Advanced Databases
MSc Data Science

DATABASE DESIGN & SQL FOR DATA ANALYSIS

NAME: ABDULAZEEZ MOMOH

STUDENT ID: @00686172

DUE DATE: 28th APRIL 2023

Table of Contents

TASK 1: DESIGN AND IMPLEMENTATION OF A LIBRARY MANAGEMENT DATABASE SYSTEM

1. INTRODUCTION

1.1 BACKGROUND

The modern library environment necessitates the creation of a sophisticated, efficient, and all-encompassing database system capable of managing various aspects, such as member information, library catalog, loan history, and overdue fine repayments. This report focuses on designing and implementing a database system for a library that is in the process of developing a new database to store and manage these critical components of their operations.

1.2 SCOPE

The client has provided a comprehensive set of requirements to guide the design of the database system. These requirements encompass information about library members, their login credentials, and optional contact information. Furthermore, the library needs to manage overdue fines, fine repayments, and record the date of membership termination. The library catalog must store details about items, their types, authors, publication years, and statuses, along with specific information for books. Another crucial aspect to consider is the loan history, including member and item details, as well as loan dates and overdue fees (calculated at a rate of 10p per day).

1.3 OBJECTIVE

As a database developer consultant, my understanding of the task is to create a database system tailored to the client's requirements while adhering to the best practices in database design and normalization maintaining data integrity, performance, and reliability using T-SQL statements in Microsoft SQL Server Management Studio. My proposed approach for tackling this task involves the following steps:

- Identify the entities and their attributes based on the client requirements.

- Define relationships between the entities and determine primary and foreign keys.
- Normalize the database design to achieve 3NF, ensuring data integrity and avoiding redundancy.
- Create the database, tables, and constraints using T-SQL statements in Microsoft SQL Server Management Studio. Justify the choice of data types for each column.
- Develop necessary database objects such as stored procedures, user-defined functions, views, and triggers to support the library's operations and reporting needs.
- Document the design process and decisions made, including any assumptions, using screenshots and T-SQL statements.
- Create a database diagram to visualize the database structure.
- Prepare a comprehensive report explaining the design process, justifications, and the T-SQL statements used.

The report will document the process of designing the database system, including the T-SQL statements used in Microsoft SQL Server Management Studio, as well as explanations and justifications for the design decisions. By following the structure outlined in this introduction and the proposed approach, the report aims to provide a comprehensive overview of the design and implementation of a library management database system tailored to the client's specific needs.

PART 1: COMPREHENSIVE DESIGN AND IMPLEMENTATION OF A LIBRARY DATABASE SYSTEM IN SQL SERVER

2.0 DATABASE DESIGN AND NORMALIZATION

In this section, I will delve into the intricacies of the library database system's design and development, highlighting the methodology, decisions, and the rationale behind them. The primary objective was to construct a database that efficiently and effectively stores information related to library members, their loan history, the library catalog, and overdue fine repayments. This design process comprised requirements gathering, making assumptions, and creating tables and their relationships in compliance with normalization rules.

2.1 ASSUMPTIONS

Before diving into the design, I formulated the following assumptions based on the client requirements to ensure a comprehensive and functional database design.

A. Overdue fines are calculated on a daily basis and don't depend on the item type.
B. Overdue fine repayments can be partial or in full, but the payment should not exceed the outstanding balance.
C. The library doesn't require information on the staff responsible for processing loans or repayments.

2.2 NORMALIZATION

Normalization is a systematic approach to organizing data in a relational database to minimize redundancy and improve data integrity. It is a step-by-step process that involves decomposing tables and establishing relationships between them based on specific rules, known as normal forms. The primary goal of normalization is to create a database design that is both efficient and flexible in terms of storage, retrieval, and maintenance of data. The importance of normalization of our database can be highlighted through its benefits which include:

1. Eliminating Data Redundancy by separating data into related tables and columns, this ensured that each piece of information is stored in only one place. This not only reduces storage requirements but also ensures consistency, as updates or deletions need to be performed in a single location.
2. Improving Data Integrity by enforcing rules and constraints to ensure that relationships between tables are properly maintained, preventing the insertion of

inaccurate or inconsistent data. This, in turn, improves the overall reliability of the database system.

3. Facilitating Easier Maintenance and Querying as changes to the data or schema can be performed with minimal impact on the rest of the system. Additionally, a normalized database structure allows for more efficient querying, as it reduces the need for complex joins and sub-queries, leading to faster response times.
4. Enhancing Flexibility by promoting a modular and flexible database design that can be easily expanded or modified to accommodate new requirements. This is particularly important for businesses and organizations that need to adapt to changing market conditions, regulations, or technological advancements.

2.3 DESIGN

To design the database in 3NF (Third Normal Form), we must first ensure that it is in 1NF (First Normal Form) and 2NF (Second Normal Form). By following these normalization steps, we can create an efficient, flexible, and scalable database design that minimizes redundancy and ensures data integrity.

1NF (First Normal Form):
In the first step, the raw data provided by the library was organized into tables, ensuring that each table contains atomic values, with no repeating groups or nested tables. Each table was assigned a primary key to uniquely identify each row. The tables created at this stage include Members, Addresses, Items, Loans, OverdueFines, and FineRepayments. The tables created at this stage are:

a. Members
b. Addresses
c. Items
d. Loans
e. OverDueFines
f. FineRepayments

2NF (Second Normal Form):
The database was further refined by ensuring that each table only contains data related to a single concept, thereby eliminating partial dependencies. For example, member address information was separated from the Members table and placed in the Addresses table, with a foreign key relationship established between the two tables. Our designs is in 2NF because all non-prime attributes depend on their respective primary keys.

3NF (Third Normal Form):
In the final step, transitive dependencies were removed by creating separate tables for concepts that are not directly dependent on the primary key. This step involved creating an Items table to store information about individual library items, and the Loans table to store information about loan transactions..
Our design adheres to 3NF by eliminating transitive dependencies between non-prime attributes:

2.4 JUSTIFICATION FOR DESIGN

1. Addresses table:

The Addresses table separates the address information from the Members table. This not only reduces redundancy, but also allows for the possibility of having multiple addresses per member in the future if needed. Additionally, updating or deleting address records becomes more efficient and manageable. The UNIQUE constraint on Address1 and Postcode helps maintain data integrity by preventing duplicate addresses.

2. Members table:

The Members table is designed to store information about library members. It is in 3NF because all its columns are functionally dependent on the primary key (MemberID) and there are no transitive dependencies or repeating groups. The Members table now references the Addresses table using a foreign key (AddressID), separating member information from their address data. This not only reduces redundancy but also allows for the possibility of having multiple addresses per member in the future if needed

3. Items table:

The Items table stores information about the library's catalog items. It is in 3NF because all columns depend on the primary key (ItemID), and there are no transitive dependencies or repeating groups. The ISBN column is only applicable to books, but it is included in this table to simplify the design, and it can be NULL for other item types.

4. Loans table:

The Loans table represents the relationship between Members and Items when an item is borrowed. It is in 3NF because all columns are functionally dependent on the primary key (LoanID), and there are no transitive dependencies or repeating groups. The table has two foreign keys, MemberID and ItemID, which link the Loans table to the Members and Items tables, respectively.

5. OverdueFines table:

The OverdueFines table stores information about the fines incurred by members for overdue items. It is in 3NF because all columns depend on the primary key (FineID), and there are no transitive dependencies or repeating groups. The table has two foreign keys, MemberID and LoanID, linking it to the Members and Loans tables, respectively.

6. FineRepayments table:

The FineRepayments table records the repayments made by members for their overdue fines. It is in 3NF because all columns depend on the primary key (RepaymentID), and there are no transitive dependencies or repeating groups. The table has one foreign key, FineID, linking it to the OverdueFines table.

The design of the database is in the 3rd Normal Form (3NF) to ensure data integrity, reduce redundancy, and improve efficiency. Each table is designed to represent a unique entity and their corresponding relationships. By designing the database in 3NF, we ensure that data is logically organized, which makes it easier to manage, query, and update. The design also eliminates the risk of data anomalies, such as insertion, update, and deletion anomalies. The use of primary and foreign keys ensures referential integrity between tables, which is crucial for maintaining data consistency in a relational database.

2.5 ASSUMPTIONS MADE DURING NORMALIZATION

While designing the database, I made the following assumptions;

1. Username uniqueness: I assumed that the Username field in the Members table must be unique for each member, so no two members can have the same username. This will ensure that there are no conflicts during the login process.
2. Item title and author uniqueness: I assumed that a combination of Title and Author should be unique in the Items table, so no two items can have the same title and author. This will help to avoid confusion and maintain a well-organized catalog.
3. Maximum length of fields: I assumed reasonable maximum lengths for fields like FirstName, LastName, Address, Username, Password, Email, PhoneNumber, Title, Author, and RepaymentMethod. These assumptions were made to allocate appropriate storage space and ensure efficient performance. However, these lengths can be adjusted according to the specific requirements of the library.
4. Library operation hours: I assumed that the library operates daily, and the calculation of overdue fines takes into account weekends and holidays. If the library has specific operational hours or holidays, the calculation of overdue fines should be adjusted accordingly. Also that the library doesn't require information on the staff responsible for processing loans or repayments.

5. Fine calculation and repayment: I assumed that the library calculates and applies overdue fines on a daily basis, and the fines are stored in the OverdueFines table. Members can make repayments, and the library records each repayment transaction in the FineRepayments table. The remaining balance can be calculated by summing the AmountRepaid values and subtracting it from the AmountDue in the OverdueFines table. Also that repayments can be partial or in full, but the payment should not exceed the outstanding balance.

3.0 TABLE AND COLUMN DESIGN

3.1 CREATING THE DATABASE

Prior to creating the table on Microsoft SQL Server Management Studio, The first step would be to create the database on Microsoft SQL Server Management Studio based on the information provided. The database will store a number of associated database objects, such as stored procedures, user-defined functions, views and triggers using the command below. The database will store all the required tables, views, stored procedures and other requirements. The below T-SQL were used for creating and calling up the database respectively.

```
---------------------------------------------------------
--Creating the Database Library
---------------------------------------------------------
CREATE DATABASE LibraryDB;

--Calling up the Database--
USE LibraryDB;
GO
```

3.2 TABLE, COLUMN DESIGN AND CREATION

To create an efficient and effective database system, we structured the data into the following tables. Each table has a primary key that uniquely identifies a record, and relationships are established using foreign keys to maintain referential integrity.

1. Addresses Table

The Addresses table contains address information for library members. The data types and purpose of each column are as follows:

- AddressID (int, PK, identity, not null): It is set as NOT NULL and as the PRIMARY KEY, ensuring that each row has a unique, non-null identifier.
- Address1 (nvarchar (50) not null): It is set as NOT NULL. It will store the first line of the address. NVARCHAR type supports international characters and the length of 50 characters is sufficient for most addresses.
- Address2 (nvarchar (50) null): It is set as NULL, allowing for optional storage of a second line of the address. NVARCHAR type supports international characters and the length of 50 characters is sufficient for most addresses.
- City (nvarchar (50) null): It is set as NULL, which means that storing the city is optional.
- Postcode (nvarchar (10)): It is set as NOT NULL, It will store the postal code for the address. NVARCHAR type supports alphanumeric codes and the length of 10 characters is sufficient for most postal codes.
  CONSTRAINT UC_Address UNIQUE (Address1, Postcode): This is a UNIQUE constraint applied to the combination of the Address1 and Postcode columns. This constraint ensures that no two rows in the table can have the same values for both Address1 and Postcode.

The below T-SQL statements were used for creating Addresses table.

```sql
--a. Creating the Addresses Table
CREATE TABLE Addresses (
AddressID INT IDENTITY NOT NULL PRIMARY KEY,
Address1 NVARCHAR(50) NOT NULL,
Address2 NVARCHAR(50) NULL,
City NVARCHAR(50) NULL,
Postcode NVARCHAR(10) NOT NULL,
CONSTRAINT UC_Address UNIQUE (Address1, Postcode)
);
```

2. Members Table

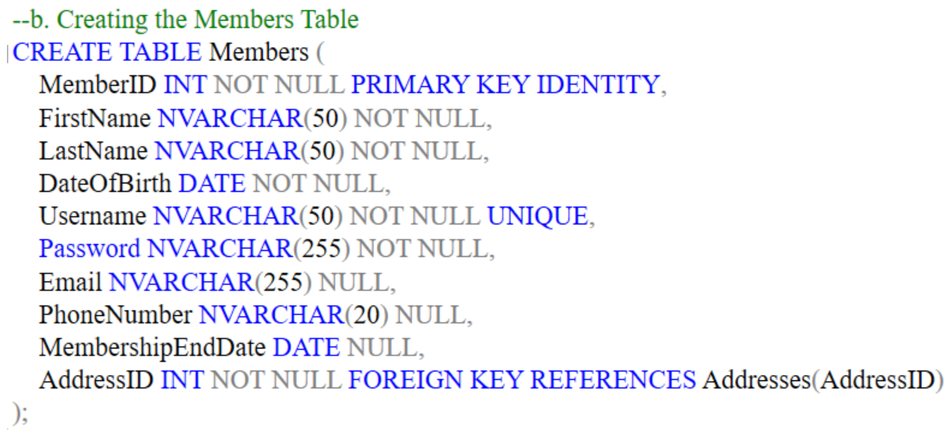
The Members table contains information about the library members, including their first name, last name, date of birth, username, password, email, phone number, membership end date and AddressID. The primary key is the MemberID, and the table uses constraints to ensure data integrity, such as unique usernames.

- MemberID (int, PK, identity): Unique identifier for each member. Using an identity column ensures a unique, auto-incrementing value for each new member added.. The INT data type is suitable for unique identifiers as it supports a wide range of numerical values.
- FirstName (nvarchar (50)): It is set to NOT NULL. It will store the first name of the member. NVARCHAR type supports international characters, and the length of 50 characters is sufficient for most names.
- LastName (nvarchar (50)): It is set to NOT NULL. It will store the last name of the member. NVARCHAR type supports international characters, and the length of 50 characters is sufficient for most names.
- DateOfBirth (date): Member's date of birth. The DATE data type is suitable for storing date values without time.
- Username (nvarchar (50), unique): Member's chosen username, NVARCHAR (50) can store a variety of usernames with different lengths and characters, and

while the UNIQUE constraint ensures that no two members can have the same username.

- Password (nvarchar (255)): It is set to NOT NULL. Member's password (should be hashed and salted).NVARCHAR (255) provides enough space to store hashed and salted passwords.
- Email (nvarchar (255), null): It is set to NULL. Member's email address (optional).NVARCHAR (255) can store email addresses of varying lengths and characters.
- PhoneNumber (nvarchar (20), null): Member's phone number (optional).NVARCHAR (20) can accommodate phone numbers with different formats and lengths.
- MembershipEndDate (date, null): Date when the membership ended (optional).The DATE data type is used for storing date values without time.
- AddressID: Non-nullable foreign key column referencing the Addresses table's AddressID column. It connects each member to their respective address.

The below T-SQL statements were used for creating the Members table.



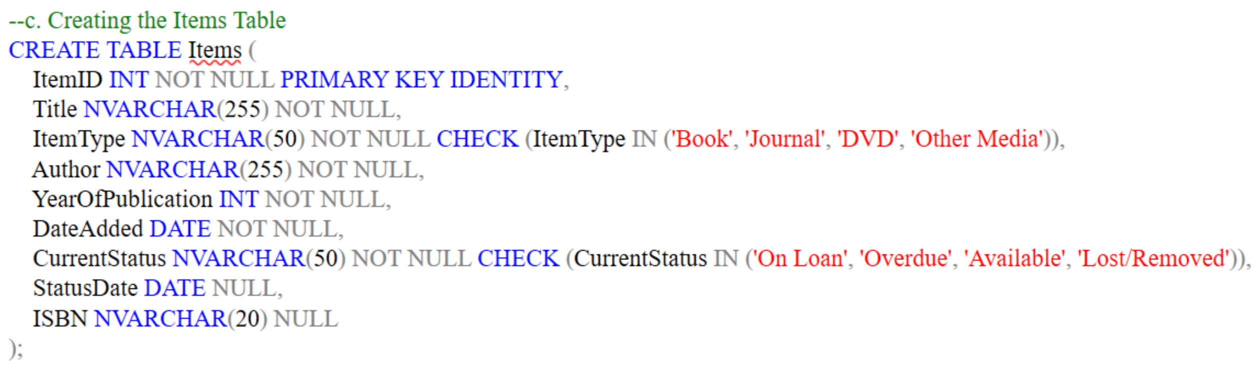
```
--b. Creating the Members Table
CREATE TABLE Members (
    MemberID INT NOT NULL PRIMARY KEY IDENTITY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DateOfBirth DATE NOT NULL,
    Username NVARCHAR(50) NOT NULL UNIQUE,
    Password NVARCHAR(255) NOT NULL,
    Email NVARCHAR(255) NULL,
    PhoneNumber NVARCHAR(20) NULL,
    MembershipEndDate DATE NULL,
    AddressID INT NOT NULL FOREIGN KEY REFERENCES Addresses(AddressID)
);
```

c. Items

The Items table stores information about the library catalogue, including the item title, item type, author, year of publication, date added to the collection, current status, status date, and ISBN (for books). The primary key is the ItemID, and constraints are used to limit the ItemType and CurrentStatus values to a predefined set.

