Coursework

Advanced Databases

MSc Data Science

University of Salford MANCHESTER

ASSIGNMENT

ON

DATABASE DESIGN AND SQL FOR DATA ANALYSIS

ONYEAJUNWANNE JUDE UFOH

@00324993

APRIL, 2023

Task 1 Report

Introduction:

This task requires the creation of a database management system that will be used to manage the operation of a library. It will require the creation of tables that will hold the library data and the procedures for accessing and retrieving data from the tables while ensuring the overall security of the database.

Assumptions:

Within the scope of **this** design, it is assumed

1. Library has one copy of each time
2. Members can take more than one loan
3. Due date for a loan is 21 days

Part 1

In this database design, we work with the given requirements and employ the bottom down approach by recognizing the entities, attributes and relationships. To recognize the entities, we look for the 'nouns' in the design requirement.

Entities (Tables)

1. **Members** that *use* the library
2. **Items** that are *used* by **Members**
3. **Loans** that *result* from **Members** *being given* **Items** from the library
4. **Fines** that *result* from **Members** *being given* **Items** on **Loan**
5. **Items** in the library such as books are written by an **author(s)**.
6. **Fine repayment** *made* on **fines**
7. **Lost/removed items** from the library.
8. **Former members** who *left* the library

Relationships and Cardinality

1. **Members/Loan:** Loans are given to members. A member can take many loans. The relationship is ***one-to-many***
2. **Loans/Items:** Items are loaned out. There is a ***one-to-many*** relationship between items and loans because one item can be loaned out severally. The primary key of the item must be referenced in the loan table as foreign key.
3. **Loans/Fines:** Fines results from loans given. Every fine is as a result of loan given out but not all loans result to fine. The relationship is ***one-to-one***. The primary key of the loan must be referenced in the fine table as a foreign key to indicate the particular loan.

4. **Fines/Fine_Repayment:** Fines get repaid. Every fine repayment is as a result of a fine incurred. This is a ***one-to-one*** relationship. The fines primary key must be referenced in the fine_repayment table as a foreign key.
5. **Items/Authors:** Authors produce/write items. It is possible for one author to write many items. One item can also be written by many authors. Therefore, the relationship between authors and items is ***many-to-many***. In this wise, a separate table must be created which will have the primary keys of both authors and items as foreign keys.
6. **Items/Lost_Items:** Items get lost. Lost_items and items have ***one-to-one*** relationship because an item must first exist before it can be said to be lost. The items primary key must be referenced in the lost_items table.
7. **Members/formerMembers:** Members and formerMembers have ***one-to-one*** relationship bacause every formerMember is a member that left the members table. The primary key of the members table is referenced in the formerMembers table as a foreign key.

Entity Relationship Diagram

Here the attributes are added to individual entities and the entities are linked to one another according to the existing relationships.

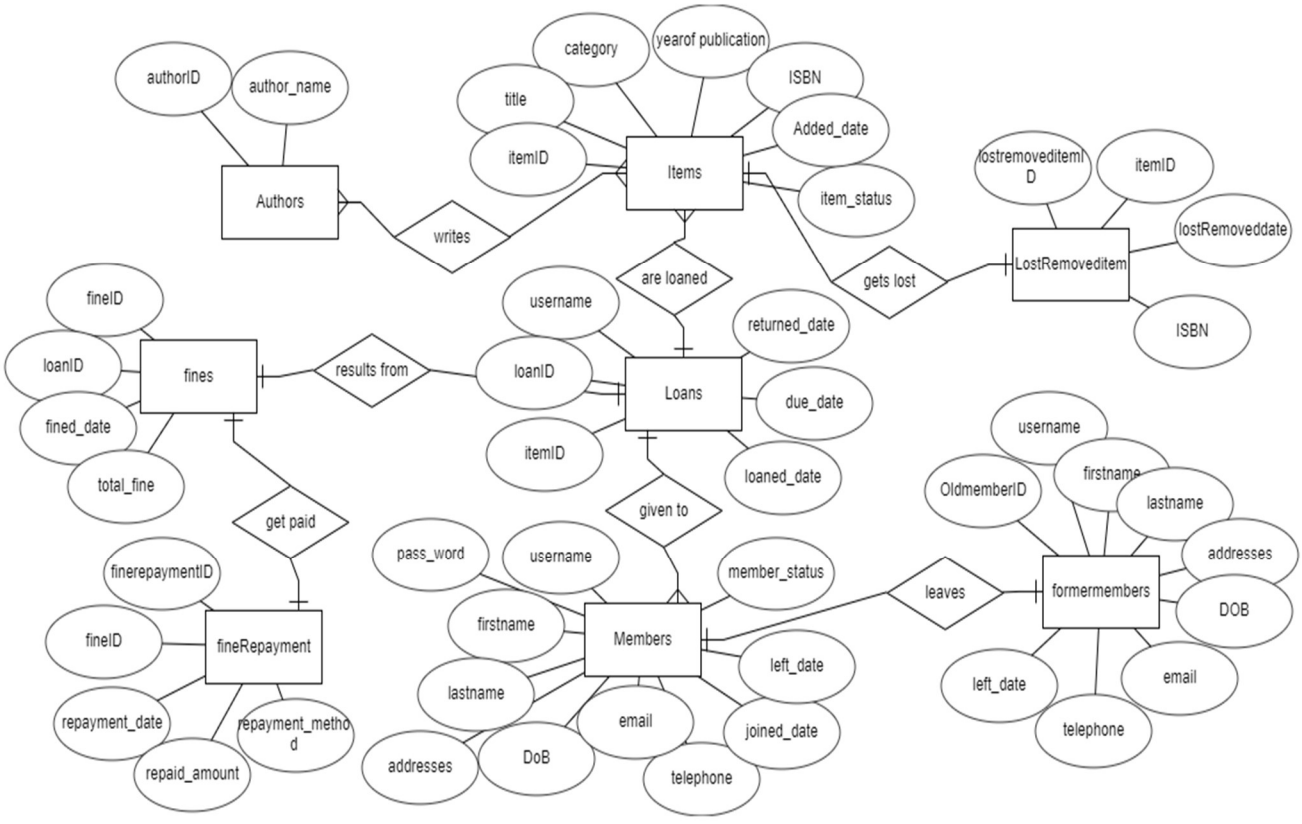
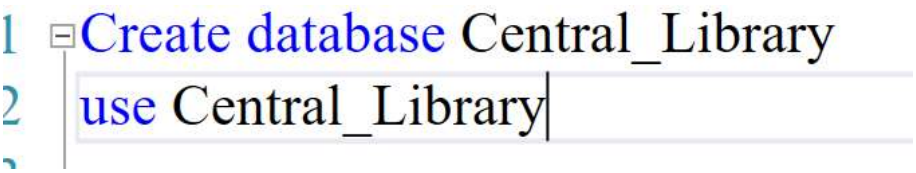


Figure 2 : Entity Relationship Diagram for Central_Library Database

CREATION OF DATABASE

The database is named Central_library and it is created in MSSM studio with the code below



CREATION OF TABLES

The designing of the tables we strictly followed the principle of 3NF normalization which requires:

1. The table must be in the second normal form first whereby partial dependencies are removed

2.There should be no transitive functional dependency in which a non-key column depends on another non key column

Creating the members table

```
create table members(
username nvarchar(10) not null primary key,
Pass_word BINARY(64) NOT NULL,
firstname nvarchar(50) not null,
lastname nvarchar(50) not null,
addresses nvarchar(100) not null,
DoB date not null,
email nvarchar(100) unique ,
telephone nvarchar(20),
joined_date date not null default getdate(),
left_date date,
member_status nvarchar(20) default 'Active',
constraint chk_members check (member_status ='Inactive' or member_status='Active' and email like '%_@_%._%' ))
```

Username is the primary key and it is of the nvarchar data type to make it accept any kind of character but it is restricted to accept only ten characters. The Pass_word is made to be a binary column with a length of 64 bytes(512bits) in order for it to store the hashed password of each member for security reasons. The email field is set to **unique** to avoid duplication. Joined_date is set to default date which implies that it assumes the current date of joining if no date is specified. Member_status is set to default active to ensure that a member becomes active as soon as he is added. The table has a 'check' constraint named 'chk_members' that ensures that the members_status is either 'Active' or 'Inactive' and that the email address is valid (i.e it contains '@' and '.' symbols). Not null fields are compulsory while fields without not null are not compulsory.

Creating Authors table

the authors table has the authorID as the primary key and it is created automatically. The author_name is given nvarchar data type since it can be any name.

```
create table authors(
authorID int identity(1,1) not null primary key,
author_name nvarchar (100)
)
```

Creating Authors/Items table

authorID and itemID are the only fields in this table and they are referenced from the authors and items tables respectively.

```
create table authors_items(
authorID int foreign key (authorID) references authors(authorID),
itemID int foreign key (itemID) references items(itemID)
)
```

Creating the items table

```
create table items (
 itemID int identity(1,1) not null primary key,
 title nvarchar(500) not null,
 category nvarchar(50) not null,
 year_of_publication int not null,
 ISBN nvarchar (50),
 added_date date default getdate() not null,
 item_status nvarchar(50) default 'Available',
 constraint chk_items check(
  (item_status='Available' or item_status='On Loan' or item_status='Removed' or item_status='overdue')
  and
  (category ='Book' or category ='Journal' or category ='DVD' or category ='Other Media')
 )
);
```

ItemID is the primary key, Title, Category, Publication date, added date, Status and ISBN. The **title** and **category** fields are nvarchar data type and because the fields are compulsory, they must not accept null values. **ISBN** also accepts nvarchar data type but since it is only applicable to books, it is nullable. The **added_date** field is compulsory and it is of the date data type and it has a default value of getdate() function. This implies that it automatically assumes the date of the day the book was added. **Item_status** is of nvarchar data type and it has a default value of 'Available'. This implies that as an item is added to the database, the item is made available. The **year_of_publication** field is given the integer data type because it is just the year value and not a date. It is also not nullable. The table has a check constraint called 'chk_items that ensures that the item status can accept any of 'Available', 'on loan', 'Removed' or 'Overdue' and that the category column can only accept any of 'Book', 'Jounal', 'DVD', or 'Other Media'.

Creating the Loans table

The loan table will have the automatically generated integer type primary key named **loanID.**
Loaned_date will be of the date data type, not null and have a default value of getdate() to ensure that

the date the loan was taken is automatically registered. **Due_date** is of the date data type and it is not null. **Returned_date** is also of the date data type and will remain null until the item is returned to the library. **Username** and **itemID** are referenced as foreign keys from members and items tables respectively.

```
create table loans(
loanID int identity (1,1) not null primary key,
username nvarchar(10) not null foreign key (username) references members(username),
itemID int not null foreign key (itemID) references items(itemID),
loaned_date date not null default getdate(),
due_date date not null,
returned_date date
)
```

Creating the Fines table.

