current date. If the loan is overdue, the function calculates
the overdue fine at a rate of 10p per day. If the loan is not overdue, the function returns a value of 0.
 */


GO
CREATE FUNCTION Func_CalculateOverdueFineAmount
(
    @loanID INT
)
RETURNS NVARCHAR(100)
AS
BEGIN
    DECLARE @overdueDays INT, @overdueFine MONEY, @date_borrow DATE

    -- selecting the day it was borrowed
    SELECT @date_borrow =   Date_Taken_Out
    FROM Loan
    WHERE LoanID = @loanID

    -- selecting the days overdue
    SELECT @overdueDays = DATEDIFF(DAY, Date_Due_Back, GETDATE())
    FROM Loan
    WHERE LoanID = @loanID

    -- calculating fine per 10p a day
    IF @overdueDays > 0
    BEGIN
        SELECT @overdueFine = @overdueDays * 0.10
    END
    ELSE
    BEGIN
        SELECT @overdueFine = 0
    END

    -- return the fine
    RETURN CONCAT('This Item was borrowed on: ', @date_borrow ,' and has a fine of : ',@overdueFine)
END;
GO

-- Testing the Function
SELECT dbo.Func_CalculateOverdueFineAmount(1) as 'Fine Narrative';

```

Figure 2.16 –overdue fine amount function calculator. Image generated from www.snappify.com

Results Messages

	Fine Narrative
1	This Item was borrowed on: 2023-02-28 and has a fine of : 0.50

Figure 2.16.1 –overdue fine amount function result

7.4 Trigger

7.4.1 Update Fine Amount Trigger

```
/*
This trigger updates the FineAmount in the Loan table when a repayment is made in the Fine_Repayment table.
It uses a transaction to ensure that the update is atomic and either fully committed or fully rolled back in case of an error
*/

GO

CREATE TRIGGER TRG_UpdateFineAmount
ON Fine_Repayment
AFTER INSERT --after insert on fine_repayment table
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @LoanID INT; -- loan id
    DECLARE @AmountPaid MONEY; -- amount repaid
    DECLARE @FineAmount MONEY; -- fine amount on the loan

    -- Begin transaction
    BEGIN TRAN;

    -- Get the LoanID and AmountPaid for the inserted repayment
    SELECT @LoanID = LoanID, @AmountPaid = AmountPaid
    FROM inserted;

    -- Get the current FineAmount for the loan
    SELECT @FineAmount = FineAmount
    FROM Loan
    WHERE LoanID = @LoanID;

    -- Update the FineAmount in the Loan table
    UPDATE Loan
    SET FineAmount = CASE
                        --checks to see if fine amount is less than 0
                        WHEN @FineAmount - @AmountPaid < 0 THEN 0
                        -- if not less than 0, it's deducted
                        ELSE @FineAmount - @AmountPaid
                    END
    WHERE LoanID = @LoanID;

    -- Commit transaction
    COMMIT TRAN;
END;
```

Figure 2.17 –Trigger to update FineAmount. Image generated from www.snappify.com

Testing the trigger :

Results Messages

	LoanID	MemberID	CopyID	Date_Taken_Out	Date_Due_Back	Date_Returned	FineAmount
1	2	1	11	2023-02-18	2023-03-08	*NULL*	0.40

Figure 2.18 Loan Table before Trigger fired.

Results Messages

	LoanID	MemberID	CopyID	Date_Taken_Out	Date_Due_Back	Date_Returned	FineAmount
1	2	1	11	2023-02-18	2023-03-08	*NULL*	0.26

Figure 2.19 Loan Table After Trigger fired.

7.4.2 Update Item Copy Status Trigger

```
/* this trigger updates the status of an item to on loan once
   there's an insert of the item copy in the loan table */
GO
CREATE TRIGGER TRG_UpdateItemCopyStatus
ON Loan
AFTER INSERT
AS
BEGIN
  SET NOCOUNT ON;
  BEGIN TRY
    BEGIN TRANSACTION;

    -- Declare variables to hold the member ID and copy ID of the item
    DECLARE @MemberID INT;
    DECLARE @CopyID INT;

    -- Get the member ID and copy ID from the inserted row in the Loan table
    SELECT @MemberID = MemberID, @CopyID = CopyID FROM inserted;