- ItemID (int, PK, identity): Unique identifier for each item. The INT data type is suitable for unique identifiers as it supports a wide range of numerical values.
- Title (nvarchar (255)): Title of the item. NVARCHAR (255) is chosen to accommodate titles with varying lengths and characters, including Unicode characters.
- ItemType (nvarchar (50)): Book, Journal, DVD, or Other Media. NVARCHAR (50) can store a variety of item types with different lengths and characters, and the CHECK constraint ensures that only valid item types are entered.
- Author (nvarchar (255)): Author of the item. NVARCHAR (255) is chosen to accommodate author names with varying lengths and characters, including Unicode characters.
- YearOfPublication (int): Year the item was published. The INT data type is suitable for storing integer values.
- DateAdded (date): Date the item was added to the collection. The DATE data type is used for storing date values without time.
- CurrentStatus (nvarchar (50)): On Loan, Overdue, Available, or Lost/Removed. NVARCHAR (50) can store a variety of statuses with different lengths and characters, and the CHECK constraint ensures that only valid statuses are entered.
- StatusDate (date, null): Date the item was marked as Lost/Removed (optional). The DATE data type is used for storing date values without time.
- ISBN (nvarchar (13), null): ISBN for books (optional). NVARCHAR (13) can accommodate ISBNs with different formats and lengths.

The below T-SQL statements were used for creating the Items table.



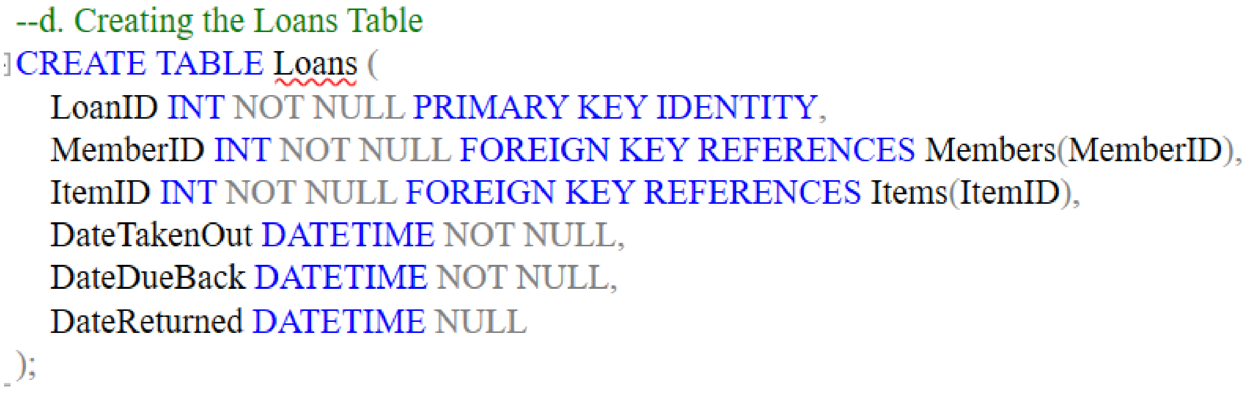
```
--c. Creating the Items Table
CREATE TABLE Items (
   ItemID INT NOT NULL PRIMARY KEY IDENTITY,
   Title NVARCHAR(255) NOT NULL,
   ItemType NVARCHAR(50) NOT NULL CHECK (ItemType IN ('Book', 'Journal', 'DVD', 'Other Media')),
   Author NVARCHAR(255) NOT NULL,
   YearOfPublication INT NOT NULL,
   DateAdded DATE NOT NULL,
   CurrentStatus NVARCHAR(50) NOT NULL CHECK (CurrentStatus IN ('On Loan', 'Overdue', 'Available', 'Lost/Removed')),
   StatusDate DATE NULL,
   ISBN NVARCHAR(20) NULL
);
```

d. Loans

The Loans table keeps a record of all current and past loans, including the member, item, date taken out, date due back, and date returned. The primary key is the LoanID, and foreign keys are used to link the MemberID and ItemID columns to the Members and Items tables, respectively.

- LoanID (int, PK, identity): Unique identifier for each loan. The INT data type is suitable for unique identifiers as it supports a wide range of numerical values.
- MemberID (int, FK): References Members table. The INT data type is used for referencing MemberID in the Member's table, ensuring referential integrity.
- ItemID (int, FK): References Items table. The INT data type is used for referencing ItemID in the Items table, ensuring referential integrity.
- DateTakenOut (datetime): Date the item was borrowed. The DATETIME data type is used for storing date and time values.
- DateDueBack (datetime): Date the item is due to be returned. The DATETIME data type is used for storing date and time values.
- DateReturned (datetime, null): Date the item was returned (optional). The DATETIME data type is used for storing date and time values

The below T-SQL statements were used for creating the Loans table.



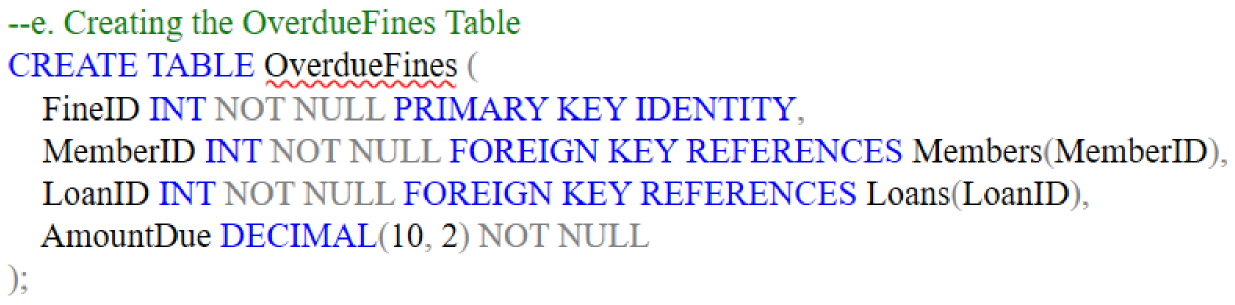
```
--d. Creating the Loans Table
CREATE TABLE Loans (
    LoanID INT NOT NULL PRIMARY KEY IDENTITY,
    MemberID INT NOT NULL FOREIGN KEY REFERENCES Members(MemberID),
    ItemID INT NOT NULL FOREIGN KEY REFERENCES Items(ItemID),
    DateTakenOut DATETIME NOT NULL,
    DateDueBack DATETIME NOT NULL,
    DateReturned DATETIME NULL
);
```

e. OverdueFines

The OverdueFines table tracks overdue fines for library members. It contains the FineID as the primary key, the MemberID and LoanID as foreign keys, and the AmountDue, which represents the fine amount owed for the overdue item.

- FineID (int, PK, identity): Unique identifier for each overdue fine. The INT data type is suitable for unique identifiers as it supports a wide range of numerical values.
- MemberID (int, FK): References Members table. The INT data type is used for referencing MemberID in the Member's table, ensuring referential integrity.
- LoanID (int, FK): References Loans table. The INT data type is used for referencing LoanID in the Loans table, ensuring referential integrity.
- AmountDue (decimal (10, 2)): Amount owed for the overdue item. The DECIMAL data type is used to store precise monetary values.

The below T-SQL statements were used for creating the OverdueFines table.



```
--e. Creating the OverdueFines Table
CREATE TABLE OverdueFines (
   FineID INT NOT NULL PRIMARY KEY IDENTITY,
   MemberID INT NOT NULL FOREIGN KEY REFERENCES Members(MemberID),
   LoanID INT NOT NULL FOREIGN KEY REFERENCES Loans(LoanID),
   AmountDue DECIMAL(10, 2) NOT NULL
);
```

f. FineRepayments

The FineRepayments table records the repayment details for overdue fines, encompassing the repayment date, amount repaid, and repayment method. The primary key, RepaymentID, uniquely identifies each repayment record. The FineID column, a foreign key, references the OverdueFines table to maintain referential integrity.

- RepaymentID (int, PK, identity): Unique identifier for each repayment. The INT data type is suitable for unique identifiers as it supports a wide range of numerical values.
- FineID (int, FK): References OverdueFines table. The INT data type is used for referencing FineID in the OverdueFines table, ensuring referential integrity.
- RepaymentDate (datetime): Date and time of the repayment. The DATETIME data type is used for storing date and time values.

The below T-SQL statements were used for creating the FineRepayments table.

```
--f. Creating the FineRepayments Table
CREATE TABLE FineRepayments (
   RepaymentID INT NOT NULL PRIMARY KEY IDENTITY,
   FineID INT NOT NULL FOREIGN KEY REFERENCES OverdueFines(FineID),
   RepaymentDate DATETIME NOT NULL,
   AmountRepaid DECIMAL(10, 2) NOT NULL,
   RepaymentMethod NVARCHAR(50) NOT NULL CHECK (RepaymentMethod IN ('Cash', 'Card'))
);
```

With the table creation complete, we have successfully implemented our 3NF database design using T-SQL statements in Microsoft SQL Server Management Studio. The data types chosen for each column are appropriate for their respective purposes and constraints are applied to ensure data integrity.

3.3 RELATIONSHIPS

Relationships exist between the tables which have been highlighted below:

- Members table to Addresses table:
  One-to-One relationship: Each member has one address, and each address is associated with one member. This is established through the foreign key MemberID in the Addresses table referencing the primary key MemberID in the Member's table.

- Members table to Loans table:
  One-to-Many relationship: Each member can have multiple loans, but each loan is associated with one member. This is established through the foreign key MemberID in the Loans table referencing the primary key MemberID in the Member's table.

- Items table to Loans table:
  One-to-Many relationship: Each item can be involved in multiple loans, but each loan is associated with one item. This is established through the foreign key ItemID in the Loans table referencing the primary key ItemID in the Items table.

- Members table to OverdueFines table:
  One-to-Many relationship: Each member can have multiple overdue fines, but each overdue fine is associated with one member. This is established through the foreign key MemberID in the OverdueFines table referencing the primary key MemberID in the Member's table.

- Loans table to OverdueFines table:
  One-to-One relationship: Each loan can have one overdue fine, and each overdue fine is associated with one loan. This is established through the foreign key LoanID in the OverdueFines table referencing the primary key LoanID in the Loans table.

- OverdueFines table to FineRepayments table:
  One-to-Many relationship: Each overdue fine can have multiple repayments, but each repayment is associated with one overdue fine. This is established through the foreign key FineID in the FineRepayments table referencing the primary key FineID in the OverdueFines table.

3.4 CONSTRAINTS AND DATA INTEGRITY

In the database design, we have used various constraints to ensure data integrity, and to maintain consistency and accuracy in the data. These constraints are as follows:

1. PRIMARY KEY: This constraint is used to uniquely identify each row in a table. It ensures that no duplicate or NULL values are entered in the primary key column. In our design, primary keys are used in all tables: MemberID, AddressID, ItemID, LoanID, FineID, and RepaymentID.

2. FOREIGN KEY: This constraint is used to maintain referential integrity between related tables. It ensures that a value in a column of one table must exist in the primary key column of another table. In our design, we used foreign keys to link the tables as follows:

- MemberID in Loans, OverdueFines, and Addresses tables referencing Members table
- ItemID in Loans table referencing Items table
- LoanID in OverdueFines table referencing Loans table
- FineID in FineRepayments table referencing OverdueFines table

3. UNIQUE: This constraint is used to ensure that all values in a column are distinct. In our design, we used the UNIQUE constraint on the Username column in the Members table to ensure that each member has a distinct username.

4. CHECK: This constraint is used to enforce specific rules or conditions on the data entered into a column. In our design, we used CHECK constraints for the following columns:

- ItemType in Items table: to ensure only valid item types ('Book', 'Journal', 'DVD', 'Other Media') are entered
- CurrentStatus in Items table: to ensure only valid status values ('On Loan', 'Overdue', 'Available', 'Lost/Removed') are entered
- RepaymentMethod in FineRepayments table: to ensure only valid repayment methods ('Cash', 'Card') are entered.

5. NOT NULL: This constraint is used to ensure that a column cannot have NULL values. In our design, we used the NOT NULL constraint on columns where a value is required for the proper functioning of the database, such as FirstName, LastName, Address1, DateOfBirth, Username, Password, Title, ItemType, Author, YearOfPublication, DateAdded, and CurrentStatus, among others.

These constraints help maintain data integrity by preventing the entry of incorrect, inconsistent, or duplicate data into the database.

3.5 DATABASE DIAGRAM

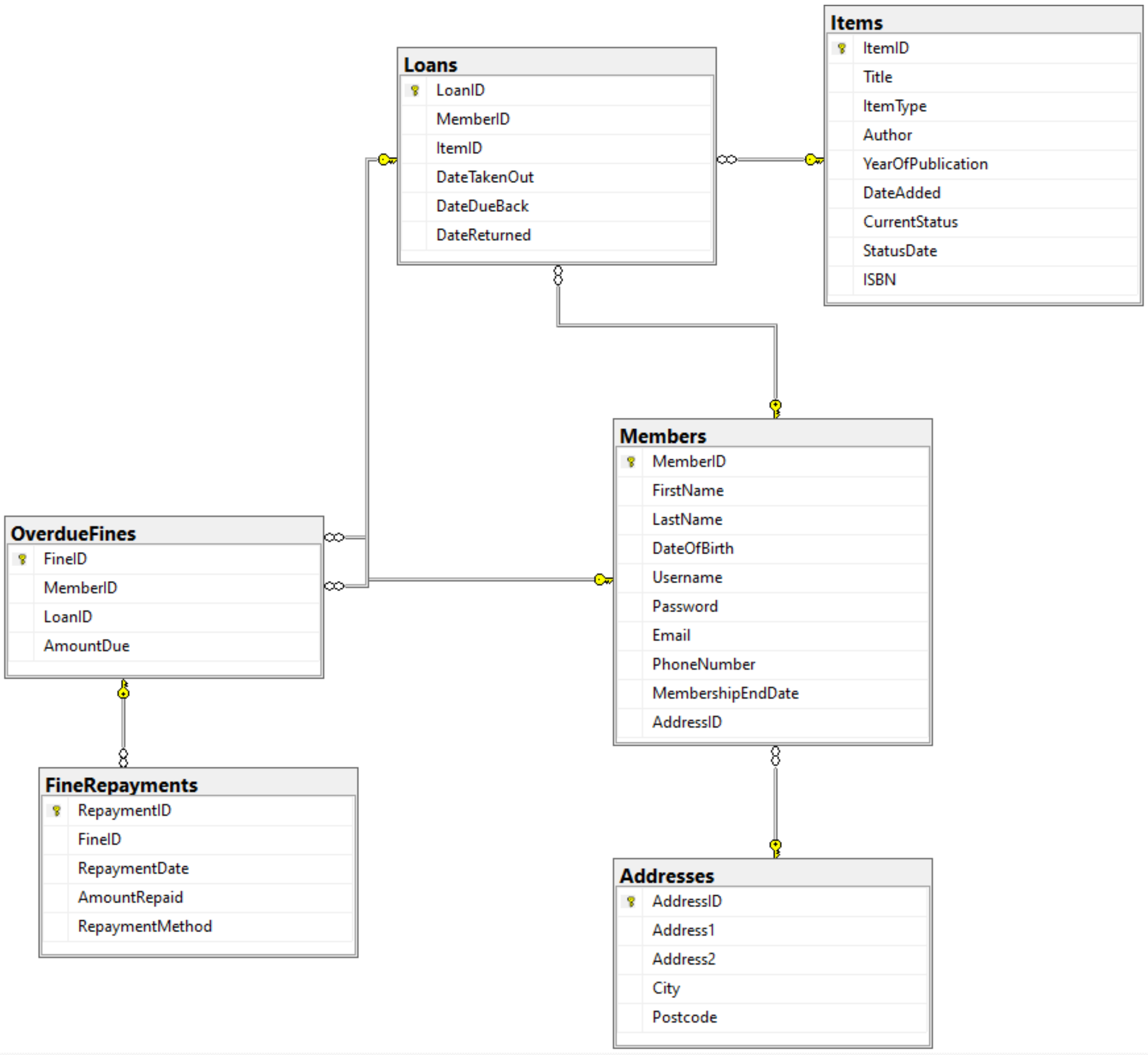


Fig 1: Relational Diagram of the Library Database

PART 2: VIEWS, STORED PROCEDURES OR USER DEFINED FUNCTION

Views, Stored Procedures, User-Defined Functions, and Triggers are essential database objects that provide different functionalities to enhance the overall performance, security, and maintainability of the database system.

1. Views:

- Purpose: Views are virtual tables based on the result set of a SELECT statement, providing a way to simplify complex queries, secure data access, or present data in a specific format.
- Implementation: In this project, we created a view to show the loan history, including details of the item borrowed, borrowed date, due date, and any associated fines for each loan.
- Benefits: Views can help simplify complex queries by encapsulating them in a single object, reduce data redundancy, and provide a level of data abstraction and security by limiting the data users can access.

2. Stored Procedures:

- Purpose: Stored procedures are precompiled SQL statements that can be executed to perform specific tasks or operations, such as data manipulation or data retrieval.
- Implementation: We created stored procedures to search the catalog, return a list of items due within five days, insert new members, and update existing member details.
- Benefits: Stored procedures enhance performance by reducing network traffic, providing a level of abstraction, and allowing code reusability. They also improve security by encapsulating the logic and preventing SQL injection attacks.