The Fines table contains the automatically generated integer type fineID as the primary key. **Fined_date** is of date data type and it is not null because it is a compulsory field. **Total_fine** has the money data type and it is not null because it is automatically calculated from the fine rate. **loanID** is a foreign key being referenced from loans table.

```
create table fines(
fineID int identity (1,1) not null primary key,
loanID int not null foreign key (loanID) references loans(loanID),
fined_date date not null,
total_fine money not null
)
```

Creating Fine Repayment table

The primary key is the automatically generated integer type **finerepaymentID**. The repayment date is given datetime data type and it is a compulsory field. The **repaid_amount** is of the money datatype and it is not null **repayment_method** has nvarchar data type and it is constrained to accept only one of two strings 'cash' or 'card'.

```
create table fineRepayment(
finerepaymentID int identity(1,1) not null primary key,
fineID int not null foreign key (fineID) references fines(fineID),
repayment_date datetime not null,
repaid_amount money not null,
repayment_method nvarchar(5) not null check (repayment_method='Cash' or repayment_method='Card')
)
```

Creating Lost/Removed Items table

The primary key will be an automatically generated integer data type field named lostremoveditemID. The itemID is referenced as a foreign key from the items table. lostRemoved_date is given the date data type and it is a compulsory field. ISBN is the same as contained in the items table and since some of the items in the library does not have an ISBN, it is not a compulsory field.

```
create table lostRemovedItems(
lostremoveditemID int identity(1,1) not null primary key,
itemID int not null foreign key (itemID) references items(itemID),
lostRemoved_date date not null,
ISBN nvarchar(50)
)
```

Creating the formerMembers table

The formermemberID is the primary key and the Username is referenced from the members table as the foreign key. The left_date is set to default to the date the formermember left the membership and joined the formerMember table. There is a constrained applied to the email to ensure that only valid emails are in the old members table.

```
---creating formerMembers
create table formerMembers(
oldmemberID int identity(1,1) not null primary key,
username nvarchar(10) not null foreign key (username) references members(username),
firstname nvarchar(50) not null,
lastname nvarchar(50) not null,
addresses nvarchar(100) not null,
DoB date not null,
email nvarchar(100) unique ,
telephone nvarchar(20),
left_date date default getdate(),
constraint chk_oldmembers check (email like '%_@_%._%' ))
```

Data type	length	columns where used	Justification
Binary	64	pass_word	used to ensure hashing
nvarchar	varying length	username, title firstname,lastname,addresses,email,telephone,member_status, author_name,ISBN,item_status,category,repayment method	because the columns can take both text,number or special characters upto the length specified
int		author_ID, itemID, loanID,fineID,lostremoveditemID,oldmemberID, year_of_publication	because the columns can be counted or indexed or incremented
date		joined_date, left_date,added_date,loaned_date,returned_date, fined_date,repayment_date	Because the fields represent a referenece to a point in time
money		repaid amount, total_fine	used to show that the field is money.

Table1: Summary of Data types used and justification

Database Diagram

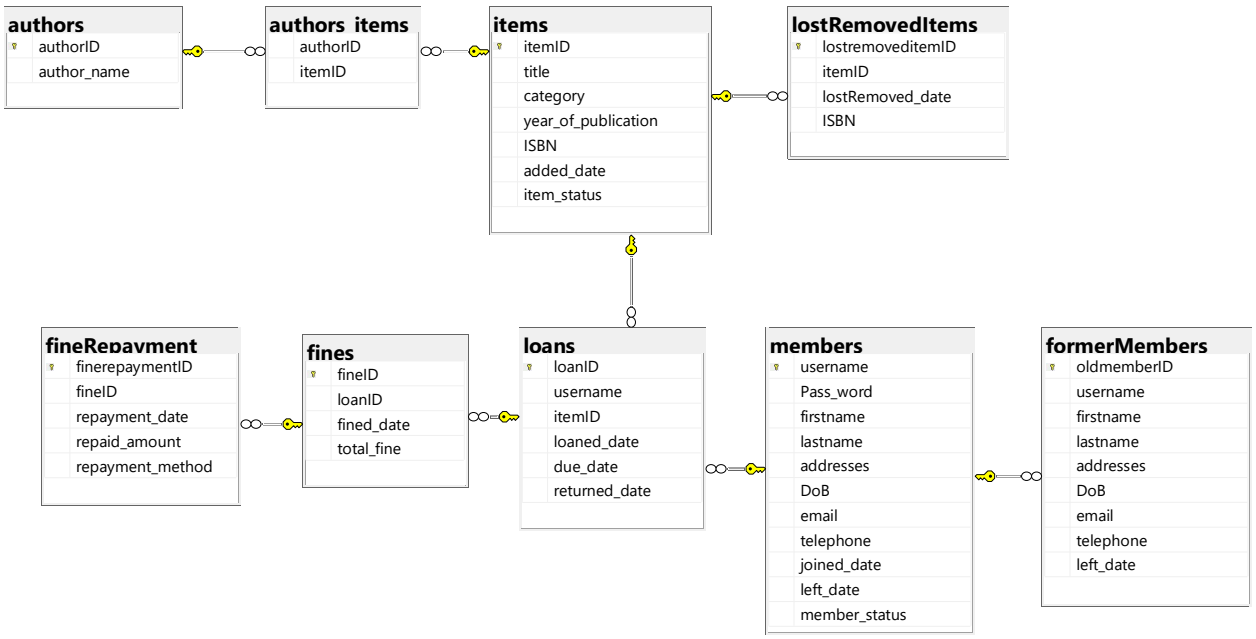


Figure 2 The Database Diagram

Part 2

2a. Query to search the catalogue for matching strings

```
379 create procedure  search_item_by_title
380   @title nvarchar(500)
381  as
382 begin
383   set nocount on;
384
385   select items.*,
386     string_agg(authors.author_name, ', ') as authors
387   from items
388   JOIN authors_items on items.itemID = authors_items.itemID
389   JOIN authors on authors_items.authorID = authors.authorID
390   where items.title LIKE '%' + @title + '%'
391   group by items.itemID, items.title, items.category,
392    items.year_of_publication, items.ISBN, items.added_date, items.item_status;
393    end
```

The procedure selects all the columns from items and uses the string_agg function to concatenate the author_name column from the authors table for each itemID. The join clause is used to link the items, authors_items and authors table together. 'Where' clause is used to filter the result to include items whose title column contains the @title parameter. The 'group by' clause groups the result by the columns in items table so as to eliminate duplicates and aggregate the author names for each item.

Example

```
395  exec search_item_by_title @title = 'python';
```

100 %

Results | Messages

	itemID	title	category	year_of_publication	ISBN	added_date	item_status	authors
1	28	Python Programing Essentials	book	2017	8-9999-4444-8	2020-07-11	Available	Azadeh Mo, Taha Mansor
2	29	Python Programing Tutorials	DVD	2022	NULL	2023-01-07	Available	John Elbert

2b. Stored Procedure for items on loan with a due date less than certain number of days

```
398 create proc Timebeforeduedate @due_in int
399  as
400 begin
401 select a.Title, b.FirstName +'  '+b.LastName as borrower, c.due_date as duedate,
402  DATEDIFF(day, (cast (getdate() as date)), c.due_date) as remainingdays
403  from items a
404  inner join loans c on a.itemID = c.itemID
405  inner join members b on c.username = b.username
406  where  DATEDIFF(day, (cast (getdate() as date)), c.due_date) < @due_in
407  and DATEDIFF(day, (cast (getdate() as date)), c.due_date) > 0 and c.returned_date is null
408  end
409  exec Timebeforeduedate @due_in = 5
```

The name of the procedure is Timebeforeduedate. It accepts an integer input and selects the item title, firstname concatenated with lastname , due_date, remaining days which is calculated as the difference

between current date and the due date. It filters the result using the 'where' clause to include only the days where due date is less than the procedure's variable and where due date is greater than zero to exclude items that have not reached the due date and where returned_date is not null to exclude the items that have been returned. With this, we can see items that have due dates less than any number of days.

Example: below are examples for items that are due in 5 and 20 days respectively

```
409  exec Timebeforeduedate @due_in = 5
```

100 %

Results | Messages

	Title	borrower	duedate	remainingdays
1	Statistical Theory	Aurlie Durand	2023-04-25	4

```
409  exec Timebeforeduedate @due_in = 20
```

.00 %

Results | Messages

	Title	borrower	duedate	remainingdays
1	Artificial Intelligence	Cletus Loggerman	2023-05-10	19
2	Statistical Theory	Aurlie Durand	2023-04-25	4

2c. stored procedure to insert a new member.

The procedure is named insertaNewMember and it accepts 11 input parameters and insert them into the members table. Because the pass_word field is hashed, nvarchar input will not be accepted. We therefore use the **CONVERT** function to convert it to binary before it is inserted into the table. The insert into statement is wrapped inside a try-catch block to ensure that if an error occurs, the catch block will execute, rolling back the transaction using rollback and throwing an error using **throw**

```
411 create a procedure insertaNewMember
412   (@username nvarchar(10), @password nvarchar(50),  @firstname nvarchar(50),
413   @lastname nvarchar(50), @address nvarchar(500), @dob date, @email nvarchar(500),
414   @phoneNo nvarchar(50), @joindate date, @leftdate date = NULL,
415   @status nvarchar(10) )
416 as
417 begin
418   begin try
419     begin transaction
420
421     insert into members
422     (Username, Pass_word, FirstName, LastName, Addresses, DoB,Email, Telephone, Joined_Date,
423      Left_Date, member_Status)
424     values  (@username,  (convert(binary, '@password'))   @firstname,  @lastname, @address,
425       @dob,  @email, @phoneNo, @joindate, @leftdate,  @status)
426       commit transaction
427   cnd try
428   begin catch
429     rollback
430     throw
431   end catch
432 end
```

Example:

```
119 exec insertNewMember @username ='mfm460',
120 @password='hjujd',
121 @firstname = 'Jude',
122 @lastname= 'okoye',
123 @address = '72 oldhan road Uk',
124 @dob = '1994-11-04',
125 @email = 'Jude3@ufoh.com', @phoneNo ='07867279655', @joindate = '2023-03-09',
126 @leftdate = null, @status= 'Active'
```

100 %

Results Messages

	username	Pass_word	firstname	lastname	addresses	DoB	email	telephone	joined_date	left_date	member_status
1	mfm460	0x4070617373776F726400000000000000000000000000000...	Jude	okoye	72 oldhan road Uk	1994-11-04	Jude3@ufoh.com	07867279655	2023-03-09	NULL	Active

2d. Stored procedure to update the details of an existing member

```
135 create procedure updateMember (@username nvarchar(50), @password  nvarchar(50)=null, @firstname nvarchar(50)=null
136 @lastname nvarchar(50)=null, @address nvarchar(500)=null, @dob date=null, @email nvarchar(500)=null,
137 @phoneNo nvarchar(50)=null, @joindate date=null,
138 @leftdate date =null, @status nvarchar(10)=null)
139 as
140 begin set nocount on;begin try
141       begin transaction;
142 update members
143  set   Username=@username,
144  Pass_word=isnull((convert(binary, @password)), Pass_word ), FirstName=isnull(@firstname, FirstName),
145  LastName=isnull(@lastname,  LastName),Addresses =isNull(@address, Addresses),
146  DoB =isnull(@dob,DoB),Email=isnull(@email, Email),Telephone=isnull(@phoneNo,Telephone),
147  Joined_Date=isnull(@joindate, Joined_Date), Left_Date=isnull(@leftdate, Left_Date),
148  member_Status=isnull(@status,member_Status)
149  where Username=@username
150  commit transaction;
151   end try begin catch if @@trancount > 0
152      begin
153         rollback transaction;
154      end;
155      throw;
156   end catch;
157 end
```