    -- Update the status of the corresponding item copy in the ItemCopy table
    UPDATE ItemCopy
    SET Status_ID = (
        -- Change the status to "On Loan"
        SELECT Status_ID from  Current_Status_Lookup WHERE Status_Name = 'On Loan'
    )
    WHERE CopyID = @CopyID;

    COMMIT TRANSACTION;
  END TRY
  BEGIN CATCH
    IF @@TRANCOUNT > 0
      ROLLBACK TRANSACTION;

    -- Log the error message
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    RAISERROR('Error occurred in TRG_UpdateItemCopyStatus: %s', 16, 1, @ErrorMessage);
  END CATCH;
END;

GO
```

Figure 2.20 –Trigger to update item copy status. Image generated from www.snappify.com

This trigger will be fired after a new row is inserted into the Loan table. It will update the Status_ID of the corresponding item copy in the ItemCopy table to (On Loan) since the item has now been borrowed.

7.5 Advice and Guidance to Client

7.5.1 Data Integrity and concurrently

Data integrity is how accurate, consistent, and reliable data in the database is, while data concurrency allows multi users to access and modify data in the database without causing inconsistencies or conflict. To ensure this in the database, some of these measures were put in place; unique constraints have been defined on the email address and username fields in the Member table to prevent duplicate values, and foreign key constraints have been defined to maintain referential integrity between tables. The use of transactions to ensure that multiple database operations are treated as a single unit of work, and the use of foreign key constraints to prevent actions that would violate referential integrity.

7.5.2 Data Security

I've put some measures in place for adequate security of the system and some of them are discussed below:

7.5.2.1 Password Hashing

I created, the 'passwordhash' column in the Member table which stores the plaintext password of a user in a hash form. By storing the password hash in the database rather than the plaintext password, an attacker will be unable to recover the original passwords if they gain unauthorized access to the database. This is due to the computational impossibility of reversing the hashing process and obtaining the original password from the hash. Hence, this improves data security.

7.5.2.2 Access control

Access control is also an important security feature a database should have and it entails managing user permissions and privileges to limit access to sensitive data and ensure data security. This can be achieved by database roles and user accounts. Different roles can be created in the client's database such as a role for a Liberian, a manager, a clerk and many more. Each of these roles will have different privileges and access which means a person who does not have VIEW access to a view in the database won't be able to see or use it.

The code below creates a user and grants the user "exec" access to the stored procedure for capturing member information. It means this user can capture the information of a new member using the stored procedure he or she has access. A user who doesn't have granted access to this stored procedure won't be able to do this. Revoking the access also helps with limiting existing access to an object for a user. These measures improve the security of the database tremendously and it's some of the advice I'm offering to the client on database security

```sql
-- Create a new database user with a login and password
CREATE LOGIN liberian_user WITH PASSWORD='newusUvxs245!';
CREATE USER liberian_user FOR LOGIN liberian_user;

-- Grant Executing access to the created User
GRANT EXEC ON dbo.SP_InsertMember TO liberian_user;

```

Figure 2.21 –Database access control. Image generated from www.snappify.com

7.5.3 Data backup and recovery

SQL Server provides different backup and recovery strategies, some of the ones I'll advice

- The client can perform a full backup, which includes backing up all data, indexes, and log files. I would advise the client to perform a full back-up once a week.
- The client could also use a differential backup to continue backup from the last full backup rather than performing a new full backup; this can be done every 12 hours or daily. this allows for a quicker restore than using only full backups.
- The client can also perform transaction log backup, which backs up the transaction log, which contains all the transactions since the last log backup or full backup, which can help the client restore the database to a specific point in time.

7.6 Conclusion

The database schema has been created with the client's requirement in mind and every other object and measure put in place was discussed in this report. The database schema I designed and documented in this report allows for the efficient management of items, members, and loans, as well as the simple storage and retrieval of information about each. The use of stored procedures improves the system's security and reliability by allowing for controlled access to the database, while the use of views allows for quick and easy retrieval of specific information.

Overall, the database schema provides a strong foundation for the client's library management system, with room for customization and expansion as needed which couldn't be discussed further due to the limitation of this report.