3. User-Defined Functions:

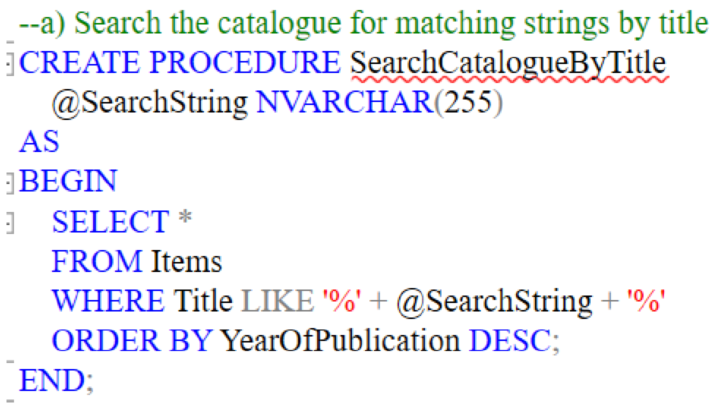
- Purpose: User-defined functions are custom functions created by the user to extend the functionality of the built-in system functions. They can be used to perform complex calculations, manipulate data, or return a table or scalar value.
- Implementation: We created a user-defined function to identify items due within five days and another to return the total number of loans made on a specified date.
- Benefits: User-defined functions allow code reusability, improve maintainability, and provide a level of abstraction by encapsulating complex logic or calculations.

4. Triggers:

- Purpose: Triggers are special types of stored procedures that automatically execute in response to specific events, such as data modification (INSERT, UPDATE, DELETE) or changes in the database schema (DDL triggers).
- Implementation: We created a trigger to update the status of an item to 'Available' when the book is returned.
- Benefits: Triggers help maintain data integrity and consistency by automatically enforcing business rules or constraints when specific events occur. They can also be used for auditing purposes, data validation, or maintaining referential integrity.

By using these database objects, we can improve the overall performance, security, and maintainability of the library's database system, ensuring that it meets the client's requirements and expectations.

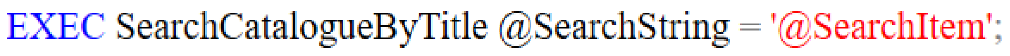
A. To search the catalogue for matching strings by title and sorting with the most recent publication date first, we created a stored procedure with the following T-SQL Statement.



```sql
--a) Search the catalogue for matching strings by title
CREATE PROCEDURE SearchCatalogueByTitle
    @SearchString NVARCHAR(255)
AS
BEGIN
    SELECT *
    FROM Items
    WHERE Title LIKE '%' + @SearchString + '%'
    ORDER BY YearOfPublication DESC;
END;
```

The SearchCatalogueByTitle is a stored procedure created to search the library's catalog for items based on a user-provided search string. The stored procedure takes an input parameter @SearchString and returns a result set containing all records from the Items table that match the search criteria.
When a user calls the SearchCatalogueByTitle stored procedure and provides a search string, the procedure searches the Items table for any titles containing the search string and returns a result set containing all matching records, ordered by their publication year in descending order.

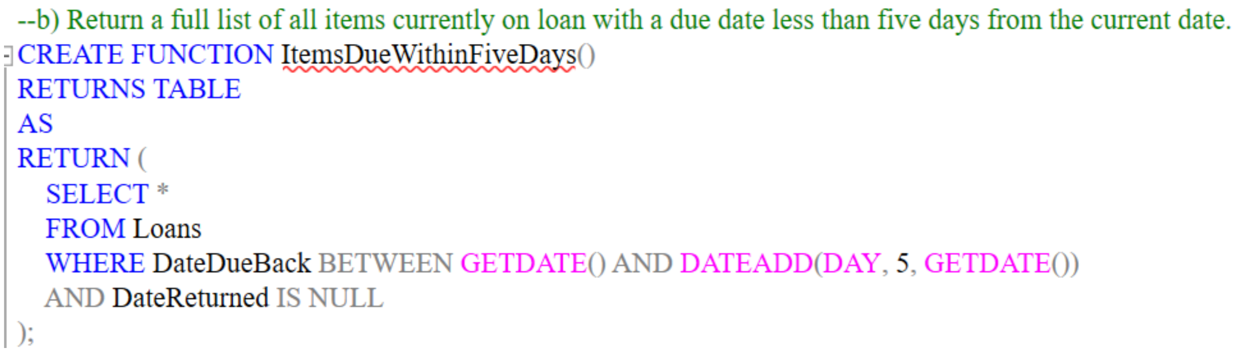
To execute the query we can use the following command



```sql
EXEC SearchCatalogueByTitle @SearchString = '@SearchItem';
```

Simply replace the placeholder values with the appropriate data in the usage example.

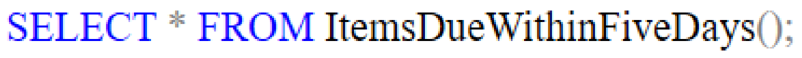
B. To return a full list of all items currently on loan which have a due date of less than five days from the current date (i.e., the system date when the query is run), we created a User-defined function using the following T-SQL statement



```
--b) Return a full list of all items currently on loan with a due date less than five days from the current date.
CREATE FUNCTION ItemsDueWithinFiveDays()
RETURNS TABLE
AS
RETURN (
    SELECT *
    FROM Loans
    WHERE DateDueBack BETWEEN GETDATE() AND DATEADD(DAY, 5, GETDATE())
    AND DateReturned IS NULL
);
```

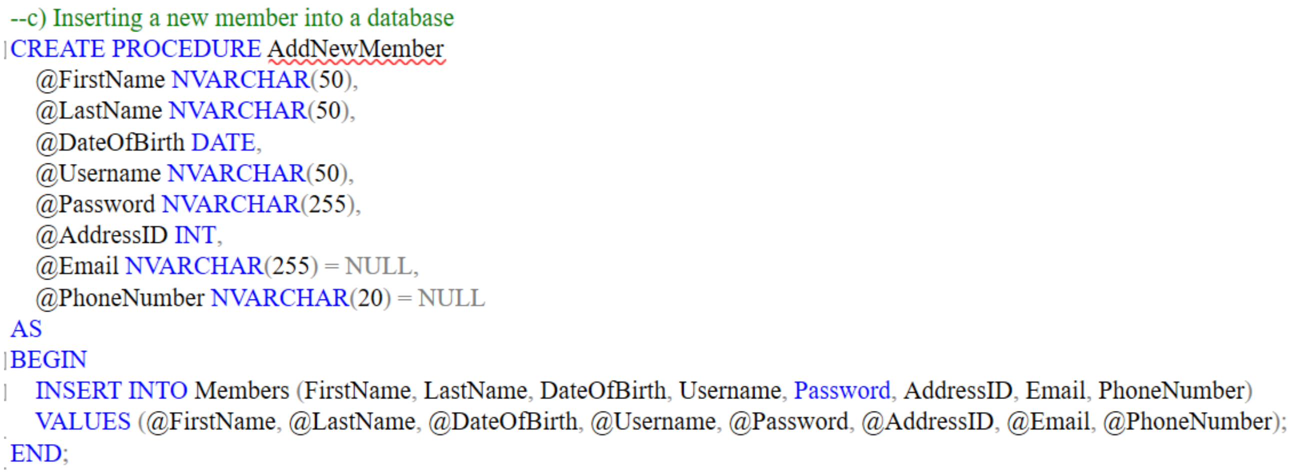
The ItemsDueWithinFiveDays is a user-defined table-valued function that returns a list of loaned items due within the next five days. It does this by querying the Loans table and filtering the records based on the DateDueBack column.
When a user calls the ItemsDueWithinFiveDays function, it returns a result set containing all records from the Loans table that meet the specified criteria: items due within the next five days and not yet returned.

To execute the query we can use the following command;



```
SELECT * FROM ItemsDueWithinFiveDays();
```

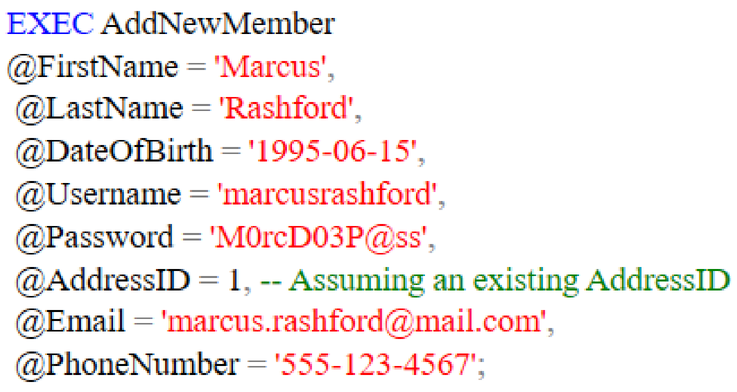
C. To insert a new member into the database, we created a stored procedure with the following T-SQL Statement.



```
--c) Inserting a new member into a database
CREATE PROCEDURE AddNewMember
    @FirstName NVARCHAR(50),
    @LastName NVARCHAR(50),
    @DateOfBirth DATE,
    @Username NVARCHAR(50),
    @Password NVARCHAR(255),
    @AddressID INT,
    @Email NVARCHAR(255) = NULL,
    @PhoneNumber NVARCHAR(20) = NULL
AS
BEGIN
    INSERT INTO Members (FirstName, LastName, DateOfBirth, Username, Password, AddressID, Email, PhoneNumber)
    VALUES (@FirstName, @LastName, @DateOfBirth, @Username, @Password, @AddressID, @Email, @PhoneNumber);
END;
```

The AddNewMember is a stored procedure that inserts a new member record into the Members table with the provided input parameters. It simplifies the process of adding new members and ensures consistent data insertion.
When a user calls the AddNewMember stored procedure and provides the required input parameters, a new member record is inserted into the Members table with the provided values. This stored procedure helps maintain data consistency and simplifies the process of adding new members to the library database.

To execute the query we can use the following sample command;



```
EXEC AddNewMember
@FirstName = 'Marcus',
 @LastName = 'Rashford',
 @DateOfBirth = '1995-06-15',
 @Username = 'marcusrashford',
 @Password = 'M0rcD03P@ss',
 @AddressID = 1, -- Assuming an existing AddressID
 @Email = 'marcus.rashford@mail.com',
 @PhoneNumber = '555-123-4567';
```

D. To update the details of an existing member, we created a stored procedure with the following T-SQL Statement.

```
CREATE PROCEDURE UpdateMemberDetails
    @MemberID INT,
    @FirstName NVARCHAR(50) = NULL,
    @LastName NVARCHAR(50) = NULL,
    @DateOfBirth DATE = NULL,
    @Username NVARCHAR(50) = NULL,
    @Password NVARCHAR(255) = NULL,
    @AddressID INT = NULL,
    @Email NVARCHAR(255) = NULL,
    @PhoneNumber NVARCHAR(20) = NULL
AS
BEGIN
    UPDATE Members
    SET FirstName = COALESCE(@FirstName, FirstName),
        LastName = COALESCE(@LastName, LastName),
        DateOfBirth = COALESCE(@DateOfBirth, DateOfBirth),
        Username = COALESCE(@Username, Username),
        Password = COALESCE(@Password, Password),
        AddressID = COALESCE(@AddressID, AddressID),
        Email = COALESCE(@Email, Email),
        PhoneNumber = COALESCE(@PhoneNumber, PhoneNumber)
    WHERE MemberID = @MemberID;
END;
```

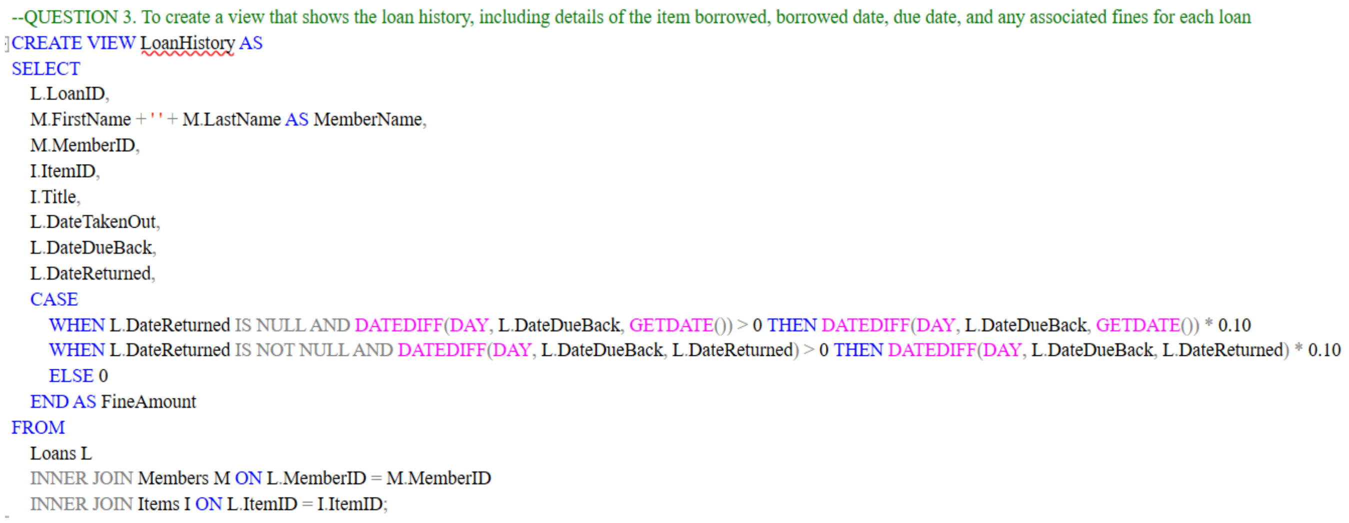
The UpdateMemberDetails stored procedure allows updating an existing member's details in the Member's table. It takes input parameters for the member's information, with default values of NULL. It updates only the columns where values are provided, leaving the rest unchanged.
The UpdateMemberDetails stored procedure allows updating specific member information in the Members table without altering the other columns, providing a flexible way to modify member records as needed.

To execute the query, we can use the following sample T-SQL statement;

```
--Update an existing member's email and phone number (MemberID = 1)
EXEC UpdateMemberDetails
@MemberID = 1,
@Email = 'marcus.rashford.updated@mail.com',
@PhoneNumber = '555-123-4567';
```

PART 3: LOAN HISTORY

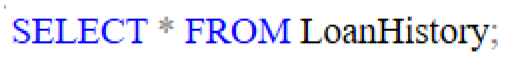
To view the loan history, showing all previous and current loans, and including details of the item borrowed, borrowed date, due date and any associated fines for each loan. You should create a view containing all the required information.



```
--QUESTION 3. To create a view that shows the loan history, 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.FirstName + ' ' + M.LastName AS MemberName,
    M.MemberID,
    I.ItemID,
    I.Title,
    L.DateTakenOut,
    L.DateDueBack,
    L.DateReturned,
    CASE
        WHEN L.DateReturned IS NULL AND DATEDIFF(DAY, L.DateDueBack, GETDATE()) > 0 THEN DATEDIFF(DAY, L.DateDueBack, GETDATE()) * 0.10
        WHEN L.DateReturned IS NOT NULL AND DATEDIFF(DAY, L.DateDueBack, L.DateReturned) > 0 THEN DATEDIFF(DAY, L.DateDueBack, L.DateReturned) * 0.10
        ELSE 0
    END AS FineAmount
FROM
    Loans L
    INNER JOIN Members M ON L.MemberID = M.MemberID
    INNER JOIN Items I ON L.ItemID = I.ItemID;
```

The LoanHistory view provides a summary of the loan history, including details about the member, the borrowed item, and any associated fines. It combines data from the Loans, Members, and Items tables using INNER JOINs. The view calculates fine amounts based on the difference between the due date and the returned date or the current date if the item is not yet returned. Fines are calculated at a rate of £0.10 per day overdue.

To execute the query, we use the following T-SQL statement;



```
SELECT * FROM LoanHistory;
```

This view shows the loan history with all previous and current loans, including details of the item borrowed, borrowed date, due date, and any associated overdue fees.

PART 4: TRIGGER

To create a trigger so that the current status of an item automatically updates to Available when the book is returned, we use the following T-SQL statement.

```
CREATE TRIGGER UpdateItemStatusOnReturn
ON Loans
AFTER UPDATE
AS
BEGIN
   IF UPDATE(DateReturned)
   BEGIN
      DECLARE @ItemID INT;
      SELECT @ItemID = ItemID FROM inserted;

      UPDATE Items
      SET CurrentStatus = 'Available'
      WHERE ItemID = @ItemID;
   END;
END;
```

The UpdateItemStatusOnReturn trigger is designed to automatically update the CurrentStatus of an item in the Items table to 'Available' when a book is returned. It's an AFTER UPDATE trigger that's created on the Loans table.

When an update occurs on the Loans table and the DateReturned column is affected, the trigger will execute. It declares a @ItemID variable and retrieves the corresponding ItemID from the inserted table, which contains the new data after the update operation. The trigger then updates the Items table, setting the CurrentStatus of the item with the specified ItemID to 'Available'. This ensures that the item's status is correctly updated when it's returned, reflecting its availability for future loans.

To execute the query, we use the following T-SQL statement;

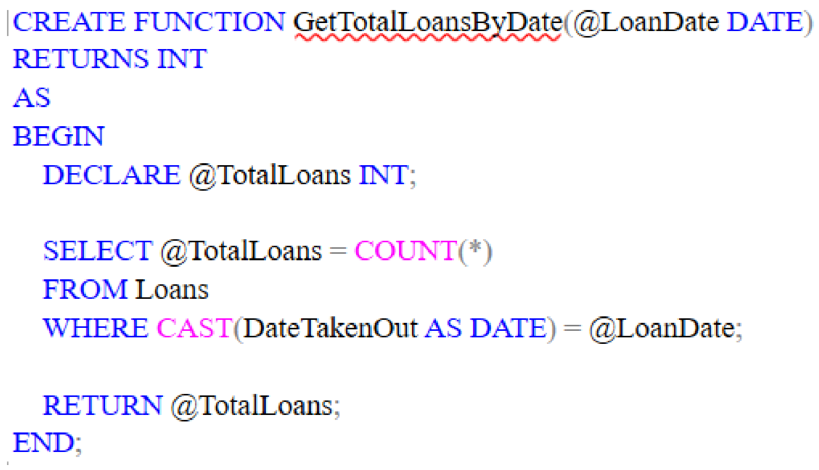
```
UPDATE Loans
SET DateReturned = GETDATE()
WHERE LoanID = <YourLoanID>;
```

We replace <YourLoanID> with the actual loan ID for which you want to update the DateReturned column.