The procedure can accept any of eleven inputs. It uses the 'where' clause to filter out the username whose details need to be updated. It then updates the member with the given field. The '**isnull'** function is used to return the default value in each field that is not affected by the update. The update is wrapped inside a **try-catch** to check for errors during insert

Example: we can update the address of the user with username= mfm460 as follows

```
450 updateMember @address = '7 Wiltshere road', @username='mfm460'
```

100 %

Results Messages

	username	Pass_word	firstname	lastname	addresses	DoB	email	telephone	joined_date	left_date	member_status
1	mfm460	0x4070617373776F726400000000000000000000000000000...	Jude	okoye	7 Wiltshere road	1994-11-04	Jude3@ufoh.com	07867279655	2023-03-09	NULL	Active

3. View of the loan history

```sql
--Question3. View of loan history
create view loan_history as
select
  a.loanID,  b.firstname,  b.lastname, b.email, b.telephone,  c.title, c.category, a.loaned_date,  a.due_date,
  a.returned_date,  d.total_fine, coalesce(e.repaid_amount, 0) as total_payment, (d.total_fine-coalesce(e.repaid_amount, 0))
  as Balance
from
  loans as  a
  inner join members as b on a.username = b.username
  inner join items as c on a.itemID = c.itemID
  inner join fines as d on a.loanID=d.loanID
  left join fineRepayment as e on e.fineID= d.fineID
```

We select the necessary columns from loans table aliased as **a**, members table aliased as **b**, items table aliased **c**, fines table aliased as **d** and fineRepayment table aliased as **e**. we inner joined all the tables with a common field except the fineRepayment table which was left joined to the fines table. This is to ensure that all the rows in the fines table will be included in the result set even when there is no matching rows in the finesRepayment table.(Because not all fines have any repayment). The **coalesce** function is used to substitute a default value of 0 for the repaid amount column in cases where there is no matching row in the fineRepayment table.

```sql
select * from loan_history
```

100 %

Results Messages

	loanID	firstname	lastname	email	telephone	title	category	loaned_date	due_date	returned_date	total_fine	total_payment	Balance
1	21	Jude	okoye	mike@yahoo.com	07867278655	Physical Anatomy	book	2023-02-21	2023-03-14	NULL	380.00	250.00	130.00
2	41	Ginnifer	Kyme	gkymen@ezinearticles.com	831-632-9655	Tableau Tutorial	DVD	2023-03-28	2023-04-18	NULL	30.00	0.00	30.00
3	40	Leoine	Ablitt	lablittl@businessinsider.com	821-623-3063	strength Training	book	2022-04-19	2022-05-10	NULL	3460.00	0.00	3460.00
4	39	Winny	O'Fog...	wofogartyk@mapquest.com	696-762-3133	Stitistics and R...	book	2022-04-12	2022-05-03	NULL	3530.00	0.00	3530.00
5	37	Penn	Willstrop	pwillstropi@omniture.com	197-701-7401	SQL made Simple	book	2023-01-24	2023-02-14	NULL	660.00	0.00	660.00
6	36	Fallon	Corton	fcortonh@cpanel.net	591-694-1521	R Programing	book	2023-01-15	2023-02-05	NULL	750.00	0.00	750.00
7	35	Land	Manser	lmanserg@themeforest.net	267-149-2034	Python Progra...	DVD	2023-03-07	2023-03-28	NULL	240.00	0.00	240.00
8	34	Siusan	Ilyenko	silyenkof@parallels.com	222-611-8778	Python Progra...	book	2023-01-12	2023-02-02	NULL	780.00	0.00	780.00
9	33	Alyse	Vinas	avinase@hud.gov	880-428-3276	Principles of Da...	book	2023-02-08	2023-03-01	NULL	510.00	0.00	510.00
10	31	Templ...	Seage	tseageb@columbia.edu	251-519-1530	Prescott's Micro...	Book	2023-01-17	2023-02-07	NULL	730.00	0.00	730.00
11	30	Johnat...	Oaken...	joakenforda@blog.com	967-681-6899	Power Bi Tutorial	DVD	2022-12-07	2022-12-28	NULL	1140.00	0.00	1140.00
12	28	Harry	Leethem	hleethem7@sogou.com	448-709-9152	Monthly Soccer...	Other ...	2023-02-14	2023-03-07	NULL	450.00	0.00	450.00
13	27	Nedi	Harrin...	nharrinson6@theatlantic.c...	737-369-1847	Microbial Culture	Book	2022-12-27	2023-01-17	NULL	940.00	0.00	940.00
14	25	Stanisl...	Elsie	selsie4@hibu.com	770-639-5396	Learning Disabi...	Journal	2023-02-27	2023-03-20	NULL	320.00	0.00	320.00
15	24	Garik	Rennock	grennock2@vimeo.com	904-552-4065	Ideas in Big Dat...	book	2023-03-01	2023-03-22	NULL	300.00	0.00	300.00
16	23	Ingeborg	Tythert...	itytherton1@jugem.jp	519-254-4467	Hands On R Pr...	DVD	2022-11-02	2022-11-23	2023-02-13	820.00	0.00	820.00
17	22	Rustie	Pandey	rpandey0@topsy.com	169-823-4482	Food Shortage ...	Journal	2023-02-28	2023-03-21	NULL	310.00	0.00	310.00

4. A trigger to set the current status of an item to available when the item is returned.

```
469 alter trigger updateStatus
470 on loans
471 after update
472 as
473 begin
474     declare @itemID int, @returned_date date
475     select @itemID = inserted.itemID, @returned_date = inserted.returned_date
476     from inserted
477     if @returned_date IS NOT NULL
478     begin
479         -- Item has been returned
480         update items
481         set item_status = 'Available' where itemID = @itemID
482         end
483 end
```

The trigger is named updateStatus. it is set on 'loans' and it is fired when the returned_date changes from being null which means an item has been returned. It moves the affected row to the temporarily created inserted table and then set the item_status field to 'Available' at the itemID that matches the one in the 'inserted' table.

5. Total loans on a specified date

```
486 create function TotalLoansOn (@date as date)
487 returns int
488 as
489 begin
490     return (select count(*) as num_loans
491     from loans
492     where loaned_date = @date)
493 end;
```

The name of the function is TotalLoansOn . It accepts a date input and returns a result of the total counts of all the records in the loans table where loaned_date is the same as the input date.

Example: to get a count of loans on 2023-02-08.

```
495  SELECT dbo.TotalLoansOn('2023-02-08') as loan_count;
```

100 %

Results | Messages

	loan_count
1	1

6. inserting Records into the Tables

Inserting records into the items, authors and authors_items table

A stored procedure named **add_items** is used and it accepts six inputs parameters. An item can have more than one author. So it accepts author1_name (First author) and all other authors as author2_name. it then declares three integer variables which will be used to store the IDs of the newly inserted author1, Author2 and the item. After inserting into the authors and items table, the procedure uses the Scope_identity() function to retrieve the IDs from the ID variables.

It inserts the retrieved IDs into the authors_items table. The insert statement is also wrapped inside a **try-catch** block such that when an error occurs during the insert, the catch block will execute rolling back the transaction using roll back and throwing an error using **throw.**

```
129 alter procedure add_items
130  (@author1_name nvarchar(100), @author2_name nvarchar(100), @item_title nvarchar(500),
131   @item_category nvarchar(50),@item_year_of_publication int, @item_ISBN nvarchar(50))
132  as begin begin try
133       begin transaction
134        declare @author1_id int declare @author2_id int  declare @item_id int
135       -- Insert first author and retrieve the new ID
136       insert into authors (author_name) values (@author1_name) set @author1_id =scope_identity()
137        -- Insert second author and retrieve the new ID
138       insert into authors (author_name) values (@author2_name) set @author2_id = scope_identity()
139        -- Insert new item and retrieve the new ID
140       insert into items (title, category, year_of_publication, ISBN)
141       values (@item_title, @item_category, @item_year_of_publication, @item_ISBN) set @item_id = scope_identity()
142        -- Insert new records into authors_items table
143       insert into authors_items (itemID, authorID) values (@item_id, @author1_id)
144      insert into authors_items (itemID, authorID) values (@item_id, @author2_id)
145       commit transaction end try
146    begin catch
147       rollback transaction
148       throw
149    end catch
150 end
```

Example:

```
137 exec add_items @author1_name = ' Tim Peter', @author2_name =null ,
138     @item_title = 'Artificial Intelligence' , @item_category= 'Book' , @item_year_of_publication = 2005,
139     @item_ISBN= '234-yut61-88'
140  ---inserting item with more tha one author
141 exec add_items @author1_name = 'John Peter', @author2_name ='Nath Mike',
142     @item_title ='Big data Analytics' , @item_category="book" ,@item_year_of_publication = 2004,
143     @item_ISBN='898uiu786'
```

100 %

Results | Messages

	itemID	title	category	year_of_publication	ISBN	added_date	item_status
1	1	Artificial Intelligence	Book	2005	234-yut61-88	2023-04-19	Available
2	2	Big data Analytics	book	2004	898uiu786	2023-04-19	Available

	authorID	author_name
1	1	Tim Peter
2	2	NULL
3	3	John Peter
4	4	Nath Mike

	authorID	itemID
1	1	1
2	2	1
3	3	2
4	4	2

Inserting Records into the Loans table

A stored procedure can be used to create a loan. The procedure will take cognizance of the fact that a loan can only be given on an **available item** and to an **active member**. The following is the stored procedure for creating a loan.

```
create PROCEDURE createLoan
@username nvarchar(10), @itemID int, @loaned_date date
as
begin
  if exists (select * from members where username = @username and member_status = 'Active')
    and exists (select * from items where itemID = @itemID and item_status = 'Available')
  begin
    insert loans (username, itemID, loaned_date, due_date, returned_date)
    values (@username, @itemID, @loaned_date, dateadd(day,21,@loaned_date), NULL)
  end
  else
  begin
    raiserror('Check that username exist and item is availaible', 16, 1)
  end
end
```

The name of the procedure is createLoan. It uses the exists function to check if the member is **active** in the members table and if the item is **available** in the items table. When both conditions are met, it goes ahead with insert operation into the loan table. If one or both conditions are not met, it raises an error message. In the insert operation, the procedure accepts three inputs. In the due_date column, the dateadd function is used to add 21 days to the loaned_date. The returned date is always set to null because it is not required at the time of creating a loan.

Example:

```
162 EXEC createLoan @username = 'ecg340', @itemID = 1, @loaned_date ='2023-04-19'
```

100 %

Results | Messages

	loanID	username	itemID	loaned_date	due_date	returned_date
1	1	ecg340	1	2023-04-19	2023-05-10	NULL

Inserting Records into the Fines table

People in the fine table are those that are in the loan table who have either not returned the material (returned_date is null) or who returned the material at a later date than the due date. A trigger that will monitor the loans table for overdue and update the fines table is needed.

```
create trigger Insert_into_fines
on loans
after insert, update
as
begin
  -- Insert fines for new overdue loans
  insert into fines (loanID, fined_date, total_fine)
  select i.loanID, dateadd(day, 22, i.loaned_date), datediff(day, i.due_date, coalesce(i.returned_date, getdate())) * 10
  from inserted i
  left join fines f on i.loanID = f.loanID
  where (i.returned_date > i.due_date OR i.returned_date IS NULL) AND getdate() > i.due_date AND f.loanID IS NULL;

  -- Update fines for existing overdue loans
  update fines
  set total_fine = datediff(day, loans.due_date, coalesce(loans.returned_date, getdate())) * 10
 from fines
  inner join loans on fines.loanID = loans.loanID
  where loans.returned_date is null and getdate() > loans.due_date and fines.total_fine <> datediff(day, loans.due_date,
   coalesce(loans.returned_date, getdate())) * 10
   and not exists(select 1 from inserted i where i.loanID = loans.loanID and i.returned_date is not null );
END
```

The trigger is named **Insert_into_fines** and it is fired after insert or update operation on the loans table. It checks the loans table for any overdue loan and moves the loan to the **inserted** table. it then selects the loanID, fined date and the total fine amount from the inserted table and checks that the loan is not already in the fines table by joining the fines and inserted tables on loanID. If the loanID is not found in the fines table, it is inserted. If however the loan is found in the fines table an update in done on it. the update involves first checking that the returned date is still null. If true which means that the loan has not been returned, the trigger will recalculate the current overdue amount and insert into total_fines. If false, it means that the loan has been returned and the update will stop.

The total fine amount is calculated using the 'datediff' function, which calculates the difference between the due_date and the returned date (or the current date if the item has not been returned yet). The '**coalesce**' function is used to handle cases where the returned date is null i.e when the item has not been returned. The difference in days is multiplied by 10pence.

The fined_date will be the next day after the due_date which is the same as adding 22 days to the loaned_date (**dateadd(day, 22, a.loaned_date)**).

The '**where'** clause is used as a filter to filter only the rows in which returned_date is null or greater than the due_date.

Example:

```
164  EXEC createLoan @username = 'ecg340', @itemID = 1, @loaned_date ='2023-04-19'
165  EXEC createLoan @username = 'myh340', @itemID = 3, @loaned  date ='2023-02-21'
```

100 %

Results | Messages

	loanID	username	itemID	loaned_date	due_date	returned_date
1	20	ecg340	1	2023-04-19	2023-05-10	NULL
2	21	myh340	3	2023-02-21	2023-03-14	NULL

	fineID	loanID	fined_date	total_fine
1	2913439	21	2023-03-15	370.00

The second loan is overdue given the loaned_date. Therefore, it is automatically moved into the fines table. The total_fine for the loanID will keep incrementing by 10 every day until the loan is returned.

Inserting into the fineRepayment table

Only Existent fines that can be repaid. Therefore, for a record to be inserted into the fineRepayment table, there must be a valid fineID. The system must be designed to check that the fineID exists. We can create a stored procedure that can check for the existence of the fineID before inserting into the fineRepayment table with the code below

```
create procedure InsertIntoFineRepayment
   @fineID int, @repaymentDate date, @repaidAmount money, @repaymentMethod varchar(50)
as
begin
  if EXISTS(select 1 from fines where fineID = @fineID)
   begin
     insert into fineRepayment(fineID, repayment_date, repaid_amount, repayment_method)
      values(@fineID, @repaymentDate, @repaidAmount, @repaymentMethod);
  end
else
   begin
     raiserror('FineID does not exist', 16, 1);
   end
end;
```

The name of the stored procedure is InsertIntoFineRepayment. To check that the ID exist, we use the line of code '**if exists(select 1 from fines where fineID =@fineID)**' it works by retrieving a single column with a constant value of 1 form the 'fines' table where the 'fineID' column matches the input parameter '@fineID'. The 'exists' function is used to return true if at least one row is returned and false if none is returned. The 'if' statement further checks the result of the exists function. If it is true, the code runs, and if not the raiserror function returns an error " fineID does not exist"

Example:

If we attempt to insert a repayment with an invalid fineID say 1, it will produce an error message below

```
213  EXEC InsertIntoFineRepayment 1, '2023-04-20', 250, 'cash';
```

100 %

Messages

```
Msg 50000, Level 16, State 1, Procedure InsertIntoFineRepayment, Line 12 [Batch Start Line 212]
FineID does not exist

Completion time: 2023-04-20T13:01:20.7034683+01:00
```

If we a use a valid fine ID,

```
214  EXEC InsertIntoFineRepayment 2913439, '2023-04-20', 250, 'cash';
```

100 %

Results | Messages

	finerepaymentID	fineID	repayment_date	repaid_amount	repayment_method
1	1	2913439	2023-04-20 00:00:00.000	250.00	cash

Inserting into the LostRemovedItemsTable

The lostRemovedItems table gets it data when an item is declared as removed/lost in the items table i.e when the item status changes to 'Removed'. This is only possible with a triggers.

Stored procedure to update item_status in items table

```
create procedure Update_Item_Status
    @itemID int,
    @newStatus nvarchar(10)
as
begin
    update items
    set item_status = @newStatus
    where itemID = @itemID
end
```

Trigger to insert into lostRemovedItems table when the item_status changes to 'Removed'

```
create trigger insertIntoLostRemovedItems
on items
after update
as
begin
    if update(item_status) -- Check if the item_status column was updated
    begin
        declare @itemID int
        select @itemID = inserted.itemID from inserted

        if (select item_status from inserted) = 'Removed'
        begin
            insert into lostRemovedItems (itemID, lostRemoved_date, ISBN)
            select itemID, getdate(), ISBN from inserted where itemID = @itemID
        end
    end
end
```

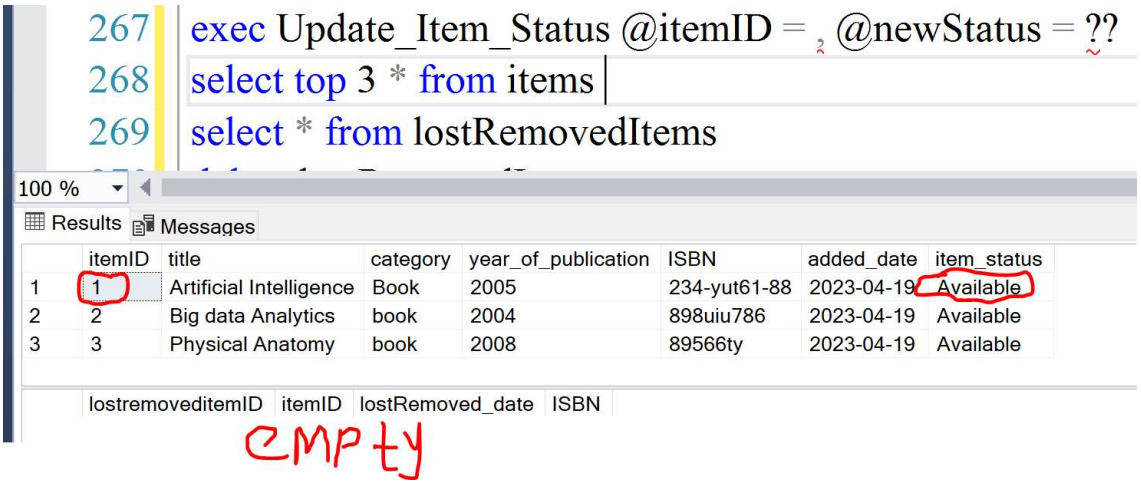
Trigger to delete from lostRemovedItems table when the item_status changes to 'Available'.

```
create trigger deleteFromLostRemovedItems
on items
after update
as
begin
   if update(item_status) -- Check if the item_status column was updated
   begin
      declare @itemID int
      select @itemID = inserted.itemID from inserted

      if (select item_status from inserted) = 'Available'
      begin
         delete from lostRemovedItems where itemID = @itemID
      end
   end
end
```

The triggers check for an update in the item_status column of the items table. If there is an update, then a temporary table consisting of the record of the row where there is an item_status update, is created. If the the item_status at this instance is 'Removed', **insertIntolostRemoveditems** trigger will trigger an insert operation into the lostRemovedItems table with the content of the temporily created insert table. When the item_status is updated back to 'Available', the **deleteFromLostRemovedItems** trigger checks for such update and creates a temporary insert table where it keeps the record of the row where the updation has happed. If the triggers finds that the item_status in the items table is now 'Available', it will trigger a delete operation from the the lostRemovedItems table where the itemID is the same as the one in the temporary insert table.

As an Example, for the itemID =1, we can see that it is showing available and the lostRemovedItems table is empty because there is no update yet.



If we update the item_status @itemID =1, the status will change to 'Removed' and it will be moved into the lostRemovedItems table as show below

```
267  exec Update_Item_Status @itemID =1 , @newStatus = 'Removed'
268  select top 3 * from items
269  select * from lostRemovedItems
```

100 %

Results | Messages

	itemID	title	category	year_of_publication	ISBN	added_date	item_status
1	1	Artificial Intelligence	Book	2005	234-yut61-88	2023-04-19	Removed
2	2	Big data Analytics	book	2004	898uiu786	2023-04-19	Available
3	3	Physical Anatomy	book	2008	89566ty	2023-04-19	Available

	lostremoveditemID	itemID	lostRemoved_date	ISBN
1	1	1	2023-04-20	234-yut61-88

Inserting into the formerMembers Table

The formerMembers is automatically created when a member in members table becomes inactive. This simply means that a formerMember is as a result of setting the status column in the members table to inactive. This can readily be done with an updating stored procedure and a trigger. When fired, the trigger will perform an insert operation into the formerMembers' table.

The Procedure for Updating member_status

```
create procedure Update_member_Status
    @username nvarchar(10),
    @newStatus nvarchar(10)
as
begin
    update members
    set member_status = @newStatus,
        left_date = case when @newStatus = 'Inactive' then getdate() else left_date end
    where username = @username
end
```

The line 'member_status = @newStatus' is used to update the new status while the line 'left_date = case when @newStatus = 'Inactive' then getdate() else left_date end' is used to update the left_date field. If the new status is active, it changes to the current date, but if it is any other value, it remains what it was before the update.

Trigger for Inserting into the formerMembers table

```
291  create trigger InsertIntoformermembers
292  on members
293  after update
294  as
295  begin
296    if update(member_status) AND EXISTS (select * from inserted where member_status = 'Inactive')
297    begin
298      insert into formerMembers (username, firstname, lastname, addresses, DoB, email, telephone)
299      select username, firstname, lastname, addresses, DoB, email, telephone
300      from deleted;
301    end;
302  end;
```

The trigger checks for an update on member_status in the members table. If there is an update and the exists function which checks if the member_status is set to inactive is true, it will insert the inactive member into formermembers table.