PART 5: TOTAL NUMBER OF LOANS

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

To create a function that returns the total number of loans made on a specified date, we use the following T-SQL statement;



```sql
CREATE FUNCTION GetTotalLoansByDate(@LoanDate DATE)
RETURNS INT
AS
BEGIN
   DECLARE @TotalLoans INT;

   SELECT @TotalLoans = COUNT(*)
   FROM Loans
   WHERE CAST(DateTakenOut AS DATE) = @LoanDate;

   RETURN @TotalLoans;
END;
```

The GetTotalLoansByDate function is a user-defined scalar function that takes a date parameter @LoanDate and returns the total number of loans made on that specific date.

Within the function, a @TotalLoans variable is declared to store the count of loans. A SELECT query is used to count the number of rows in the Loans table where the DateTakenOut matches the provided @LoanDate. The CAST function is used to ensure that only the date part of the DateTakenOut is considered, ignoring any time components.

Finally, the function returns the calculated @TotalLoans value. This function can be useful for generating reports or analyzing loan activity on a specific date.

To test the function, we use the following T-SQL statement

```sql
SELECT dbo.GetTotalLoansByDate('YYYY-MM-DD') AS TotalLoans;
```

By replacing 'YYYY-MM-DD' with the actual date for which you want to get the total number of loans. The result will show the total loans made on the specified date.

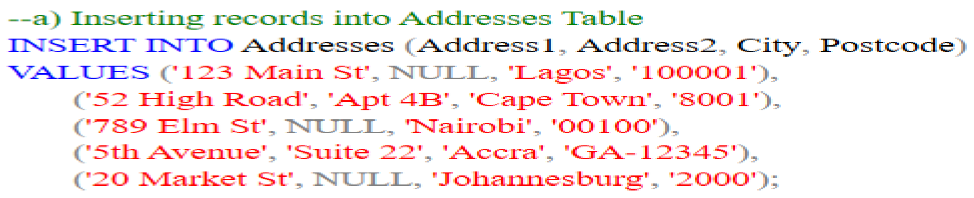
PART 6: INSERTING RECORDS INTO THE TABLE AND TESTING DATABASE OBJECTS

4.0. INSERTING RECORDS

I Inserted 5 sample fictitious data each of the 6 tables, which were adequate to test if all SELECT queries, user-defined functions, stored procedures, and triggers earlier created were working.

1. **Addresses Table**

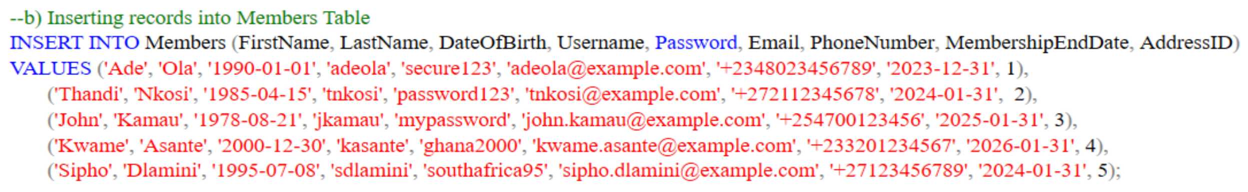
   Inserting sample records into the Addresses table was done using the T-SQL statement:



```sql
--a) Inserting records into Addresses Table
INSERT INTO Addresses (Address1, Address2, City, Postcode)
VALUES ('123 Main St', NULL, 'Lagos', '100001'),
       ('52 High Road', 'Apt 4B', 'Cape Town', '8001'),
       ('789 Elm St', NULL, 'Nairobi', '00100'),
       ('5th Avenue', 'Suite 22', 'Accra', 'GA-12345'),
       ('20 Market St', NULL, 'Johannesburg', '2000');
```

2. **Members Table**

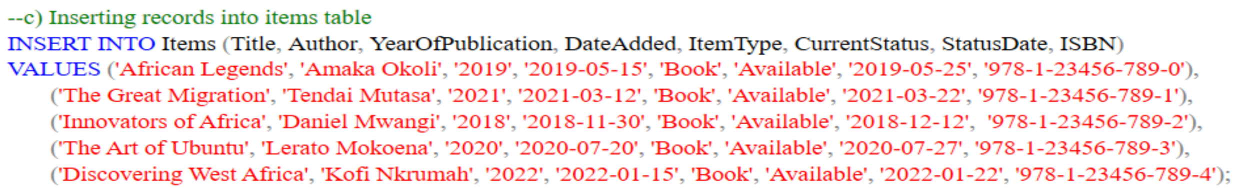
   Inserting sample records into the Members table was done using the T-SQL statement:



```sql
--b) Inserting records into Members Table
INSERT INTO Members (FirstName, LastName, DateOfBirth, Username, Password, Email, PhoneNumber, MembershipEndDate, AddressID)
VALUES ('Ade', 'Ola', '1990-01-01', 'adeola', 'secure123', 'adeola@example.com', '+2348023456789', '2023-12-31', 1),
      ('Thandi', 'Nkosi', '1985-04-15', 'tnkosi', 'password123', 'tnkosi@example.com', '+272112345678', '2024-01-31',  2),
      ('John', 'Kamau', '1978-08-21', 'jkamau', 'mypassword', 'john.kamau@example.com', '+254700123456', '2025-01-31', 3),
      ('Kwame', 'Asante', '2000-12-30', 'kasante', 'ghana2000', 'kwame.asante@example.com', '+233201234567', '2026-01-31', 4),
      ('Sipho', 'Dlamini', '1995-07-08', 'sdlamini', 'southafrica95', 'sipho.dlamini@example.com', '+27123456789', '2024-01-31', 5);
```

3. **Items Table**

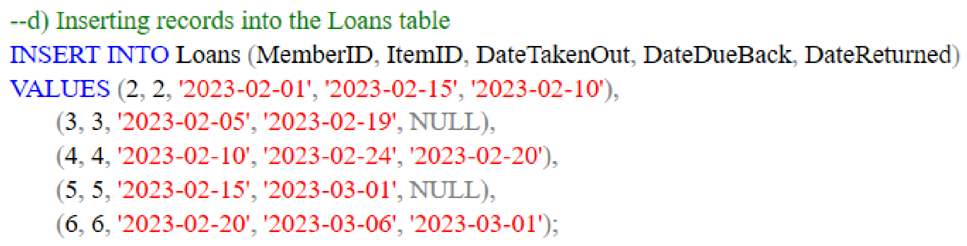
   Inserting sample records into the Items table was done using the T-SQL statement:



```
--c) Inserting records into items table
INSERT INTO Items (Title, Author, YearOfPublication, DateAdded, ItemType, CurrentStatus, StatusDate, ISBN)
VALUES ('African Legends', 'Amaka Okoli', '2019', '2019-05-15', 'Book', 'Available', '2019-05-25', '978-1-23456-789-0'),
      ('The Great Migration', 'Tendai Mutasa', '2021', '2021-03-12', 'Book', 'Available', '2021-03-22', '978-1-23456-789-1'),
      ('Innovators of Africa', 'Daniel Mwangi', '2018', '2018-11-30', 'Book', 'Available', '2018-12-12',  '978-1-23456-789-2'),
      ('The Art of Ubuntu', 'Lerato Mokoena', '2020', '2020-07-20', 'Book', 'Available', '2020-07-27', '978-1-23456-789-3'),
      ('Discovering West Africa', 'Kofi Nkrumah', '2022', '2022-01-15', 'Book', 'Available', '2022-01-22', '978-1-23456-789-4');
```

4. Loans Table

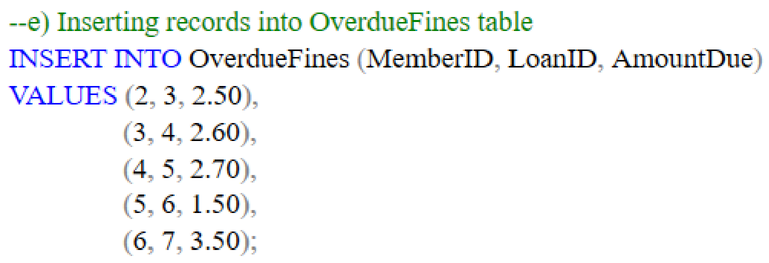
   Inserting sample records into the Loans table was done using the T-SQL statement:



```
--d) Inserting records into the Loans table
INSERT INTO Loans (MemberID, ItemID, DateTakenOut, DateDueBack, DateReturned)
VALUES (2, 2, '2023-02-01', '2023-02-15', '2023-02-10'),
      (3, 3, '2023-02-05', '2023-02-19', NULL),
      (4, 4, '2023-02-10', '2023-02-24', '2023-02-20'),
      (5, 5, '2023-02-15', '2023-03-01', NULL),
      (6, 6, '2023-02-20', '2023-03-06', '2023-03-01');
```

5. OverdueFines Table

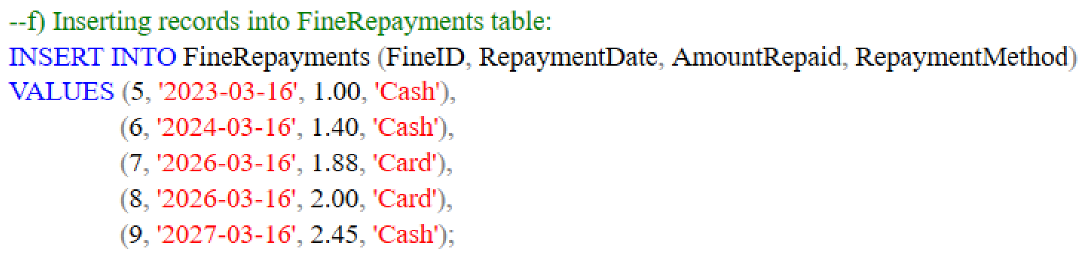
   Inserting sample records into the OverdueFines table was done using the T-SQL statement:



```
--e) Inserting records into OverdueFines table
INSERT INTO OverdueFines (MemberID, LoanID, AmountDue)
VALUES (2, 3, 2.50),
           (3, 4, 2.60),
           (4, 5, 2.70),
           (5, 6, 1.50),
           (6, 7, 3.50);
```

6. FineRepayments Table

   Inserting sample records into the FineRepayments table was done using the T-SQL statement:



```
--f) Inserting records into FineRepayments table:
INSERT INTO FineRepayments (FineID, RepaymentDate, AmountRepaid, RepaymentMethod)
VALUES (5, '2023-03-16', 1.00, 'Cash'),
           (6, '2024-03-16', 1.40, 'Cash'),
           (7, '2026-03-16', 1.88, 'Card'),
           (8, '2026-03-16', 2.00, 'Card'),
           (9, '2027-03-16', 2.45, 'Cash');
```

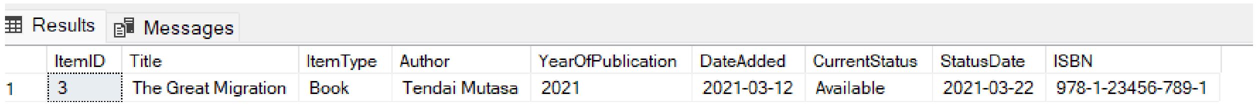
4.1 TESTING THE DATABASE OBJECTS

Now that we have inserted these records, we can test the SELECT queries, user-defined functions, stored procedures, and triggers by executing the following commands:

1. To test the stored procedure **'SearchCatalogueByTitle':**

```
--1. Search the catalogue for matching character strings by title:
EXEC SearchCatalogueByTitle @SearchString = 'Great';
```

Result: Once executed, this stored procedure will return a result set with items that have a title containing the word 'Great', sorted by their publication year in descending order.

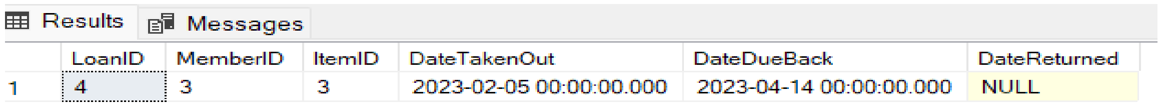


	ItemID	Title	ItemType	Author	YearOfPublication	DateAdded	CurrentStatus	StatusDate	ISBN
1	3	The Great Migration	Book	Tendai Mutasa	2021	2021-03-12	Available	2021-03-22	978-1-23456-789-1

2. To test the user-defined function **'ItemsDueWithinFiveDays':**

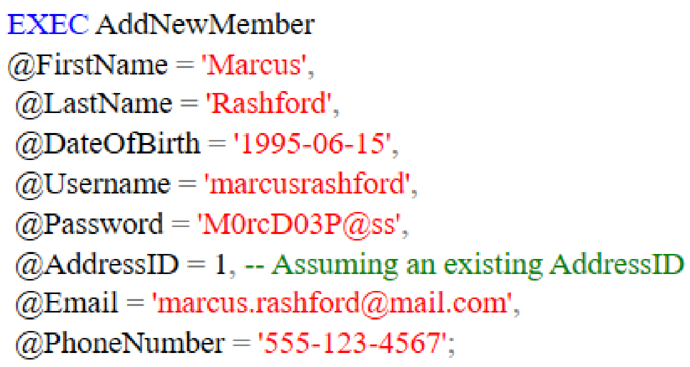
```
--2. Return a full list of all items currently on loan with a due date less than five days from the current date:
SELECT * FROM ItemsDueWithinFiveDays();
```

Result: By executing this SQL statement, you will receive a result set containing all the loan records for items that are due within the next five days and have not been returned yet.



	LoanID	MemberID	ItemID	DateTakenOut	DateDueBack	DateReturned
1	4	3	3	2023-02-05 00:00:00.000	2023-04-14 00:00:00.000	NULL

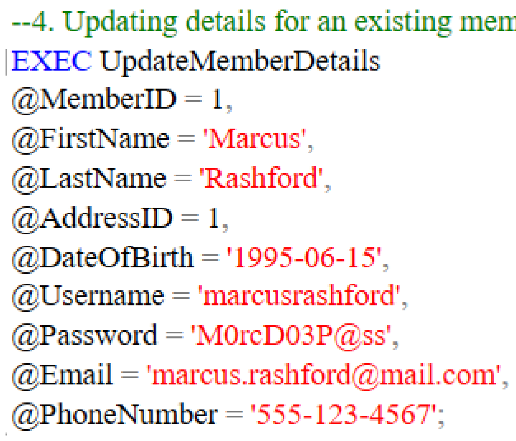
3. To test the stored procedure **'AddNewMember':**



```
EXEC AddNewMember
@FirstName = 'Marcus',
 @LastName = 'Rashford',
 @DateOfBirth = '1995-06-15',
 @Username = 'marcusrashford',
 @Password = 'M0rcD03P@ss',
 @AddressID = 1, -- Assuming an existing AddressID
 @Email = 'marcus.rashford@mail.com',
 @PhoneNumber = '555-123-4567';
```

Result: Upon executing this statement, a new member record will be inserted into the Members table with the provided information.

4. To test the stored procedure **'UpdateMemberDetails':**

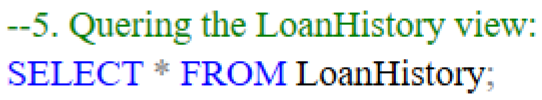


```
--4. Updating details for an existing member:
EXEC UpdateMemberDetails
@MemberID = 1,
@FirstName = 'Marcus',
@LastName = 'Rashford',
@AddressID = 1,
@DateOfBirth = '1995-06-15',
@Username = 'marcusrashford',
@Password = 'M0rcD03P@ss',
@Email = 'marcus.rashford@mail.com',
@PhoneNumber = '555-123-4567';
```

Result:

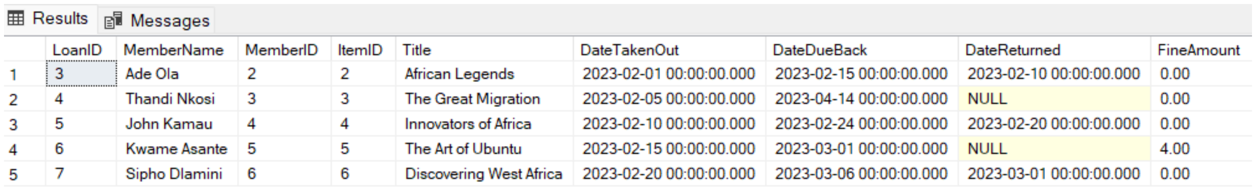
Upon executing this statement, the specified member record (with MemberID = 1) will be updated with the provided information. If any of the optional parameters are not provided, the existing values for those fields will be retained.

5. To test the view **'LoanHistory':**



```
--5. Quering the LoanHistory view:
SELECT * FROM LoanHistory;
```

Result: This query will return the entire loan history, allowing you to analyze the data and identify any overdue items, fines, and trends in borrowing. Recall that the fines are calculated at 10p per day for overdue items.