Trigger for deleting from formerMembers table

```
304 create trigger DeleteFromformerMembers
305 on members
306 after update
307 as
308 begin
309     if update(member_status)
310     begin
311         delete from formerMembers
312         where username IN (select username from deleted)
313         and exists (select 1 from inserted where inserted.username = formerMembers.username
314                     and inserted.member_status = 'Active')
315     end
316 end
```

The trigger creates two temporary tables, deleted(that contains old values of the updated rows) and inserted(that contains new values of the inserted rows). The 'if update(member_status)' statement is used to check if the member_status column is updated and if so, the trigger deletes any rows in the formermembers where the username matches a username in the deleted table. It then checks if there is a row in the inserted where the username matches the username in the formerMembers table and the member_status is 'Active'. If such a row exists, it means the member is active and should not be deleted.

Example:

With all the members active, the formerMembers table is empty.

```
319 select top 3* from members
320 select  * from formerMembers
321 exec Update_member_Status @username =??,  @newStatus =??
```

100 %

Results | Messages

	username	Pass_word	firstname	lastname	addresses	DoB	email	telephone	joined_date	left_date	member_status
1	ecg340	0x4070617373776F72640000000000000000000000000000...	Cletus	Loggerman	34 Urmston Drive	1996-12-07	ghtey@yahoo.com	07869564455	2023-04-15	NULL	Active
2	mfm460	0x4070617373776F72640000000000000000000000000000...	Jude	okoye	72 oldhan road Uk	1994-11-04	Jude3@ufoh.com	07867279655	2023-03-09	NULL	Active
3	myh340	0x4070617373776F72640000000000000000000000000000...	Jude	okoye	72 oldhan road Uk	1994-11-04	mike@yahoo.com	07867278655	2023-04-17	NULL	Active

oldmemberID	username	firstname	lastname	addresses	DoB	email	telephone	left_date

empty

Once a member status is updated to inactive, the trigger creates that member in the formerMembers table

```
319  select top 3* from members
320  select  * from formerMembers
321  exec Update_member_Status @username ='ecg340',  @newStatus ='inactive'
```

100 %

Results | Messages

	username	Pass_word	firstname	lastname	addresses	DoB	email	telephone	joined_date	left_date	member_status
1	ecg340	0x4070617373776F72640000000000000000000000000000...	Cletus	Loggerman	34 Urmston Drive	1996-12-07	ghtey@yahoo.com	0786956455	2023-04-15	2023-04-20	inactive
2	mfm460	0x4070617373776F72640000000000000000000000000000...	Jude	okoye	72 oldhan road Uk	1994-11-04	Jude3@ufoh.com	07867279655	2023-03-09	NULL	Active
3	myh340	0x4070617373776F72640000000000000000000000000000...	Jude	okoye	72 oldhan road Uk	1994-11-04	mike@yahoo.com	07867278655	2023-04-17	NULL	Active

	oldmemberID	username	firstname	lastname	addresses	DoB	email	telephone	left_date
1	1	ecg340	Cletus	Loggerman	34 Urmston Drive	1996-12-07	ghtey@yahoo.com	0786956455	2023-04-20

7 . Other Views / Stored

i. Store Procedure for Members Login

Since the members can login online, a procedure can be created for logging in as follows

```
321 create procedure MemberLogin @username nvarchar(10), @password binary(64), @result nvarchar(50) output
322 as
323 begin
324     set nocount on;
325         declare @db_password binary(64)
326     declare @member_status nvarchar(20)
327     select @db_password = Pass_word, @member_status = member_status
328     from members
329     where username = @username
330         if (@db_password IS NOT NULL)
331     begin
332         if (@db_password = @password)
333         begin
334             if (@member_status = 'Active')
335             begin set @result = 'Welcome.'; end
336         else begin set @result = 'Your account is inactive'; end  end
337         else begin set @result = 'Invalid password.'; end end
338         else begin set @result = 'Username does not exist.'; end
339 end
```

it is checks if a member is active and the password is valid before giving access to the member.

ii. Stored procedure for updating the loans table

We can create a stored procedure to update the loans table by inserting the returned date once a loan is returned.

```
create proc returnLoan
    @loanID int
as
begin
    update loans
    set returned_date = getdate()
    where loanID = @loanID
end
```

Basically, it inserts the current date (getdate()) into the returned_date field.

iii. View of all the items in the library

We can get the views of all the items in the library with the code below.

```
499 create view items_details as
500 select a.itemID, a.title, string_agg(b.author_name, ', ') as authors, a.category,
501  a.year_of_publication, a.ISBN, a.added_date, a.item_status
502 from items as a
503 join authors_items as c on a.itemID = c.itemID
504 join authors as b on c.authorID = b.authorID
505 group by a.itemID, a.title, a.category,
506   a.year_of_publication, a.ISBN, a.added_date, a.item_status;
507
508  select * from items_details
```

100 %

Results Messages

	itemID	title	authors	category	year_of_publication	ISBN	added_date	item_status
1	1	Artificial Intelligence	Tim Peter	Book	2005	234-yut61-88	2023-04-19	Removed
2	2	Big data Analytics	John Peter, Nath Mike	book	2004	898uiu786	2023-04-19	Available
3	3	Physical Anatomy	Andrew Nice, Terry Jones	book	2008	89566ty	2023-04-19	Available
4	4	English Syntax	Janet Huds	book	2010	898uity	2023-04-19	Available
5	5	Modern day slavery	The Mirror	Journal	2008	NULL	2023-04-19	Available
6	6	Artificial Intelligen...	Jude Ufoh	Journal	2018	NULL	2021-11-04	Available
7	7	Big Data in Real L...	Nath Topping	DVD	2022	NULL	2022-12-08	Available

Security of the Database

As a security measure, it is important to limit what different users of the database can have access to. This can be achieved with the use of Schema. In this library we can create different user groups and assign them to different schemas. Since the library is particular about loan given out, we can create a **Finance** schema and assign all those incharge of loans and fines to it. we can also create **Main** schema and assign it to general libray staff and a **General** schema for members.

```
511 create schema Main
512  create schema Finance
513  Create schema General
```

The database so far has 9 tables, 11 procedures, 2 views and 1 function. We can assign different database objects to the various schemas as shown in the table below. Once the assigment is done, only people in the assigned schema will have access to the object in the schema. For example, members can only have access to **memberlogin procedure**, **search_item_by_title** procedure and **items_details** view.

S/N	Database Object	Type	Assigned
1	add_items	procedure	Main
2	createLoan	procedure	Main
3	Update_Item_Status	procedure	Main
4	Update_member_Status	procedure	Main
5	Timebeforeduedate	procedure	Main
6	insertaNewMember	procedure	Main
7	updateMember	procedure	Main
8	items	table	Main
9	lostRemovedItems	table	Main
10	authors	table	Main
11	authors_items	table	Main
12	formerMembers	table	Main
23	members	table	Main
13	items_details	view	General
21	MemberLogin	procedure	General
22	search_item_by_title	procedure	General
14	returnLoan	procedure	Finance
15	InsertIntoFineRepayment	procedure	Finance
16	TotalLoansOn	function	Finance
17	loans	table	Finance
18	fines	table	Finance
19	fineRepayment	table	Finance
20	loan_history	view	Finance

```
522  --transfering objects to Main schema
523  ALTER SCHEMA Main TRANSFER dbo.add_items
524  ALTER SCHEMA Main TRANSFER createLoan
525  ALTER SCHEMA Main TRANSFER Update_Item_Status
526  ALTER SCHEMA Main TRANSFER Update_member_Status
527  ALTER SCHEMA Main TRANSFER Timebeforeduedate
528  ALTER SCHEMA Main TRANSFER insertaNewMember
529  ALTER SCHEMA Main TRANSFER updateMember
530  ALTER SCHEMA Main TRANSFER items
531  ALTER SCHEMA Main TRANSFER lostRemovedItems
532  ALTER SCHEMA Main TRANSFER authors
533  ALTER SCHEMA Main TRANSFER authors_items
534  ALTER SCHEMA Main TRANSFER formerMembers
535  ALTER SCHEMA Main TRANSFER members
536  --transfering objects to General schema
537  ALTER SCHEMA General TRANSFER items_details
538  ALTER SCHEMA General TRANSFER MemberLogin
539  ALTER SCHEMA General TRANSFER search_item_by_title
```

```
--transfering objects to General schema
ALTER SCHEMA Finance TRANSFER returnLoan
ALTER SCHEMA Finance TRANSFER InsertIntoFineRepayment
ALTER SCHEMA Finance TRANSFER TotalLoansOn
ALTER SCHEMA Finance TRANSFER loans
ALTER SCHEMA Finance TRANSFER Fines
ALTER SCHEMA Finance TRANSFER fineRepayment
ALTER SCHEMA Finance TRANSFER loan_history
```

Since members can signup online, the **server** can be programmed to accept a member's username and password at the time of signup and use them to create login and password.

```
CREATE LOGIN mfm460
WITH PASSWORD = 'okppg';
```

This can then be used by the database to create a user

```
CREATE USER mfm460 FOR LOGIN mfm460;
GO
```

The user can then be granted access to the General schema of the database

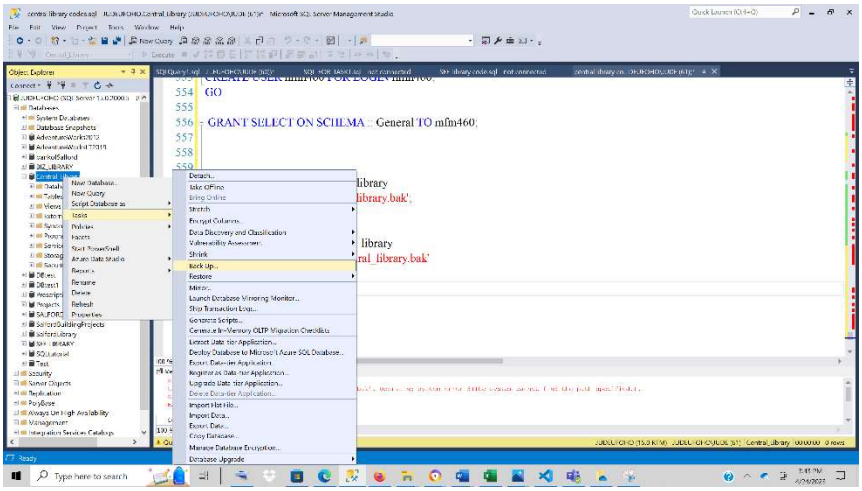
```
GRANT SELECT ON SCHEMA :: General TO mfm460;
```

Backup and Recovery

It is important to regularly create a backup for the libray data so that in the event of accident, the data can be restored. The backup can be saved in a safe disk such as the D drive in the main system server with the following code

```
BACKUP DATABASE central_library
TO DISK = 'D:\backup\central_library.bak';
```

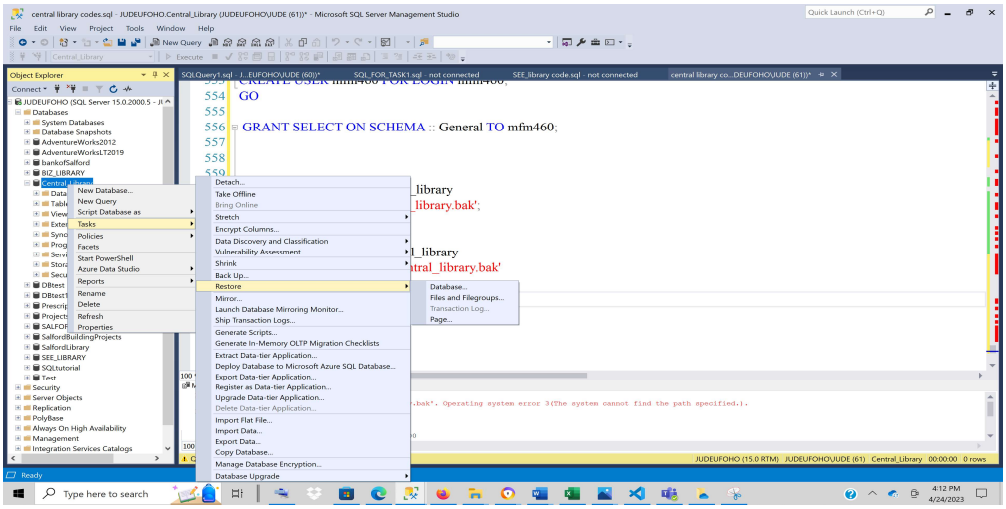
Backup can also be done manualy by clicking on the database, tasks , Backup and following the prompts.