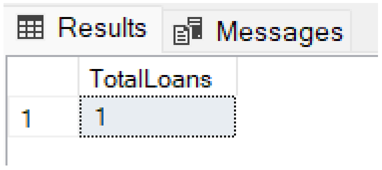
	LoanID	MemberName	MemberID	ItemID	Title	DateTakenOut	DateDueBack	DateReturned	FineAmount
1	3	Ade Ola	2	2	African Legends	2023-02-01 00:00:00.000	2023-02-15 00:00:00.000	2023-02-10 00:00:00.000	0.00
2	4	Thandi Nkosi	3	3	The Great Migration	2023-02-05 00:00:00.000	2023-04-14 00:00:00.000	NULL	0.00
3	5	John Kamau	4	4	Innovators of Africa	2023-02-10 00:00:00.000	2023-02-24 00:00:00.000	2023-02-20 00:00:00.000	0.00
4	6	Kwame Asante	5	5	The Art of Ubuntu	2023-02-15 00:00:00.000	2023-03-01 00:00:00.000	NULL	4.00
5	7	Sipho Dlamini	6	6	Discovering West Africa	2023-02-20 00:00:00.000	2023-03-06 00:00:00.000	2023-03-01 00:00:00.000	0.00

6. To test the function **'GetTotalLoansByDate'**:

```
--6. Using the GetTotalLoansByDate function:
SELECT dbo.GetTotalLoansByDate('2023-02-15') AS TotalLoans;
```

Result: By executing this statement, you'll get a result set with a single column and a single row:

TotalLoans: The total number of loans made on the specified date, '2023-02-15''. This query allows you to analyze the loan activity for a specific date and may help you identify trends or patterns in borrowing activity.



	TotalLoans
1	1

7. Updating a loan's DateReturned column and test the trigger that updates the item's status:

```
UPDATE Loans
SET DateReturned = GETDATE()
WHERE LoanID = 2;
SELECT * FROM Items WHERE ItemID = 5;
```

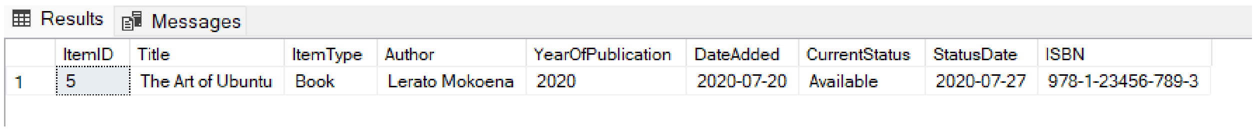
Result: This pair of SQL statements updates the Loans table and then queries the Items table to verify the changes.

UPDATE Loans SET DateReturned = GETDATE () WHERE LoanID = 2;:

This statement updates the DateReturned column in the Loans table for the loan with LoanID = 2. It sets the DateReturned value to the current date and time using the GETDATE () function, which means the item with LoanID = 2 is now marked as returned.

SELECT * FROM Items WHERE ItemID = 5;

This statement queries the Items table for the item with ItemID = 5. It returns all columns for the specified item, allowing you to verify if the CurrentStatus of the item has been updated to 'Available' after the loan was marked as returned.



	ItemID	Title	ItemType	Author	YearOfPublication	DateAdded	CurrentStatus	StatusDate	ISBN
1	5	The Art of Ubuntu	Book	Lerato Mokoena	2020	2020-07-20	Available	2020-07-27	978-1-23456-789-3

PART 7: ADDITIONAL DATABASE OBJECTS

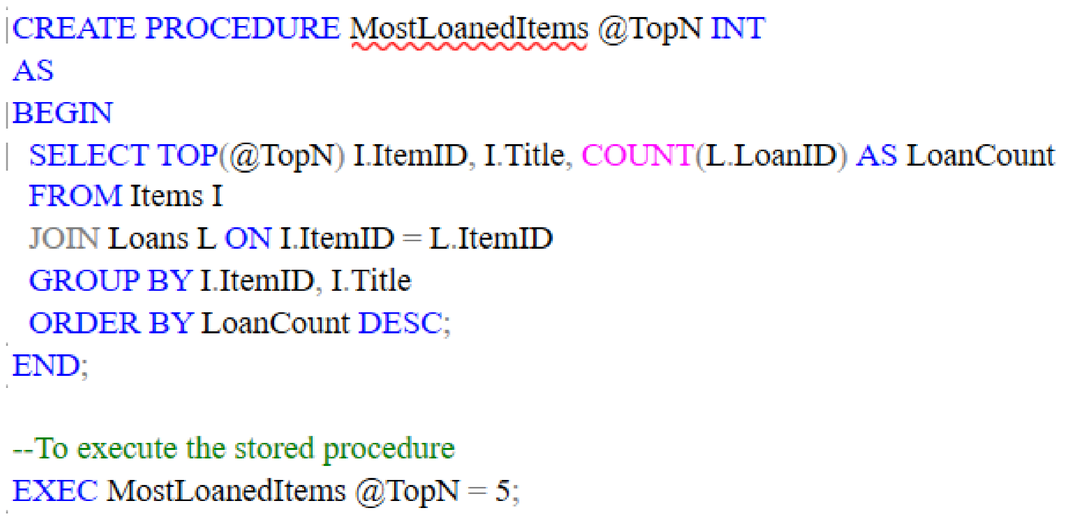
Below are some additional database objects that could be relevant and useful for the library management system.

1. To create a view for **'MembersWithOverdueitems'**:

```
CREATE VIEW MembersWithOverdueItems
AS
SELECT M.MemberID, M.FirstName + ' ' + M.LastName AS MemberName, I.ItemID, I.Title, L.DateDueBack, DATEDIFF(DAY, L.DateDueBack, GETDATE()) AS DaysOverdue
FROM Members M
JOIN Loans L ON M.MemberID = L.MemberID
JOIN Items I ON L.ItemID = I.ItemID
WHERE L.DateReturned IS NULL AND L.DateDueBack < GETDATE();
GO
--To query this view:
SELECT * FROM MembersWithOverdueItems;
GO
```

Function: This view lists members with overdue items, along with the item details and the number of days overdue.

2. To create a stored procedure for **'MostLoanedItems'**

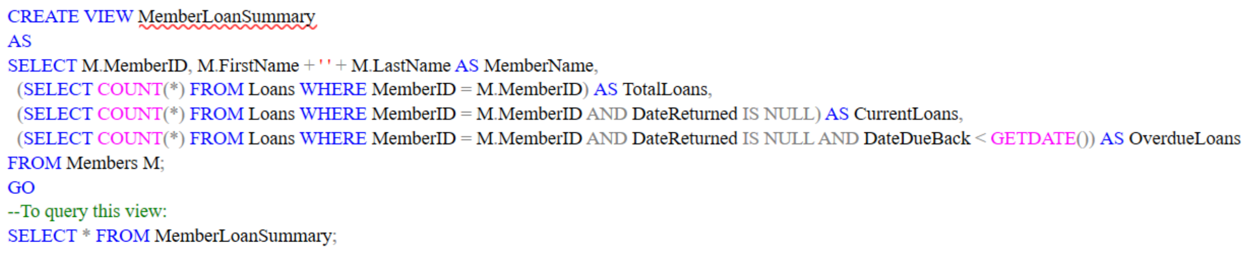


```sql
CREATE PROCEDURE MostLoanedItems @TopN INT
AS
BEGIN
  SELECT TOP(@TopN) I.ItemID, I.Title, COUNT(L.LoanID) AS LoanCount
  FROM Items I
  JOIN Loans L ON I.ItemID = L.ItemID
  GROUP BY I.ItemID, I.Title
  ORDER BY LoanCount DESC;
END;

--To execute the stored procedure
EXEC MostLoanedItems @TopN = 5;
```

Function: This stored procedure retrieves the top N most loaned items in the library.

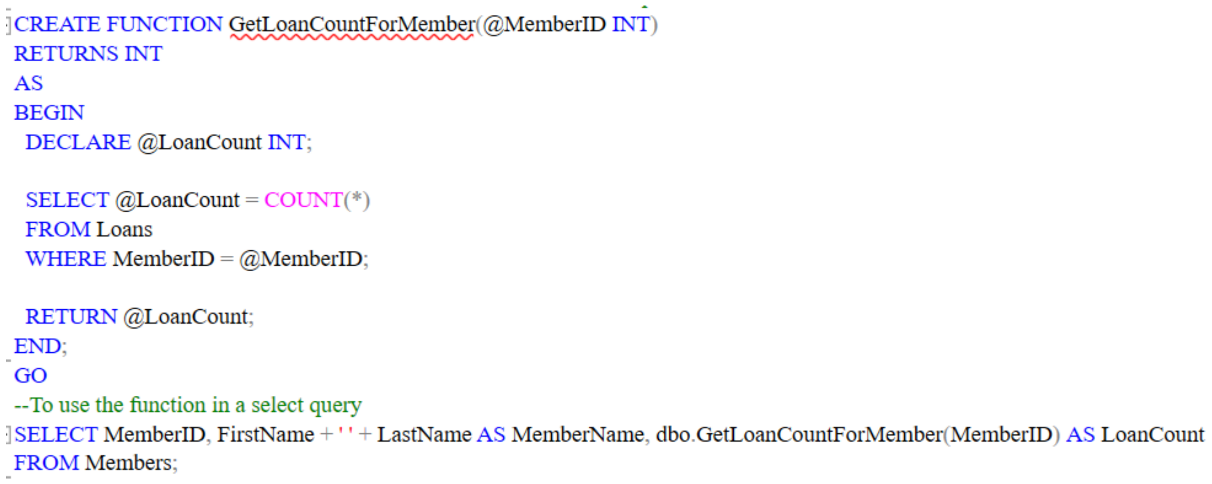
3. To create a view for the **'MemberLoanSummary'**



```sql
CREATE VIEW MemberLoanSummary
AS
SELECT M.MemberID, M.FirstName + ' ' + M.LastName AS MemberName,
  (SELECT COUNT(*) FROM Loans WHERE MemberID = M.MemberID) AS TotalLoans,
  (SELECT COUNT(*) FROM Loans WHERE MemberID = M.MemberID AND DateReturned IS NULL) AS CurrentLoans,
  (SELECT COUNT(*) FROM Loans WHERE MemberID = M.MemberID AND DateReturned IS NULL AND DateDueBack < GETDATE()) AS OverdueLoans
FROM Members M;
GO
--To query this view:
SELECT * FROM MemberLoanSummary;
```

Function: This view provides a summary of each member's loan activity, including the total loans, current loans, and overdue loans.

4. To create a User-defined function **'GetLoanCountForMember'**



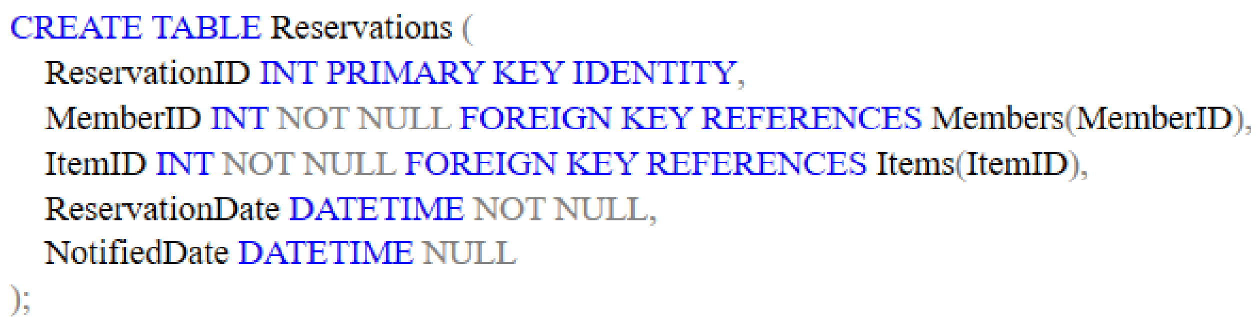
```sql
CREATE FUNCTION GetLoanCountForMember(@MemberID INT)
RETURNS INT
AS
BEGIN
  DECLARE @LoanCount INT;

  SELECT @LoanCount = COUNT(*)
  FROM Loans
  WHERE MemberID = @MemberID;

  RETURN @LoanCount;
END;
GO
--To use the function in a select query
SELECT MemberID, FirstName + ' ' + LastName AS MemberName, dbo.GetLoanCountForMember(MemberID) AS LoanCount
FROM Members;
```

Function: This user-defined function returns the total number of loans for a specific member.

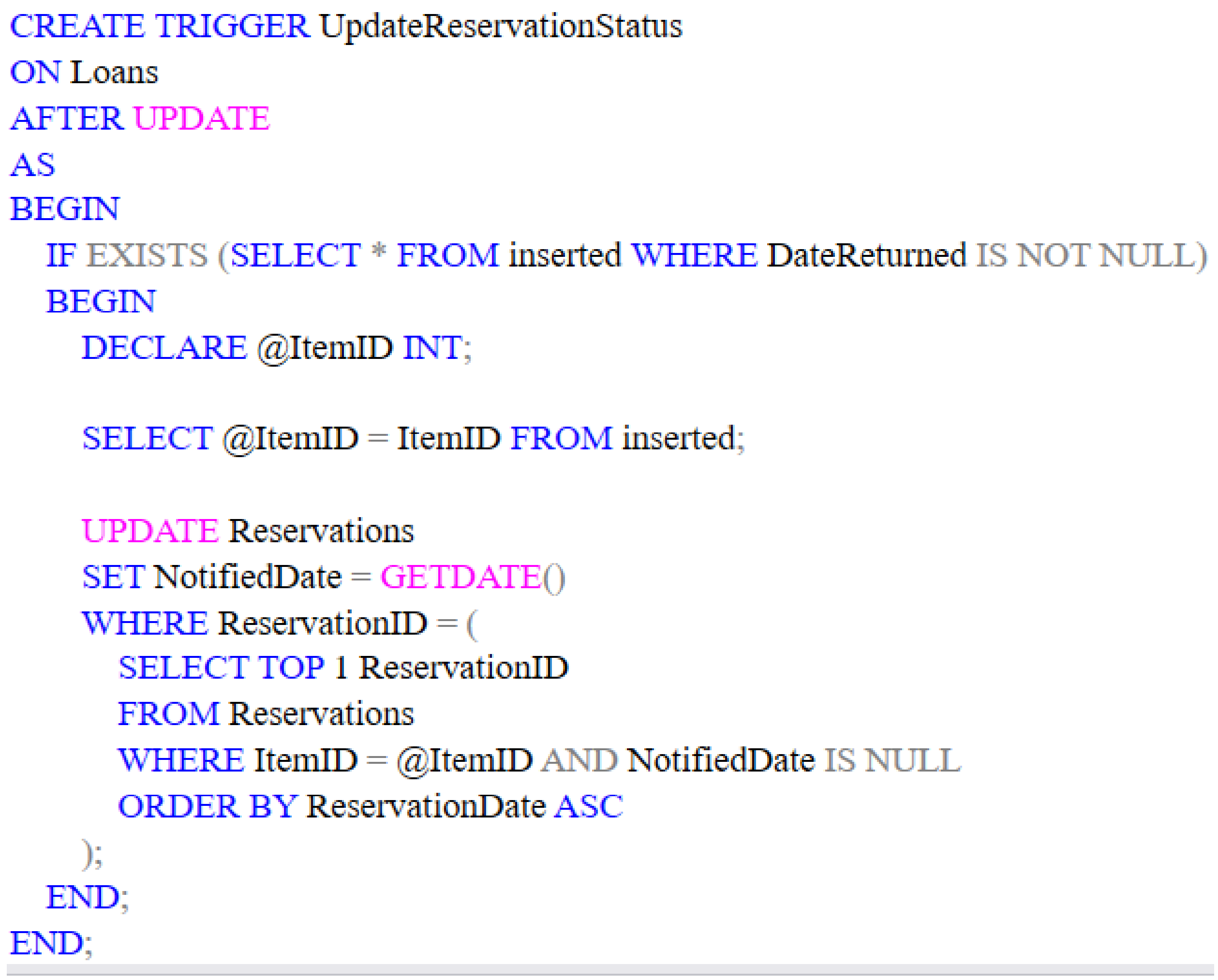
5. Creating a Reservations table



```sql
CREATE TABLE Reservations (
    ReservationID INT PRIMARY KEY IDENTITY,
    MemberID INT NOT NULL FOREIGN KEY REFERENCES Members(MemberID),
    ItemID INT NOT NULL FOREIGN KEY REFERENCES Items(ItemID),
    ReservationDate DATETIME NOT NULL,
    NotifiedDate DATETIME NULL
);
```

Function: A table to manage item reservations by members. When an item is not available for loan, members can reserve it, and when the item becomes available, the member is notified.

6. To create a Trigger to **'UpdateReservationStatus':**



```sql
CREATE TRIGGER UpdateReservationStatus
ON Loans
AFTER UPDATE
AS
BEGIN
   IF EXISTS (SELECT * FROM inserted WHERE DateReturned IS NOT NULL)
   BEGIN
      DECLARE @ItemID INT;

      SELECT @ItemID = ItemID FROM inserted;

      UPDATE Reservations
      SET NotifiedDate = GETDATE()
      WHERE ReservationID = (
         SELECT TOP 1 ReservationID
         FROM Reservations
         WHERE ItemID = @ItemID AND NotifiedDate IS NULL
         ORDER BY ReservationDate ASC
      );
   END;
END;
```

Function: This trigger automatically updates the reservation status when an item is returned, notifying the member with the earliest reservation.

5.0 RECOMMENDATIONS

Data Integrity and Concurrency:

Use transactions to maintain data integrity and Isolation levels to manage concurrency. Transactions ensure that all changes are either committed together or rolled back. For example, when updating the loans and members' tables, wrap the operations in a transaction to ensure both updates succeed or fail together. SQL Server supports different isolation levels that determine how transactions interact. The appropriate isolation level should be selected based on the needs of the library system to balance data consistency and performance. For instance, using the READ COMMITTED isolation level ensures that only committed data is read, preventing issues like dirty reads.

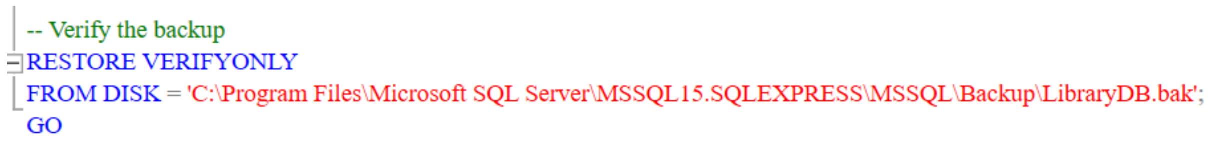
Database Security:

Restrict user access with roles, use parameterized queries/stored procedures to prevent SQL injection, and encrypt sensitive data.Create different user roles with specific access rights and assign them to library staff based on their responsibilities. For example, a librarian role may have read and write access to the members and loans tables, while an administrator role may have additional access to manage database objects and security settings. Use of parameterized queries and stored procedures ensures that user inputs are treated as data, not as part of the SQL code, reducing the risk of unauthorized code execution.Encrypt sensitive data, such as passwords and personal information, using appropriate encryption algorithms. SQL Server supports various encryption methods. Choose the right encryption method based on the library's requirements and regulatory compliance needs.

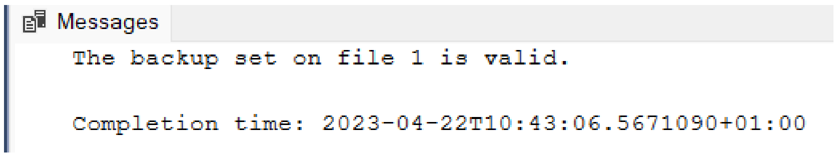
Database Backup and Recovery:

Develop a backup strategy with full, differential, and log backups. Regularly test backup/recovery procedures and implement a disaster recovery plan.Develop a backup strategy that includes full, differential, and log backups. Full backups contain all the data in the database, while differential backups store changes since the last full backup, and log backups store transaction log records. Schedule backups based on the library's data recovery requirements.

Regularly test the backup and recovery procedures to ensure that they work as expected. This can help identify any issues and minimize downtime in case of a database failure. Create a disaster recovery plan that outlines the steps to be taken in the event of a database failure, including how to restore backups, recover data, and resume operations. The plan should also address hardware failures, power outages, and other potential issues that may impact the database system.



```
-- Verify the backup
RESTORE VERIFYONLY
FROM DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\Backup\LibraryDB.bak';
GO
```



Messages

```
The backup set on file 1 is valid.

Completion time: 2023-04-22T10:43:06.5671090+01:00
```

6. 0 CONCLUSIONS

In conclusion, the database design solution presented for the library management system is a comprehensive and efficient approach to managing member information, catalog, loan history, and overdue fine repayments. The design follows the principles of normalization up to the third normal form (3NF), ensuring data integrity and reducing redundancy.
The key functionality provided includes the ability to:

- Manage member records, allowing for the addition, updating, and retrieval of member information.
- Maintain a catalog of items, with the flexibility to store and search for item details such as title, author, and publication year.
- Track loans and their status, ensuring that due dates, return dates, and overdue fines are managed effectively.
- Calculate and manage overdue fines, providing insights into outstanding fines and facilitating the fine repayment process.
- Implement views, stored procedures, user-defined functions, and triggers to optimize performance, simplify complex queries, and automate specific tasks.
- The solution also emphasizes data integrity, concurrency, database security, and backup and recovery strategies, ensuring that the library's data remains consistent, secure, and readily available in case of emergencies.

Overall, the proposed database system will help the library streamline its operations, improve the management of its resources, and enhance the experience for both staff and members.

TASK 2: ANALYSIS OF PRESCRIPTION DATA IN BOLTON: CREATING AND IMPLEMENTING THE PRESCRIPTIONS DATABASE

1. INTRODUCTION

The healthcare industry is constantly evolving, with an increasing reliance on technology to manage and streamline various aspects of patient care. One critical area where technology has made significant advancements is in the management of patient prescriptions. Efficient and accurate prescription management is crucial to ensure patients receive the right medications at the right time, and to reduce the risk of medication errors and potential adverse reactions.

The task at hand is to create a database named PrescriptionsDB, import three CSV files containing prescription data, and design the necessary primary and foreign key constraints to establish relationships between the tables. The three tables include Medical_Practice, Drugs, and Prescriptions, with unique identifiers for each practice, drug, and prescription, respectively.

As a database consultant for a pharmaceutical company, the goal is to analyse the prescribing data to gain insights into the types of medication prescribed, the organisations that are prescribing them, and the quantities prescribed.

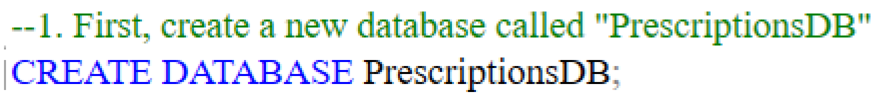
In this report, I will detail the steps taken to create the PrescriptionsDB, import the CSV files, and set up the necessary primary and foreign key constraints. I will also provide a detailed explanation of each T-SQL statement and include screenshots of the process where necessary. Finally, I will provide a brief overview of the database design solution and the key functionality it provides.

PART 1: DESIGN AND IMPLEMENTATION OF A PRESCRIPTIONS DATABASE

1.0 CREATING THE DATABASE

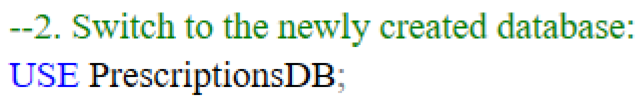
The first step would be to create the database on Microsoft SQL Server Management Studio based on the information provided. As requested, the database was created with the title ‘PrescriptionsDB.

The below T-SQL statements were used in creating the database.



```
--1. First, create a new database called "PrescriptionsDB"
CREATE DATABASE PrescriptionsDB;
```

We then switched to the newly created database using the below T-SQL statement.



```
--2. Switch to the newly created database:
USE PrescriptionsDB;
```

1.1 CREATING THE TABLES

Three CSV files containing prescription data namely Medical_Practice, Drugs and Prescriptions were provided.

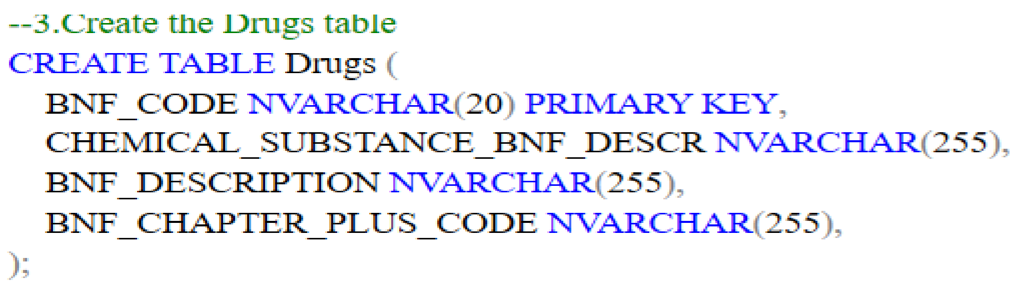
- The Medical_Practice.csv file has 60 records and provides the names and addresses of the medical practices which have prescribed medication within Bolton. The PRACTICE_CODE column provides a unique identifier for each practice.
- The Drugs.csv file provides details of the different drugs that can be prescribed. This includes the chemical substance, and the product description. The BNF_CHAPTER_PLUS_CODE column provides a way of categorizing the drugs based on the British National Formulatory (BNF) Chapter that includes the

prescribed product. For example, an antibiotic such as Amoxicillin is categorised under ‘05: Infections’. The BNF_CODE column provides a unique identifier for each drug.

- The Prescriptions.csv file provides a breakdown of each prescription. Each row corresponds to an individual prescription, and each prescription is linked to a practice via the PRACTICE_CODE and the drug via the BNF_CODE. It also specifies the quantity (the number of items in a pack) and the items (the number of packs). The PRESCRIPTION_CODE column provides a unique identifier for each prescription.

I created the tables using the following T-SQL Statements.

1. Drugs Table:



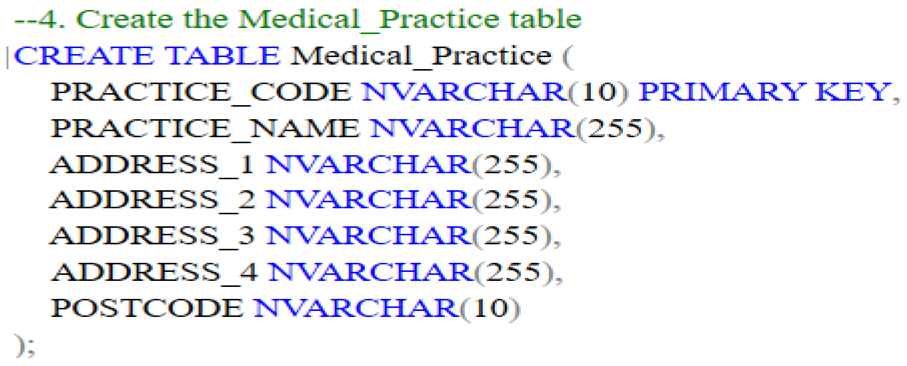
```
--3.Create the Drugs table
CREATE TABLE Drugs (
   BNF_CODE NVARCHAR(20) PRIMARY KEY,
   CHEMICAL_SUBSTANCE_BNF_DESCR NVARCHAR(255),
   BNF_DESCRIPTION NVARCHAR(255),
   BNF_CHAPTER_PLUS_CODE NVARCHAR(255),
);
```

The table has four columns with the following names and data types:

- BNF_CODE: NVARCHAR (20) - This column will store the unique identifier for each drug using a string data type that can store up to 20 characters. It is set as the primary key for the table, meaning that it will uniquely identify each row in the table.
- CHEMICAL_SUBSTANCE_BNF_DESCR: NVARCHAR (255) - This column will store the chemical substance of the drug using a string data type that can store up to 255 characters.

- BNF_DESCRIPTION: NVARCHAR (255) - This column will store the product description of the drug using a string data type that can store up to 255 characters.
- BNF_CHAPTER_PLUS_CODE: NVARCHAR (255) - This column will categorize the drugs based on the British National Formulatory (BNF) Chapter that includes the prescribed product. It will also use a string data type that can store up to 255 characters.

2. Medical Practice Table

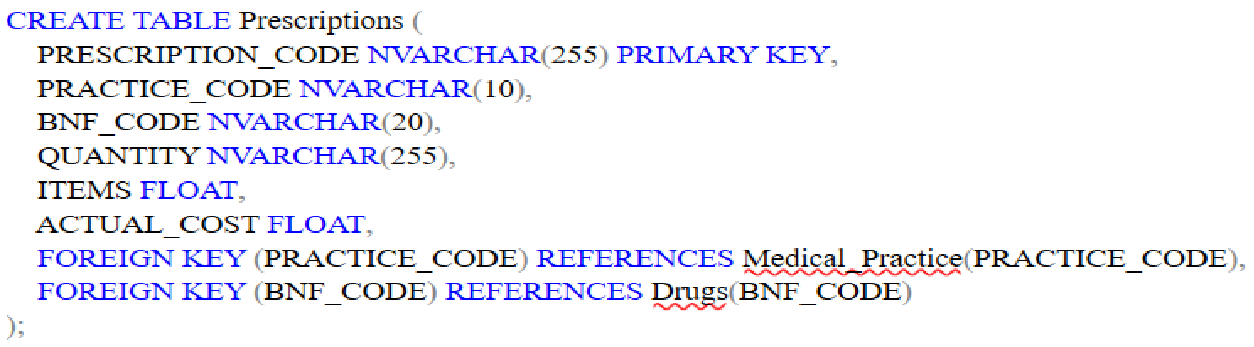


```sql
--4. Create the Medical_Practice table
CREATE TABLE Medical_Practice (
    PRACTICE_CODE NVARCHAR(10) PRIMARY KEY,
    PRACTICE_NAME NVARCHAR(255),
    ADDRESS_1 NVARCHAR(255),
    ADDRESS_2 NVARCHAR(255),
    ADDRESS_3 NVARCHAR(255),
    ADDRESS_4 NVARCHAR(255),
    POSTCODE NVARCHAR(10)
);
```

The data types used in the table are NVARCHAR, which is used for storing variable-length Unicode character string data, and is commonly used for storing text data that may contain non-English characters.

- The "PRACTICE_CODE" column is the primary key for this table, and it is defined as NVARCHAR (10), which means it can store up to 10 characters. This column will be used to uniquely identify each medical practice in the table.
- The "PRACTICE_NAME" column is defined as NVARCHAR (255) and will store the name of each medical practice.
- The "ADDRESS_1", "ADDRESS_2", "ADDRESS_3", and "ADDRESS_4" columns are defined as NVARCHAR (255) and will store the address of each medical practice.
- Finally, the "POSTCODE" column is defined as NVARCHAR (10) and will store the postal code of each medical practice.

3. The Prescriptions Table



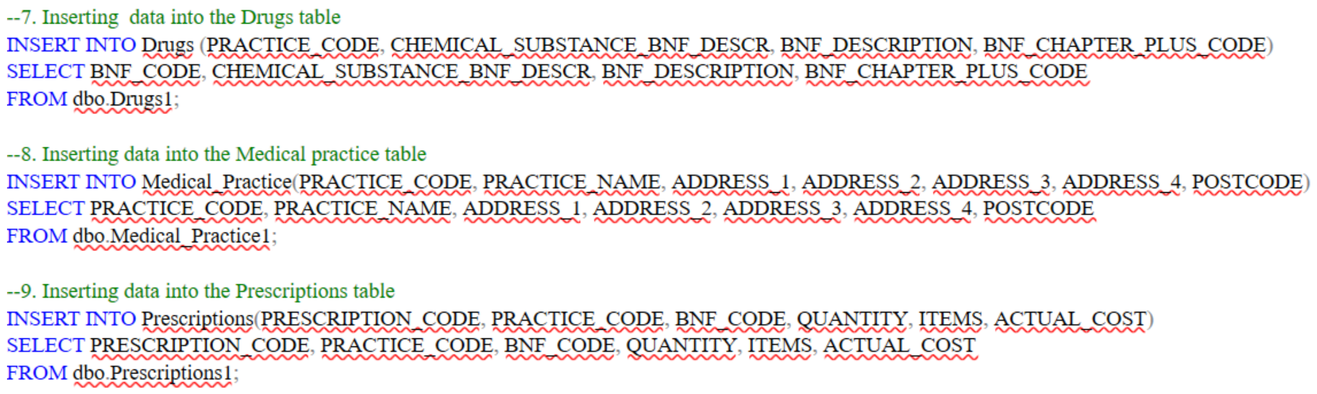
```
CREATE TABLE Prescriptions (
   PRESCRIPTION_CODE NVARCHAR(255) PRIMARY KEY,
   PRACTICE_CODE NVARCHAR(10),
   BNF_CODE NVARCHAR(20),
   QUANTITY NVARCHAR(255),
   ITEMS FLOAT,
   ACTUAL_COST FLOAT,
   FOREIGN KEY (PRACTICE_CODE) REFERENCES Medical_Practice(PRACTICE_CODE),
   FOREIGN KEY (BNF_CODE) REFERENCES Drugs(BNF_CODE)
);
```

- The "PRESCRIPTION_CODE" column is set as the primary key, which means it uniquely identifies each record in the table.
- The "PRACTICE_CODE" column is a foreign key that references the "PRACTICE_CODE" column of the "Medical_Practice" table. This means that it links each prescription to the medical practice that issued it.
- The "BNF_CODE" column is also a foreign key that references the "BNF_CODE" column of the "Drugs" table. This means that it links each prescription to the specific drug that was prescribed.
- The "QUANTITY" column is of data type NVARCHAR (255), which means it can store a string of up to 255 characters. This column stores the quantity of the prescribed drug.
- The "ITEMS" column is of data type FLOAT, which means it can store a floating-point number. This column stores the number of items prescribed.
- The "ACTUAL_COST" column is also of data type FLOAT, which means it can store a floating-point number. This column stores the actual cost of the prescription.

1.2 IMPORTING THE DATA

The 3 CSV files were renamed to prevent a conflict with the created tables when importing. The files were then imported using the SSMS interface and the appropriate data types were selected.

The data was then inserted into the created tables using the following T-SQL Statements.



```
--7. Inserting  data into the Drugs table
INSERT INTO Drugs (PRACTICE_CODE, CHEMICAL_SUBSTANCE_BNF_DESCR, BNF_DESCRIPTION, BNF_CHAPTER_PLUS_CODE)
SELECT BNF_CODE, CHEMICAL_SUBSTANCE_BNF_DESCR, BNF_DESCRIPTION, BNF_CHAPTER_PLUS_CODE
FROM dbo.Drugs1;

--8. Inserting data into the Medical practice table
INSERT INTO Medical_Practice(PRACTICE_CODE, PRACTICE_NAME, ADDRESS_1, ADDRESS_2, ADDRESS_3, ADDRESS_4, POSTCODE)
SELECT PRACTICE_CODE, PRACTICE_NAME, ADDRESS_1, ADDRESS_2, ADDRESS_3, ADDRESS_4, POSTCODE
FROM dbo.Medical_Practice1;

--9. Inserting data into the Prescriptions table
INSERT INTO Prescriptions(PRESCRIPTION_CODE, PRACTICE_CODE, BNF_CODE, QUANTITY, ITEMS, ACTUAL_COST)
SELECT PRESCRIPTION_CODE, PRACTICE_CODE, BNF_CODE, QUANTITY, ITEMS, ACTUAL_COST
FROM dbo.Prescriptions1;
```

1.3 CONSTRAINTS AND RELATIONSHIPS

The tables in the PrescriptionsDB database are linked through foreign key relationships. These relationships establish a connection between two tables by referencing a column in one table (the foreign key) to a primary key column in another table. This enforces data integrity by ensuring that the data in the foreign key column must have a corresponding value in the primary key column of the referenced table.