To restore the database, log into the master database on the server then use the following code to restore the database

```
--restoring the database
RESTORE DATABASE central_library
FROM  DISK = 'D:\backup\central_library.bak'
```

It can also be done manually by connecting to an instance of SQL server and create a name for the database. On the created name, right click, point to task and then to restore and follow the prompts.



Conclusion:

The central_library was designed with (3NF) level of database normarlisation. Data integrity has been enforced by the use of constraints, data validation principles and error detection codes. Security of the database has been enforced with the use of password hashing and assignment of different users to different schemas. The database also implements data concurrency and recovery by the use of 'read commited' in creating and updating members and items tables.

TASK 2 REPORT

Introduction

This task requires creation of database tables and importing the provided flat files into the tables. the tables and the combination of views from the table can be used to analyse the data.

Part 1

Relationships

1. **Medical_Practice/Drugs**: Medical_Practice **prescribes** drugs. One drug can be prescribed by many medical practices and one Medical_Practice can predict many drugs. This is **many-to-many**.
2. **Medical_Practice/Prescriptions:** Medical_Practice **makes** prescription. One Medical_Practice can make several prescriptions with different prescription codes. Therefore, the relationship is **one-to-many**.
3. **Drugs/Prescriptions**: Drugs **are prescribed.** One type of drug can appear in different prescriptions. This is a **one-to-many** relationship.

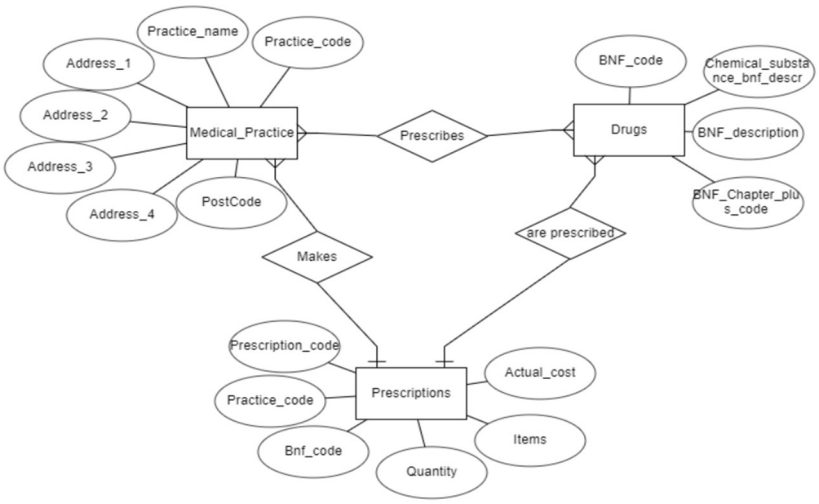


Figure 1: Entity Relationship Diagram

Creation of Database and Tables

Database

The database is created and used in Microsoft SQL Server Management Studio with the name, PrescriptionsDB as shown below.

```
1  create database PrescriptionsDB
2  use PrescriptionsDB
3  go
```

Creation of Medical_Practice Table

The table consists of seven columns. Looking at the data, all the columns contain a combination of number and alphabet, therefore nvarchar data type is used for all. The **Primary key** is the Practice_code because it is unique to each record in the table. It is expected that each practice has a name, address and a postcode. Therefore those columns must not be left null.

```
create table Medical_Practice(
PRACTICE_CODE nvarchar(100) not null primary key,
PRACTICE_NAME  nvarchar(100) not null,
ADDRESS_1  nvarchar(100) not null,
ADDRESS_2  nvarchar(100),
ADDRESS_3  nvarchar(100) ,
ADDRESS_4  nvarchar(100) ,
POSTCODE  nvarchar(100) not null
)
```

Creation of Drugs Table

The table consists of four columns. Since the columns contain the combination of numbers and alphabets and special characters, nvarchar is used as the data type. The Bnf_Code is unique to each record, therefore, it is chosen as the primary key. Finally, it expected that each row has a record therefore all the columns are made to be **not null**

```
create table Drugs(
BNF_CODE nvarchar(50) not null primary key,
CHEMICAL_SUBSTANCE_BNF_DESCR nvarchar(50) not null,
BNF_DESCRIPTION nvarchar(100) not null,
BNF_CHAPTER_PLUS_CODE nvarchar(50) not null
)
```

Creation of Prescriptions Table

The table consists of 6 columns. The prescription code is unique to each row; therefore it is taken as the primary key and it is give the data type of integer since it is made up of consecutive positive whole numbers with zero as the starting number. Both practice_code and BNF_code are foreign keys that reference the Medical_Practice and Drugs table respectively and they retain their data type and characteristics as they were in their respective tables. The Quantity and items are of float datatype and they can be optionally null. Actual_cost is money therefore it is represented with money data type and it is nullable.

```
create table Prescriptions(
PRESCRIPTION_CODE int primary key,
PRACTICE_CODE nvarchar(100) not null
foreign key (PRACTICE_CODE) references Medical_Practice(PRACTICE_CODE),
BNF_CODE nvarchar(50) not null foreign key (BNF_CODE) references drugs(BNF_CODE),
QUANTITY float,
ITEMS float,
ACTUAL_COST money
)
```

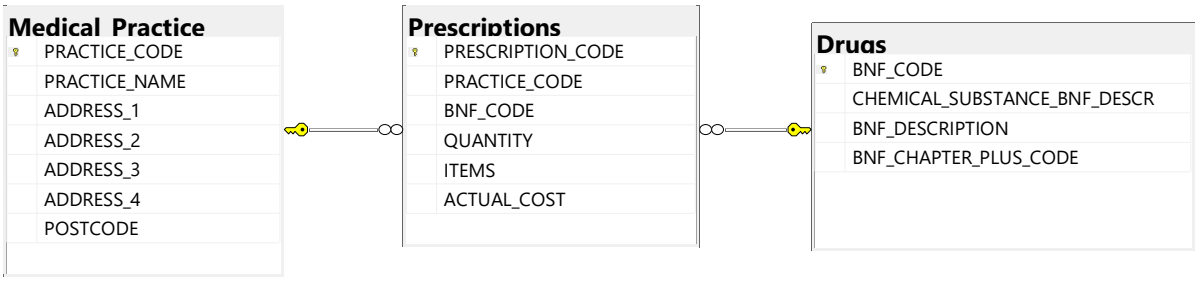
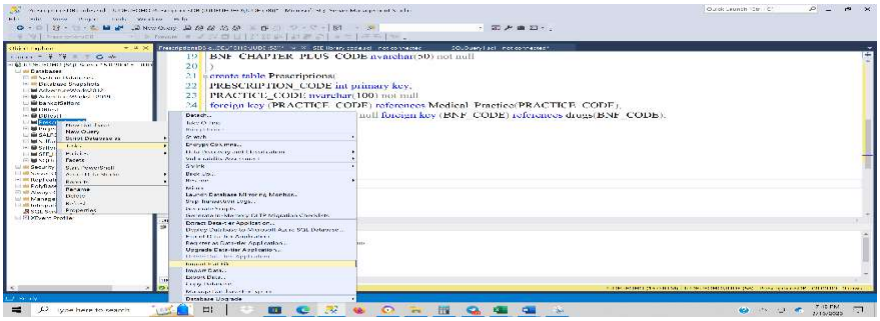


Fig: Database Diagram

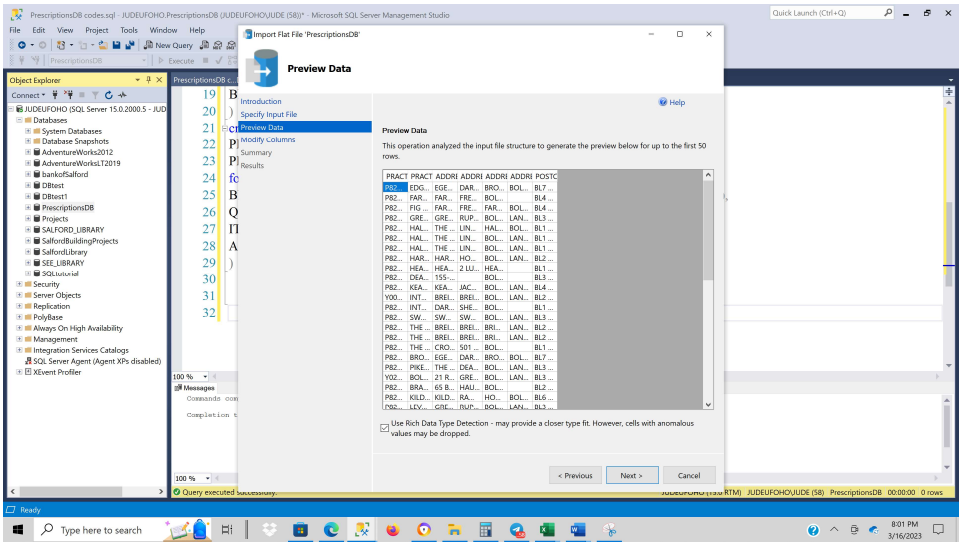
Importing Flat Files into the Database

The data for the assignment were saved in a folder in the computer. The data are Medical_practice.csv, drugs_data.csv and prescriptions_data.csv. These data needed to be imported into the created database so that we can use them to fill up the created tables. The process of importing them is described below.

On the object column we click the database name, point to task, then point to import flat file to import each of the csv files into the database.

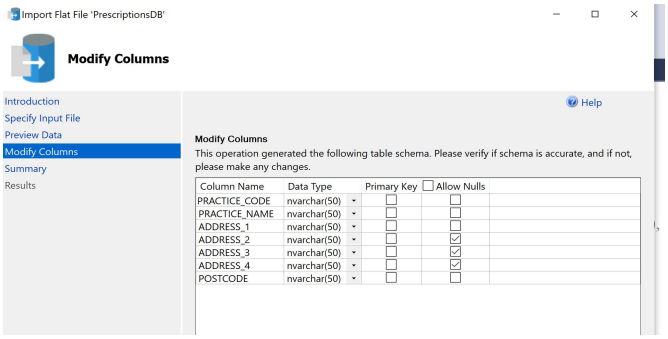


The first step is to specify the input data and this is done by clicking on the browse button on the import flat file wizard and locating the the folder where the csv file in kept. The next step is to preview the data

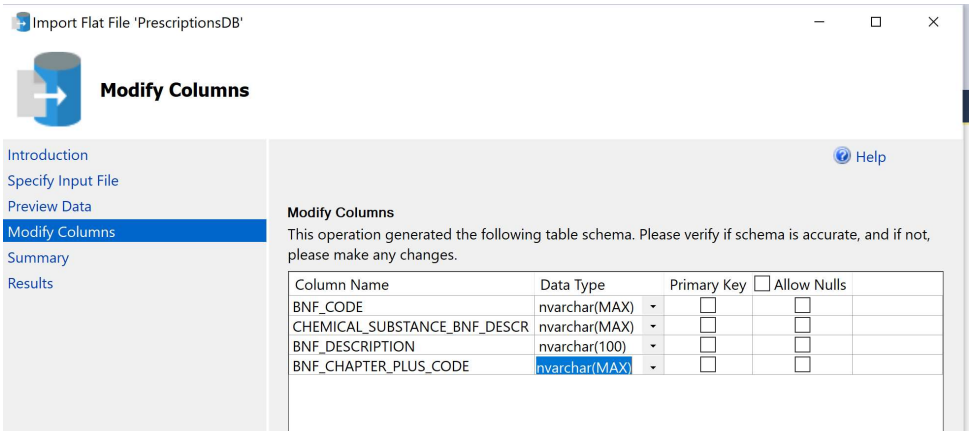


After the data preview, click next to the modify the data. this part is important as it helps to ensure the right data type are imported.

For **Medical_Practice_data**, the data type for the columns are nvarchar

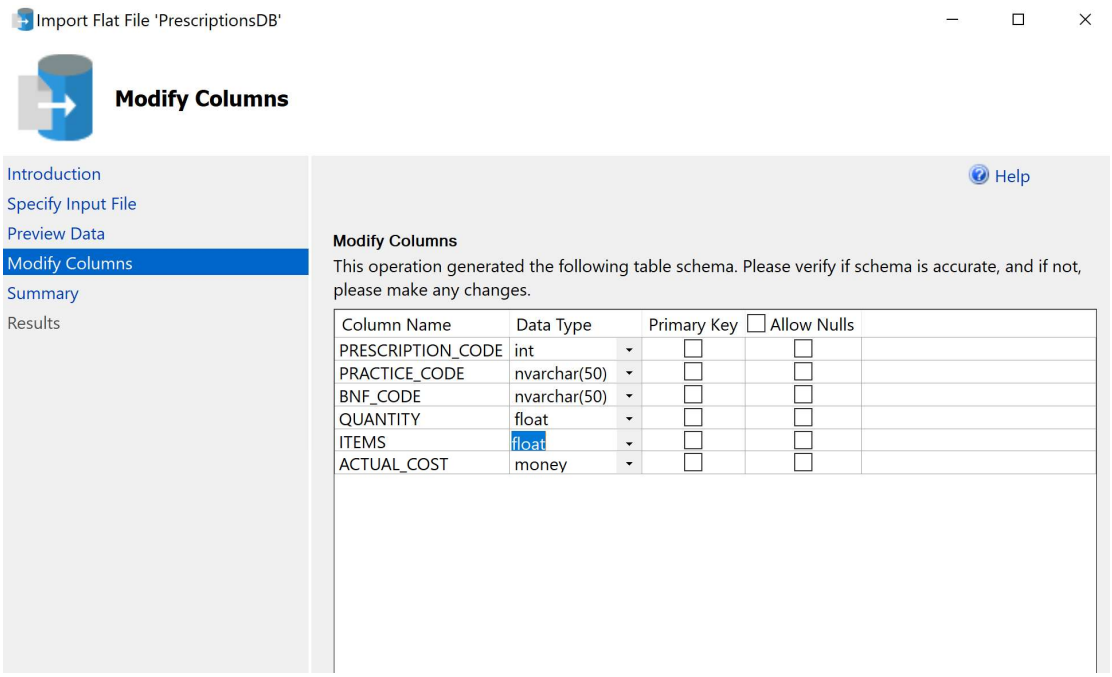


For **drugs_data,** the data type for the all the columns are nvarchar(MAX) because some of the columns contain very long data.



For **Prescriptions_data**, the different columns are assigned the various data types as shown below

The prescription_code is assigned the integer data type. Practice_code and Bnf_code are assigned the nvarchar data type because they contain strings. Quantity and items are assigned the float data type because we could have half or quarter of a pack . Actual_cost is assigned the money data type.



Inserting Data into the Various Tables

Inserting into the Medical_Practice Table

We insert into the medical_practice table from the imported from the inserted medical_practice_data.

```
------putting data into the Medical_Practice
insert into Medical_Practice(Practice_code, Practice_name,address_1,address_2,address_3,address_4,postcode)
select m.Practice_code, m.Practice_name,m.address_1,m.address_2, m.address_3, m.address_4, m.postcode
from Medical_Practice_data as m
```

Filling up the Drugs Table

We insert into the drugs table from the imported drugs_data

```
------putting data into the Drugs
insert into drugs(bnf_code, chemical_substance_bnf_descr,bnf_description, bnf_chapter_plus_code)
select d.bnf_code, d.chemical_substance_bnf_descr,d.bnf_description,d.bnf_chapter_plus_code
from drugs_data as d
```

Filling up the Prescriptions Table

We insert into the prescriptions table from the imported prescriptions_data.

```
------putting data into Prescriptions
insert into Prescriptions(prescription_code, practice_code,bnf_code,quantity,items,actual_cost)
select p.prescription_code, p.practice_code,p.bnf_code, p.quantity,p.items,actual_cost
from prescriptions_data as p
```

After filling the tables, we delete the imported files with the code below.

```
47  ----dropping the data tables
48  drop table prescriptions_data
49  drop table drugs_data
50  drop table Medical_practice_data
```

Part 2

Question2: Query that returns the details of all the drugs which are in the form of tables or capsules.

The query is done on the drugs table. It requires the use the '**like'** operator to match the string that have tablets or capsules as the last word in the BNF_DESCRIPTION column. The code is shown below.

```
52  --Question1| Querry that returns details of drugs in the form of tables or capsules
53  select * from drugs
54  where BNF_DESCRIPTION  like '%' + 'tablets'
55    or    BNF_DESCRIPTION  like '%' + 'capsules'
56
```

100 %

Results | Messages

	BNF_CODE	CHEMICAL_SUBSTANCE_BNF_DESCR	BNF_DESCRIPTION	BNF_CHAPTER_PLUS_CODE
28	0102000P0BCABAB	Mebeverine hydrochloride	Colofac 135mg tablets	01: Gastro-Intestinal System
29	0102000P0BCAEAD	Mebeverine hydrochloride	Colofac MR 200mg capsules	01: Gastro-Intestinal System
30	0102000T0AAAAAA	Peppermint oil	Peppermint oil 0.2ml gastro-resistant caps...	01: Gastro-Intestinal System
31	0102000T0BBAAAF	Peppermint oil	Colpermin gastro-resistant modified-relea...	01: Gastro-Intestinal System
32	0102000T0BCAAAA	Peppermint oil	Mintec 0.2ml gastro-resistant capsules	01: Gastro-Intestinal System
33	0102000Y0AAAFAF	Propantheline bromide	Propantheline bromide 15mg tablets	01: Gastro-Intestinal System
34	0102000Y0BBAAAF	Propantheline bromide	Pro-Banthine 15mg tablets	01: Gastro-Intestinal System
35	0103010D0AAAAAA	Cimetidine	Cimetidine 200mg tablets	01: Gastro-Intestinal System
36	0103010D0AAABAB	Cimetidine	Cimetidine 400mg tablets	01: Gastro-Intestinal System
37	0103010D0AAACAC	Cimetidine	Cimetidine 800mg tablets	01: Gastro-Intestinal System
38	0103010H0AAAAAA	Famotidine	Famotidine 20mg tablets	01: Gastro-Intestinal System
39	0103010H0AAABAB	Famotidine	Famotidine 40mg tablets	01: Gastro-Intestinal System
40	0103010N0AAAAAA	Nizatidine	Nizatidine 150mg capsules	01: Gastro-Intestinal System
41	0103010N0AAABAB	Nizatidine	Nizatidine 300mg capsules	01: Gastro-Intestinal System
42	0103030S0AAAAAA	Sucralfate	Sucralfate 1g tablets	01: Gastro-Intestinal System
43	0103040M0AAAAAA	Misoprostol	Misoprostol 200microgram tablets	01: Gastro-Intestinal System
44	0103050E0AAAAAA	Esomeprazole	Esomeprazole 20mg gastro-resistant tablets	01: Gastro-Intestinal System
45	0103050E0AAABAB	Esomeprazole	Esomeprazole 40mg gastro-resistant tablets	01: Gastro-Intestinal System

Question3. Query to return the total quantity of each prescription rounded to an integer

This requires the use of the floor function to round the product of the quantity and items to a whole number by using the **floor** function. The selection is made from the prescriptions table only since it contains all the required columns.

```
57  --Question3 | Querry that returns the total quantity of each prescription
58  select prescription_code, practice_code, bnf_code, floor(quantity*items) as TotalQuantity
59  from Prescriptions
60
```

100 %

Results Messages

	prescription_code	practice_code	bnf_code	TotalQuantity
1	0	P82034	0205051L0AAABAB	308
2	1	P82034	0205051L0AAABAB	224
3	2	P82034	0205051L0AAACAC	336
4	3	P82034	0205051L0AAACAC	84
5	4	P82034	0205051L0AAACAC	560
6	5	P82034	0205051L0AAADAD	168
7	6	P82034	0205051L0AAADAD	448
8	7	P82034	0205051L0AAADAD	672
9	8	P82034	0205051L0AAADAD	252
10	9	P82034	0205051M0AAAAAA	30
11	10	P82034	0205051M0AAAAAA	28
12	11	P82034	0205051M0AAAAAA	56
13	12	P82034	0205051M0AAABAB	90
14	13	P82034	0205051M0AAAFAF	56
15	14	P82034	0205051M0AAAFAF	84
16	15	P82034	0205051M0AAAFAF	30
17	16	P82034	0205051R0AAAAAA	112
18	17	P82034	0205051R0AAAAAA	112

Question4. Query to return a list of distinct chemical substances which appears in the drug table.