In this case, the Prescriptions table has two foreign key relationships:

- PRACTICE_CODE: This column in the Prescriptions table is a foreign key that references the PRACTICE_CODE column in the Medical_Practice table. This relationship links a prescription to the medical practice that issued it. The primary key in the Medical_Practice table is PRACTICE_CODE, which uniquely identifies each medical practice.
- BNF_CODE: This column in the Prescriptions table is a foreign key that references the BNF_CODE column in the Drugs table. This relationship links a prescription to the specific drug prescribed. The primary key in the Drugs table is BNF_CODE, which uniquely identifies each drug.

These foreign key relationships create a connection between the tables, allowing you to perform complex queries that involve data from multiple tables. In the database diagram, these relationships are typically represented as lines connecting the foreign key columns to their corresponding primary key columns in the referenced tables.

The relationships between the tables in the PrescriptionsDB database are as follows:

- Medical_Practice to Prescriptions:

This is a one-to-many relationship. A single medical practice (represented by a unique PRACTICE_CODE in the Medical_Practice table) can issue multiple prescriptions (rows in the Prescriptions table with only one medical practice.

- Drugs to Prescriptions:

This is also a one-to-many relationship. A single drug (represented by a unique BNF_CODE in the Drugs table) can be prescribed multiple times (rows in the Prescriptions table with the same BNF_CODE). However, each prescription is for a single drug.

In summary;

Medical_Practice (PRACTICE_CODE) -- (one-to-many) --> Prescriptions (PRACTICE_CODE)

Drugs (BNF_CODE) -- (one-to-many) --> Prescriptions (BNF_CODE).

1.4 DATABASE DIAGRAM

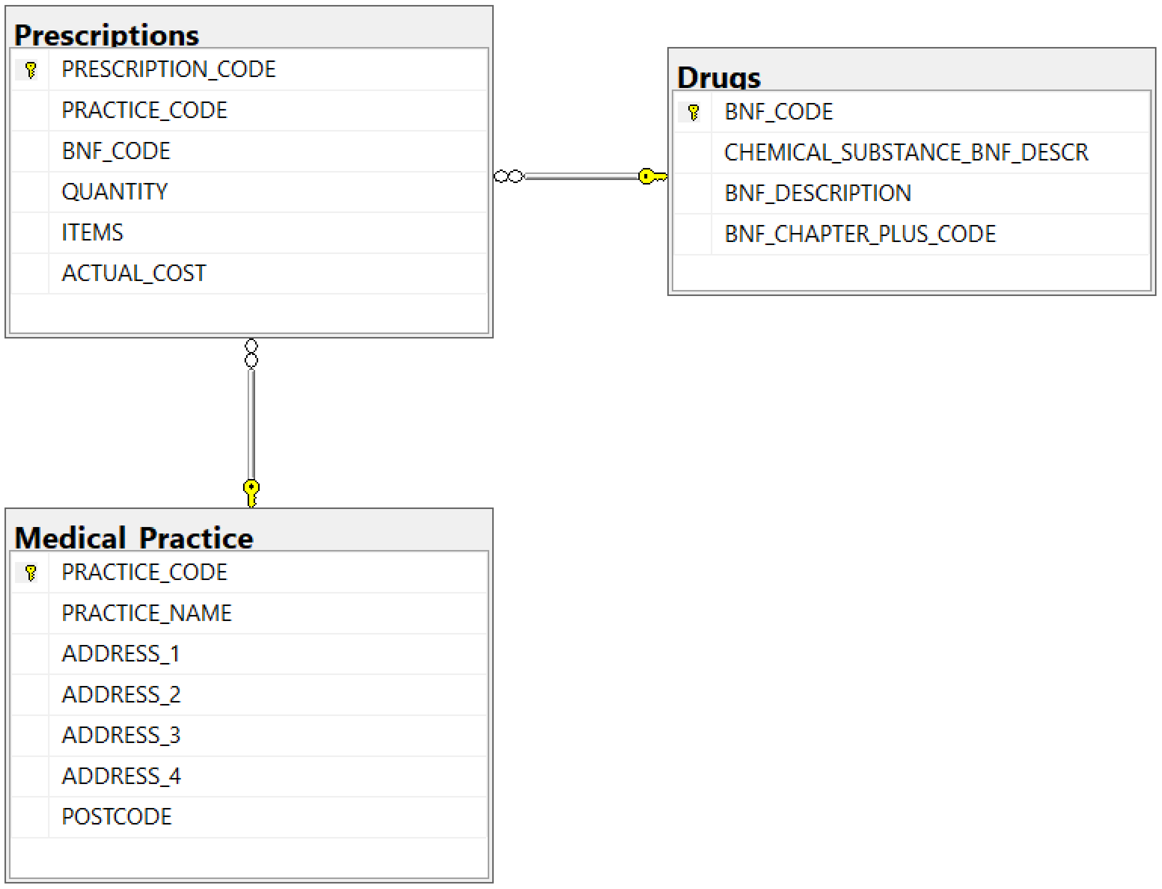


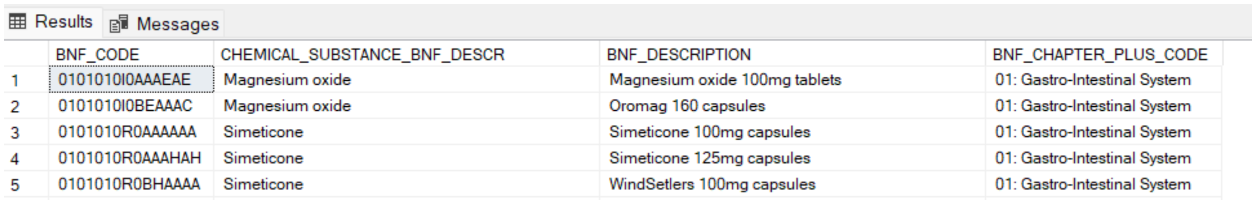
Fig 1: Relational Diagram of the Prescriptions Database

PART 2: DETAILS OF DRUGS IN TABLETS OR CAPSULES

To Return Details Of All Drugs Which Are In The Form Of Tablets Or Capsules, You Can Use The Following Query:

```
SELECT *
FROM Drugs
WHERE BNF_DESCRIPTION LIKE '%tablet%' OR BNF_DESCRIPTION LIKE '%capsule%';
```

This query searches for rows in the Drugs table where the BNF_DESCRIPTION contains either the word "tablet" or "capsule". The % symbol acts as a wildcard, allowing for any number of characters before or after the specified word. Below is a screenshot of the first 5 rows.



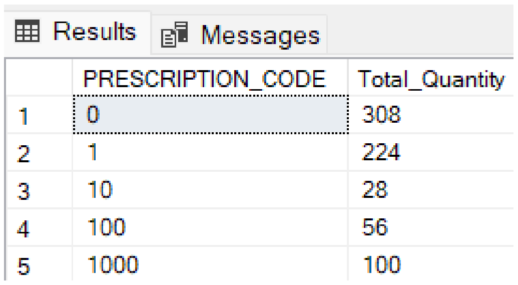
	BNF_CODE	CHEMICAL_SUBSTANCE_BNF_DESCR	BNF_DESCRIPTION	BNF_CHAPTER_PLUS_CODE
1	0101010I0AAAEAE	Magnesium oxide	Magnesium oxide 100mg tablets	01: Gastro-Intestinal System
2	0101010I0BEAAAC	Magnesium oxide	Oromag 160 capsules	01: Gastro-Intestinal System
3	0101010R0AAAAAA	Simeticone	Simeticone 100mg capsules	01: Gastro-Intestinal System
4	0101010R0AAAHAH	Simeticone	Simeticone 125mg capsules	01: Gastro-Intestinal System
5	0101010R0BHAAAA	Simeticone	WindSetlers 100mg capsules	01: Gastro-Intestinal System

PART 3: TOTAL QUANTITY OF EACH PRESCRIPTION

To Calculate The Total Quantity For Each Prescription By Multiplying The Number Of Items By The Quantity And Rounding The Result To The Nearest Integer Value, You Can Use The Following Query:

```
SELECT PRESCRIPTION_CODE, ROUND(ITEMS * QUANTITY, 0) AS Total_Quantity
FROM Prescriptions;
```

This query selects the PRESCRIPTION_CODE and calculates the Total_Quantity by multiplying ITEMS and QUANTITY for each row in the Prescriptions table. The ROUND function is used to round the result to the nearest integer value (0 decimal places). Below is a screenshot of the first 5 rows.



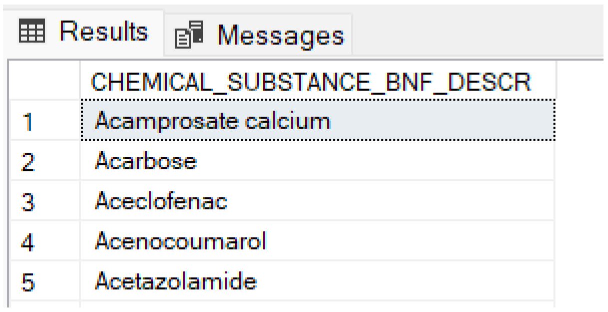
	PRESCRIPTION_CODE	Total_Quantity
1	0	308
2	1	224
3	10	28
4	100	56
5	1000	100

PART 4: DISTINCT CHEMICAL SUBSTANCES

To Return A List Of Distinct Chemical Substances That Appear In The Drugs Table, You Can Use The Following Query:

```
SELECT DISTINCT CHEMICAL_SUBSTANCE_BNF_DESCR
FROM Drugs
ORDER BY CHEMICAL_SUBSTANCE_BNF_DESCR;
```

This query uses the DISTINCT keyword to select unique values of the CHEMICAL_SUBSTANCE_BNF_DESCR column from the Drugs table. The ORDER BY clause sorts the results in alphabetical order. Below is a screenshot of the first 5 rows.



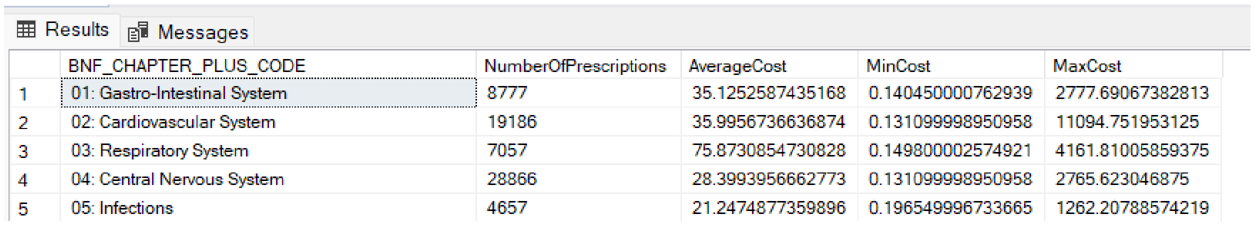
	CHEMICAL_SUBSTANCE_BNF_DESCR
1	Acamprosate calcium
2	Acarbose
3	Aceclofenac
4	Acenocoumarol
5	Acetazolamide

PART 5: NUMBER OF PRESCRIPTIONS

To Return The Number Of Prescriptions For Each Bnf_chapter_plus_code, Along With The Average Cost, Minimum Cost, And Maximum Cost For That Chapter Code, You Can Use The Following Query:

```
SELECT
   D.BNF_CHAPTER_PLUS_CODE,
   COUNT(P.PRESCRIPTION_CODE) AS NumberOfPrescriptions,
   AVG(P.ACTUAL_COST) AS AverageCost,
   MIN(P.ACTUAL_COST) AS MinCost,
   MAX(P.ACTUAL_COST) AS MaxCost
FROM
   Prescriptions AS P
JOIN
   Drugs AS D ON P.BNF_CODE = D.BNF_CODE
GROUP BY
   D.BNF_CHAPTER_PLUS_CODE
ORDER BY
   D.BNF_CHAPTER_PLUS_CODE;
```

This query joins the Prescriptions table (P) with the Drugs table (D) on the BNF_CODE column. It then groups the results by the BNF_CHAPTER_PLUS_CODE column and calculates the count, average, minimum, and maximum cost for each group. The results are ordered by the BNF_CHAPTER_PLUS_CODE. Below is a screenshot of the first 5 rows.



	BNF_CHAPTER_PLUS_CODE	NumberOfPrescriptions	AverageCost	MinCost	MaxCost
1	01: Gastro-Intestinal System	8777	35.1252587435168	0.140450000762939	2777.69067382813
2	02: Cardiovascular System	19186	35.9956736636874	0.131099998950958	11094.751953125
3	03: Respiratory System	7057	75.8730854730828	0.149800002574921	4161.81005859375
4	04: Central Nervous System	28866	28.3993956662773	0.131099998950958	2765.623046875
5	05: Infections	4657	21.2474877359896	0.196549996733665	1262.20788574219

PART 6: MOST EXPENSIVE PRESCRIPTION BY EACH PRACTICE

To return the most expensive prescription prescribed by each practice, where the cost is more than £4000, you can use the following query:

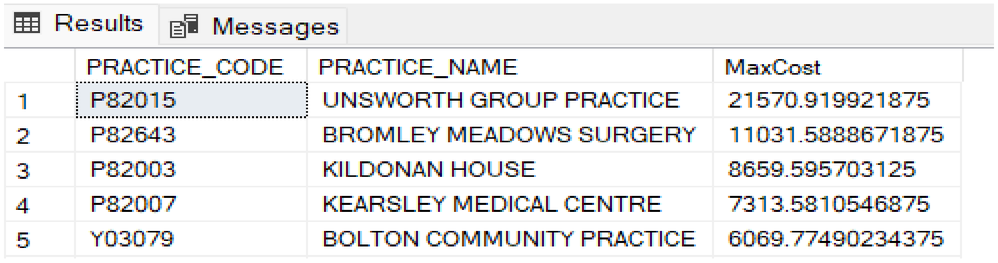
```
WITH MostExpensivePrescriptions AS (
   SELECT
      P.PRACTICE_CODE,
      MAX(P.ACTUAL_COST) AS MaxCost
   FROM
      Prescriptions AS P
   GROUP BY
      P.PRACTICE_CODE
   HAVING
      MAX(P.ACTUAL_COST) > 4000
)

SELECT
   M.PRACTICE_CODE,
   MP.PRACTICE_NAME,
   M.MaxCost
FROM
   MostExpensivePrescriptions AS M
JOIN
   Medical_Practice AS MP ON M.PRACTICE_CODE = MP.PRACTICE_CODE
ORDER BY
   M.MaxCost DESC;
```

This query first creates a Common Table Expression (CTE) named MostExpensivePrescriptions that groups the Prescriptions table by the

PRACTICE_CODE and calculates the maximum cost for each group, filtering only those with a maximum cost greater than £4000.

Then, the main query selects the PRACTICE_CODE, PRACTICE_NAME, and the maximum cost from the CTE (MostExpensivePrescriptions) and the Medical_Practice table, joined on the PRACTICE_CODE column. The results are ordered by the maximum cost in descending order. Below is a screenshot of the first 5 rows.



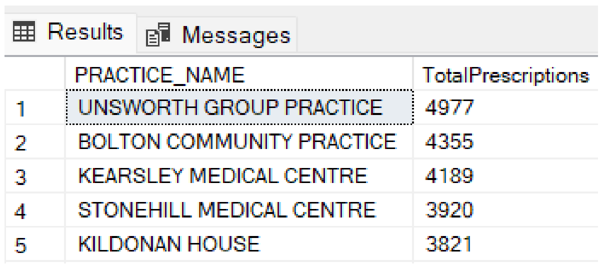
	PRACTICE_CODE	PRACTICE_NAME	MaxCost
1	P82015	UNSWORTH GROUP PRACTICE	21570.919921875
2	P82643	BROMLEY MEADOWS SURGERY	11031.5888671875
3	P82003	KILDONAN HOUSE	8659.595703125
4	P82007	KEARSLEY MEDICAL CENTRE	7313.5810546875
5	Y03079	BOLTON COMMUNITY PRACTICE	6069.77490234375

PART 7: ADDITIONAL QUERIES

1. Find the total number of prescriptions for each medical practice, along with their practice name, sorted by the number of prescriptions in descending order.

```
SELECT
   MP.PRACTICE_NAME,
   COUNT(P.PRESCRIPTION_CODE) AS TotalPrescriptions
FROM
   Medical_Practice AS MP
JOIN
   Prescriptions AS P ON MP.PRACTICE_CODE = P.PRACTICE_CODE
GROUP BY
   MP.PRACTICE_NAME
ORDER BY
   TotalPrescriptions DESC;
```

This query retrieves the total number of prescriptions issued by each medical practice and displays the results in descending order based on the total number of prescriptions. The result will be a list of medical practices along with the total number of prescriptions they have issued, sorted in descending order by the number of prescriptions. Below is a screenshot of the first 5 rows.



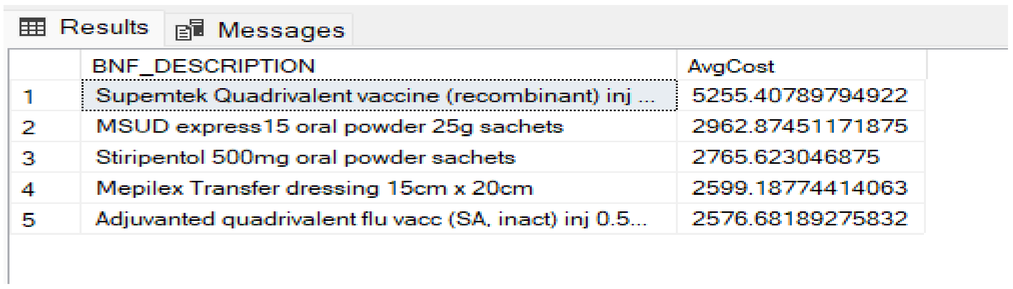
Results | Messages

	PRACTICE_NAME	TotalPrescriptions
1	UNSWORTH GROUP PRACTICE	4977
2	BOLTON COMMUNITY PRACTICE	4355
3	KEARSLEY MEDICAL CENTRE	4189
4	STONEHILL MEDICAL CENTRE	3920
5	KILDONAN HOUSE	3821

2. Find the top 5 drugs with the highest average cost per prescription.

```
SELECT
   TOP 5 D.BNF_DESCRIPTION,
   AVG(P.ACTUAL_COST) AS AvgCost
FROM
   Drugs AS D
JOIN
   Prescriptions AS P ON D.BNF_CODE = P.BNF_CODE
GROUP BY
   D.BNF_DESCRIPTION
ORDER BY
   AvgCost DESC;
```

The result of this query will be a list of the top 5 drugs with the highest average prescription cost, sorted in descending order by the average cost of each drug. Below is a screenshot of the result.



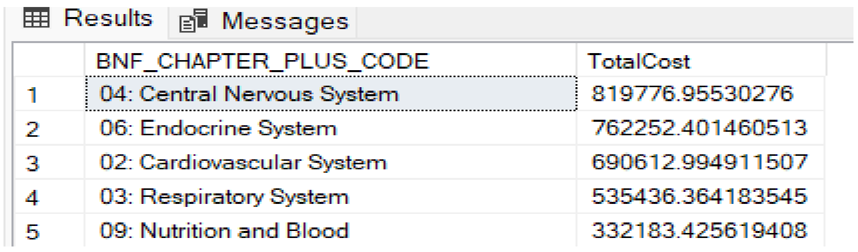
Results | Messages

	BNF_DESCRIPTION	AvgCost
1	Supemtek Quadrivalent vaccine (recombinant) inj ...	5255.40789794922
2	MSUD express15 oral powder 25g sachets	2962.87451171875
3	Stiripentol 500mg oral powder sachets	2765.623046875
4	Mepilex Transfer dressing 15cm x 20cm	2599.18774414063
5	Adjuvanted quadrivalent flu vacc (SA, inact) inj 0.5...	2576.68189275832

3. Find the total cost of prescriptions for each BNF_CHAPTER_PLUS_CODE, only including chapters with a total cost greater than £10,000.

```
SELECT
   D.BNF_CHAPTER_PLUS_CODE,
   SUM(P.ACTUAL_COST) AS TotalCost
FROM
   Drugs AS D
JOIN
   Prescriptions AS P ON D.BNF_CODE = P.BNF_CODE
GROUP BY
   D.BNF_CHAPTER_PLUS_CODE
HAVING
   SUM(P.ACTUAL_COST) > 10000
ORDER BY
   TotalCost DESC;
```

The result of this query will be a list of BNF chapter codes and their total cost where the total cost of prescriptions under each chapter code is greater than 10,000, sorted in descending order by the total cost. Below is a screenshot of the first 5 rows.

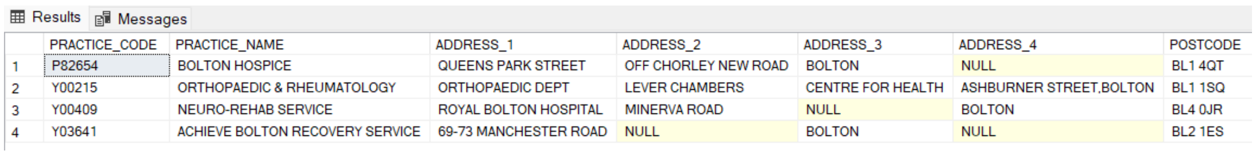


	BNF_CHAPTER_PLUS_CODE	TotalCost
1	04: Central Nervous System	819776.95530276
2	06: Endocrine System	762252.401460513
3	02: Cardiovascular System	690612.994911507
4	03: Respiratory System	535436.364183545
5	09: Nutrition and Blood	332183.425619408

4. Find the medical practices that have never prescribed any drugs in the '05: Infections' BNF chapter.

```
SELECT
   MP.*
FROM
   Medical_Practice AS MP
WHERE
   NOT EXISTS (
      SELECT 1
      FROM
         Prescriptions AS P
      JOIN
         Drugs AS D ON P.BNF_CODE = D.BNF_CODE
      WHERE
         MP.PRACTICE_CODE = P.PRACTICE_CODE AND D.BNF_CHAPTER_PLUS_CODE = '05: Infections'
   );
```

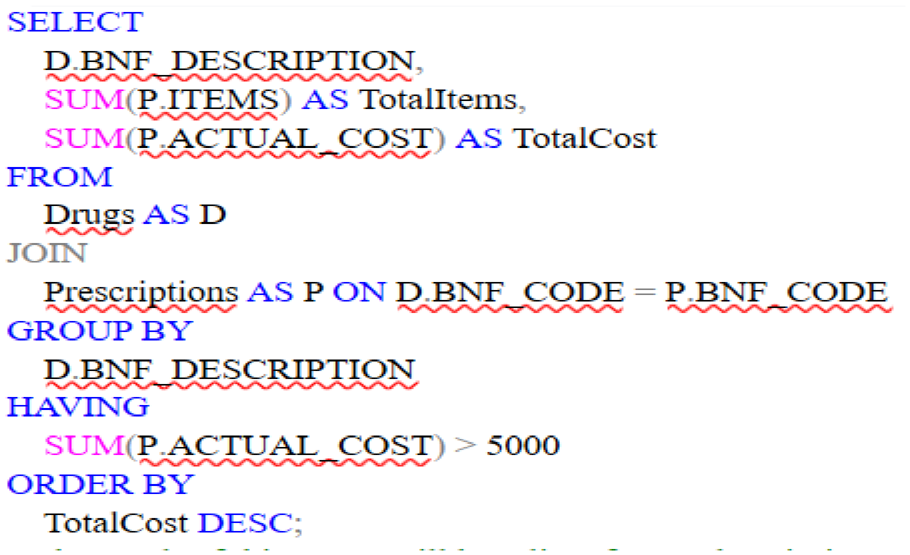
The result of this query will be a list of all medical practice details that have not prescribed any medication in the BNF chapter code '05: Infections'. Below is a screenshot of the result.



Results | Messages

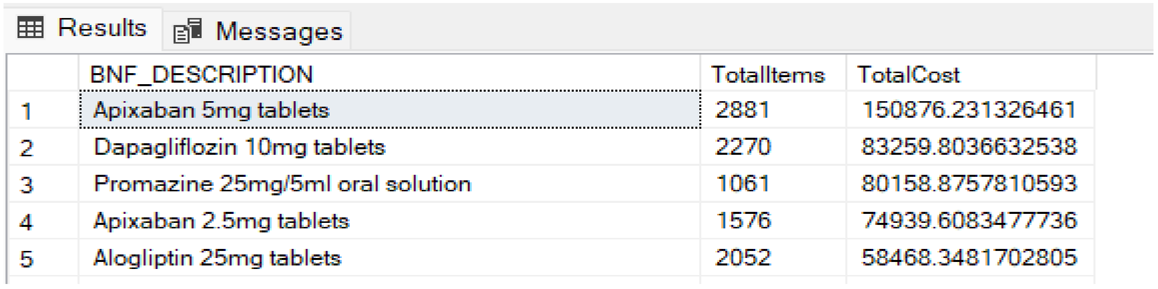
	PRACTICE_CODE	PRACTICE_NAME	ADDRESS_1	ADDRESS_2	ADDRESS_3	ADDRESS_4	POSTCODE
1	P82654	BOLTON HOSPICE	QUEENS PARK STREET	OFF CHORLEY NEW ROAD	BOLTON	NULL	BL1 4QT
2	Y00215	ORTHOPAEDIC & RHEUMATOLOGY	ORTHOPAEDIC DEPT	LEVER CHAMBERS	CENTRE FOR HEALTH	ASHBURNER STREET,BOLTON	BL1 1SQ
3	Y00409	NEURO-REHAB SERVICE	ROYAL BOLTON HOSPITAL	MINERVA ROAD	NULL	BOLTON	BL4 0JR
4	Y03641	ACHIEVE BOLTON RECOVERY SERVICE	69-73 MANCHESTER ROAD	NULL	BOLTON	NULL	BL2 1ES

5. Find the total items and total cost for each drug, where the total cost is greater than £5000, sorted by total cost in descending order.



```
SELECT
    D.BNF_DESCRIPTION,
    SUM(P.ITEMS) AS TotalItems,
    SUM(P.ACTUAL_COST) AS TotalCost
FROM
    Drugs AS D
JOIN
    Prescriptions AS P ON D.BNF_CODE = P.BNF_CODE
GROUP BY
    D.BNF_DESCRIPTION
HAVING
    SUM(P.ACTUAL_COST) > 5000
ORDER BY
    TotalCost DESC;
```

The result of this query will be a list of BNF descriptions, total items prescribed, and the total cost of the drugs with a total cost greater than 5000, ordered by the total cost in descending order. Below is a screenshot of the first 5 rows.

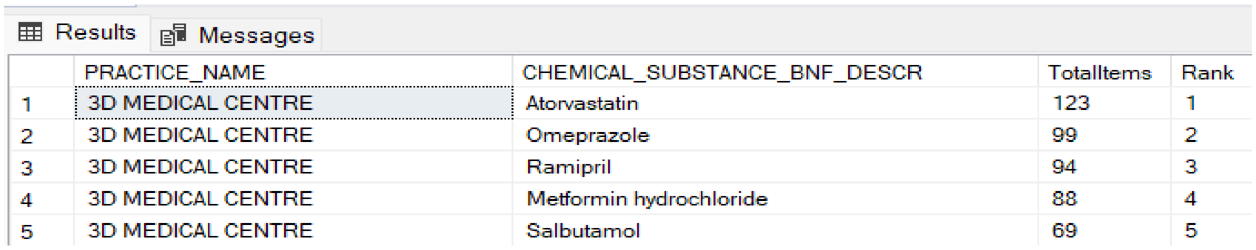


	BNF_DESCRIPTION	TotalItems	TotalCost
1	Apixaban 5mg tablets	2881	150876.231326461
2	Dapagliflozin 10mg tablets	2270	83259.8036632538
3	Promazine 25mg/5ml oral solution	1061	80158.8757810593
4	Apixaban 2.5mg tablets	1576	74939.6083477736
5	Alogliptin 25mg tablets	2052	58468.3481702805

6. Find the top 5 most prescribed drugs by total items for each medical practice.

```
WITH PracticeDrugs AS (
  SELECT
    MP.PRACTICE_NAME,
    D.CHEMICAL_SUBSTANCE_BNF_DESCR,
    SUM(P.ITEMS) AS TotalItems
  FROM
    Medical_Practice AS MP
  JOIN
    Prescriptions AS P ON MP.PRACTICE_CODE = P.PRACTICE_CODE
  JOIN
    Drugs AS D ON P.BNF_CODE = D.BNF_CODE
  GROUP BY
    MP.PRACTICE_NAME, D.CHEMICAL_SUBSTANCE_BNF_DESCR
),
Ranking AS (
  SELECT
    PD.PRACTICE_NAME,
    PD.CHEMICAL_SUBSTANCE_BNF_DESCR,
    PD.TotalItems,
    ROW_NUMBER() OVER (PARTITION BY PD.PRACTICE_NAME ORDER BY PD.TotalItems DESC) AS Rank
  FROM
    PracticeDrugs AS PD
)
SELECT
  R.PRACTICE_NAME,
  R.CHEMICAL_SUBSTANCE_BNF_DESCR,
  R.TotalItems,
  R.Rank
FROM
  Ranking AS R
WHERE
  R.Rank <= 5;
```

The result of this query returns the top 5 most prescribed drugs by total items for each medical practice. A common table expression (CTE) is used to group the total items by medical practice and drug. The ROW_NUMBER () function is used to rank the drugs for each medical practice by total items in descending order. The final result is filtered to show only the top 5 drugs for each medical practice. Below is a screenshot of the first 5 rows.

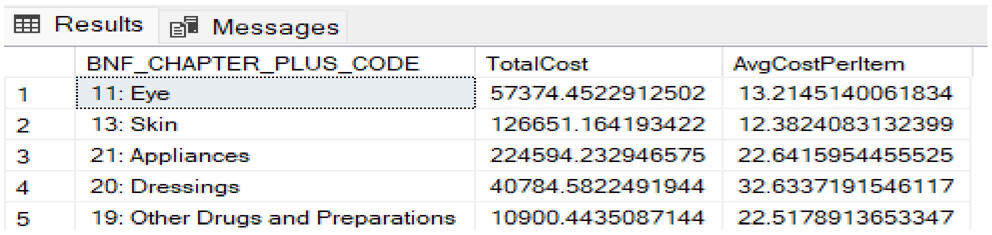


	PRACTICE_NAME	CHEMICAL_SUBSTANCE_BNF_DESCR	TotalItems	Rank
1	3D MEDICAL CENTRE	Atorvastatin	123	1
2	3D MEDICAL CENTRE	Omeprazole	99	2
3	3D MEDICAL CENTRE	Ramipril	94	3
4	3D MEDICAL CENTRE	Metformin hydrochloride	88	4
5	3D MEDICAL CENTRE	Salbutamol	69	5

7. Find the total cost and average cost per item for each BNF chapter code.

```
SELECT
    D.BNF_CHAPTER_PLUS_CODE,
    SUM(P.ACTUAL_COST) AS TotalCost,
    AVG(P.ACTUAL_COST / P.ITEMS) AS AvgCostPerItem
FROM
    Drugs AS D
JOIN
    Prescriptions AS P ON D.BNF_CODE = P.BNF_CODE
GROUP BY
    D.BNF_CHAPTER_PLUS_CODE;
```

The result of this query returns the total cost and average cost per item for each BNF chapter code. The query joins the Drugs and Prescriptions tables and groups the results by BNF chapter code. The total cost is calculated by summing the actual cost of the prescriptions, and the average cost per item is calculated by dividing the actual cost by the number of items and taking the average.

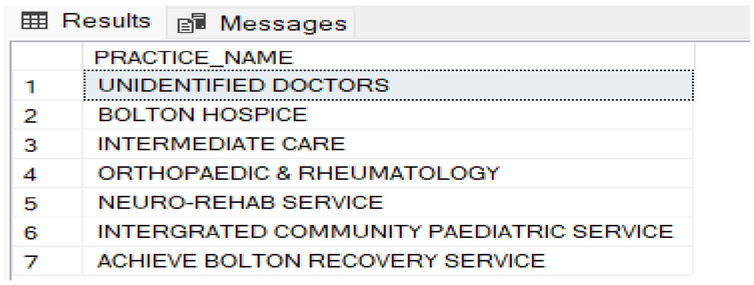


	BNF_CHAPTER_PLUS_CODE	TotalCost	AvgCostPerItem
1	11: Eye	57374.4522912502	13.2145140061834
2	13: Skin	126651.164193422	12.3824083132399
3	21: Appliances	224594.232946575	22.6415954455525
4	20: Dressings	40784.5822491944	32.6337191546117
5	19: Other Drugs and Preparations	10900.4435087144	22.5178913653347

8. Find the medical practices that have never prescribed a specific drug.

```
SELECT
    MP.PRACTICE_NAME
FROM
    Medical_Practice AS MP
WHERE NOT EXISTS (
    SELECT 1
    FROM
        Prescriptions AS P
    JOIN
        Drugs AS D ON P.BNF_CODE = D.BNF_CODE
    WHERE
        MP.PRACTICE_CODE = P.PRACTICE_CODE
        AND D.CHEMICAL_SUBSTANCE_BNF_DESCR = 'Amoxicillin'
);
```

The result of this query returns the medical practices that have never prescribed a specific drug (e.g., Amoxicillin). The query uses a subquery with the NOT EXISTS clause to check for the absence of a prescription for the specified drug in each medical practice. The subquery joins the Prescriptions and Drugs tables and filters the results based on the practice code and the specified drug's chemical substance description. Below is a screenshot of the result.



Results | Messages

	PRACTICE_NAME
1	UNIDENTIFIED DOCTORS
2	BOLTON HOSPICE
3	INTERMEDIATE CARE
4	ORTHOPAEDIC & RHEUMATOLOGY
5	NEURO-REHAB SERVICE
6	INTERGRATED COMMUNITY PAEDIATRIC SERVICE
7	ACHIEVE BOLTON RECOVERY SERVICE

2.0 CONCLUSIONS

The database design solution for PrescriptionsDB includes three tables: Medical_Practice, Drugs, and Prescriptions. The Medical_Practice table includes practice details, such as practice code, name, address, and postcode. The Drugs table includes details of prescribed drugs, such as the BNF code, chemical substance, and product description. The Prescriptions table includes details of individual prescriptions, such as the prescription code, practice code, BNF code, quantity, items, and actual cost.

The key functionality provided by the database includes the ability to store and retrieve information about medical practices, prescribed drugs, and individual prescriptions. The database also includes foreign key constraints, which ensure that the relationships between the tables are maintained and that data integrity is maintained.