In this case, we make use of the **distinct** key word and the selection is made from the drugs table only

```
61  --Question4. Query to return a list of distinct chemical substances which appears in the drug table.
62  select distinct chemical_substance_bnf_descr as Chemical_Substances
63  from drugs
```

100 %

Results Messages

	Chemical_Substances
1	Acamprosate calcium
2	Acarbose
3	Aceclofenac
4	Acenocoumarol
5	Acetazolamide
6	Acetic acid
7	Acetylcysteine
8	Aciclovir
9	Acipimox
10	Aclidinium bromide
11	Aclidinium bromide/formoterol
12	Acrivastine
13	Adapalene
14	Adapalene and benzoyl peroxide
15	Adhesive Discs/Rings/Pads/Plasters
16	Adhesive Dressing Remover Ster Silicone
17	Adhesive Removers (Sprays/Liquids/Wipes)
18	Adrenaline

Question5. Query that returns the number of prescriptions for each BNF_CHAPTER_PLUS_CODE

In this case, we use the aggregate function **count** in order to get the number of Bnf_Chapter_Plus_Codes from drugs table, **avg** to get the average cost from the Prescriptions table, **max** and **min** to get the maximum and minimum costs from the Prescriptions table. Since two tables are involved, we use the **inner join** to join them on a common column which in this case is **BNF_CODE** and finally group and order them by the BNF_CHAPTER_PLUS_CODE

```
66  --Question5. Query that returns the number of prescriptions for each BNF_CHAPTER_PLUS_CODE
67  select a. BNF_CHAPTER_PLUS_CODE, count(a.BNF_CHAPTER_PLUS_CODE) as
68  No_of_Prescriptions, avg(b. actual_cost) as Average_Cost,
69  max(b.actual_cost) as Max_Cost, min(b.actual_cost) as Min_Cost
70  from drugs as a inner join Prescriptions as b on a. BNF_CODE = b. BNF_CODE
71  group by a. BNF_CHAPTER_PLUS_CODE
72  order by  a. BNF_CHAPTER_PLUS_CODE
```

100 %

Results | Messages

	BNF_CHAPTER_PLUS_CODE	No_of_Prescriptions	Average_Cost	Max_Cost	Min_Cost
1	01: Gastro-Intestinal System	8777	35.1253	2777.69	0.14
2	02: Cardiovascular System	19186	35.9956	11094.75	0.13
3	03: Respiratory System	7057	75.873	4161.81	0.15
4	04: Central Nervous System	28866	28.3993	2765.62	0.13
5	05: Infections	4657	21.2475	1262.21	0.20
6	06: Endocrine System	12462	61.1662	3490.70	0.17
7	07: Obstetrics, Gynaecology and Urinary-Tract Di...	3999	25.4563	803.49	0.15
8	08: Malignant Disease and Immunosuppression	754	54.9382	2197.16	0.29
9	09: Nutrition and Blood	7944	41.8157	4250.44	0.14
10	10: Musculoskeletal and Joint Diseases	3634	16.2458	1476.64	0.27
11	11: Eye	2676	21.4404	434.80	1.34
12	12: Ear, Nose and Oropharynx	1274	23.6805	315.78	1.68
13	13: Skin	5692	22.2507	1402.48	0.49
14	14: Immunological Products and Vaccines	126	1049.5571	21570.92	7.29
15	15: Anaesthesia	271	35.6664	345.25	0.28
16	19: Other Drugs and Preparations	338	32.2493	1193.35	0.43
17	20: Dressings	1014	40.2213	2599.19	0.22
18	21: Appliances	5856	38.353	2552.92	0.21
19	22: Incontinence Appliances	535	42.8652	538.89	2.87
20	23: Stoma Appliances	1697	82.2616	882.53	1.59

Question6. Query that returns the most expensive prescription by each practice for prescriptions that are more than £4000.

For each medical practice, the most expensive prediction will be maximum actual_cost which we can get by using aggregate function **max** on the actual_cost column on the prescriptions table and **inner joining** Medical_Practice and Prescriptions table on practice code which is common to both. Since we are considering all the maximum costs above £4000, we use the **group by** clause to group them according to the practice_name and **order by** clause to arrange them in order of magnitude. **desc** is used to ensure that the highest comes first

```
74  --Question6. Query that returns the most expensive prescription by each practice for prescriptions that are more than £4000.
75  select a.practice_name, max(b.actual_cost)as most_expensive_prescription
76  from Medical_Practice as a inner join prescriptions as b on a. practice_code = b.practice_code
77  where actual_cost >4000
78  group by a.practice_name
79  order by most_expensive_prescription desc
```

100 %

Results | Messages

	practice_name	most_expensive_prescription
1	UNSWORTH GROUP PRACTICE	21570.92
2	BROMLEY MEADOWS SURGERY	11031.59
3	KILDONAN HOUSE	8659.60
4	KEARSLEY MEDICAL CENTRE	7313.58
5	BOLTON COMMUNITY PRACTI...	6069.77
6	HARWOOD MEDICAL CENTRE	4626.96
7	MANDALAY MEDICAL CENTRE	4377.65
8	CROMPTON VIEW SURGERY	4250.44
9	DALEFIELD SURGERY	4031.96

Question7 Further Queries

a. The drugs most prescribed based on total quantity

```
86  select a.bnf_description, sum(b.quantity) as Total_Quantity from
87  drugs as a
88  inner join Prescriptions as b on a.BNF_CODE = b.BNF_CODE
89  group by a.bnf_description
90  order by total_quantity desc
91
```

100 %

Results | Messages

	bnf_description	Total_Quantity
1	Ensure Plus milkshake style liquid (9 flavours)	1448600
2	Jevity 1.5kcal liquid	778000
3	Ensure Compact liquid (4 flavours)	776500
4	Ensure TwoCal liquid (4 flavours)	488800
5	Ensure Plus Juce liquid (6 flavours)	370480
6	PaediaSure Plus fibre liquid (4 flavours)	316800
7	Fortisip Bottle (8 flavours)	310000
8	PaediaSure Plus liquid (3 flavours)	297800
9	Ensure Plus Fibre liquid (5 flavours)	256600
10	Jevity Plus liquid	236000
11	Fortisip Compact Protein liquid (9 flavours)	230000

We select the drug name from drugs and the sum of quantity as total_quantity from the prescriptions table and join them on common bnf_code, group them according to the drugs name and order them by the total_quantity.

b. The top 5 most expensive drugs based on cost_per_items

```
92  --top 5 most expensive drugs
93  select top 5 a.bnf_description, max(floor(b.actual_cost/(b.quantity*b.items))) as Cost_per_item from
94  drugs as a
95  inner join Prescriptions as b on a.BNF_CODE = b.BNF_CODE
96  group by a.bnf_description, a.BNF_CODE
97  order by cost_per_item desc
```

100 %

Results | Messages

	bnf_description	Cost_per_item
1	Lanreotide 120mg/0.5ml inj pre-filled syringes	876.00
2	Saizen 20mg/2.5ml solution for injection cartrid...	433.00
3	Norditropin NordiFlex 15mg/1.5ml inj pf pens	325.00
4	Norditropin FlexPro 15mg/1.5ml inj pre-filled pens	298.00
5	Navina Smart System	294.00

We select the bnf_description and calculate the cost_per_item as actual cost divided by the product of quantity and items from prescription table and join them on bnf_code, group by bnf_description and order by cost_per_item. ‘Max’ is used to select the highest value of each drug cost_per_item while ‘floor’ is used to round the value to a whole number.

c. Count of prescription, total quantity and total cost for each medical Practice

```
100  -- count of  prescription, total quantity and total_cost by each medical practice
101  select a.practice_code, a.practice_name, count(a.practice_code) as num_of_prescriptions,
102  sum(b.quantity*b.items) as Total_Quantity, sum(b.actual_cost) as Total_Cost
103  from Medical_Practice a
104  inner join prescriptions b on a.PRACTICE_CODE=b.PRACTICE_CODE
105  group by a.practice_code, a.practice_name
106  order by Total_Quantity desc
```

100 %

Results Messages

	practice_code	practice_name	num_of_prescriptions	Total_Quantity	Total_Cost
1	P82015	UNSWORTH GROUP PRACTICE	4977	3248492	333909.68
2	Y03079	BOLTON COMMUNITY PRACTICE	4355	2299676	227230.72
3	P82008	STONEHILL MEDICAL CENTRE	3920	2039421	190890.80
4	P82007	KEARSLEY MEDICAL CENTRE	4189	1974334	202286.60
5	P82003	KILDONAN HOUSE	3821	1564774	193615.87
6	P82031	HEATON MEDICAL CENTRE	3377	1513651	150512.05
7	P82016	HARWOOD MEDICAL CENTRE	3676	1491050	170756.00
8	P82001	THE DUNSTAN PARTNERSHIP	3695	1477656	158931.70
9	P82004	SWAN LANE MEDICAL CENTRE	3058	1435293	121368.48
10	P82006	DR MALHOTRA & PARTNERS	3479	1427642	141759.98
11	P82002	PIKES LANE 1	3173	1343323	117979.97
12	P82010	DALEFIELD SURGERY	2850	1260821	122959.92
13	P82009	ST HELENS ROAD PRACTICE	2879	1201573	108584.58
14	P82023	MANDALAY MEDICAL CENTRE	2783	1164737	132516.10
15	P82005	STABLE FOLD SURGERY	2656	1140167	102223.70

The aggregate functions **count** and **sum** were respectively used to get the count of practice_code and sum of total_quantity and actual_cost for each medical_practice in prescriptions. the result was joined to the practice_code and name in the the medical_practice table, grouped by practice_code and practice_name and ordered by total_quantity.

d. The number of medical_practice in each postal district

In UK, the first part of a post code is the postal district. we can use the **string_split** function to split each postcode at the space delimiter and filter out the first part (which is having BL) using the 'where' clause and 'like' operator and set it to value. We then take the count of the distinct values.

```
108  --number of medical practice in each postal district
109  SELECT  value as Postal_District, count(value) as No_of_Medical_Practice
110  FROM Medical_Practice
111  CROSS APPLY STRING_SPLIT(postcode, ' ')
112  WHERE value  LIKE '%BL%'
113  group by value
```

100 %

Results Messages

	Postal_District	No_of_Medical_Practice
1	BL1	22
2	BL2	7
3	BL3	19
4	BL4	5
5	BL5	2
6	BL6	2
7	BL7	3

e. Medical_Practice and the drugs they prescribed most

```
--medical practice and the drugs they prescribed most
with count_of_prescription  as (
  select p.practice_code, d.bnf_code, count(*) AS prescriptions
  from Prescriptions p
  JOIN Drugs d on d.bnf_code = p.BNF_CODE
  group by p.practice_code, d.bnf_code),
max_count_of_prescription as (
  select practice_code, max(prescriptions) as max_prescriptions
  from count_of_prescription
  group by practice_code)
select a.practice_name as MedicalPractice, b.bnf_description as MostPrescribedDrug,
j.prescriptions as CountOfMostPrescribedDrug
from Medical_Practice a
INNER JOIN Prescriptions c on a.practice_code = c.practice_code
INNER JOIN Drugs b on b.bnf_code = c.BNF_CODE
INNER JOIN count_of_prescription as j on j.practice_code = a.practice_code AND j.bnf_code = b.bnf_code
INNER JOIN max_count_of_prescription as k on k.practice_code = a.practice_code AND k.max_prescriptions = j.prescriptions
group by a.practice_name, b.bnf_description, j.prescriptions;
```

100 %

Results | Messages

	MedicalPractice	MostPrescribedDrug	CountOfMostPrescribedDrug
1	3D MEDICAL CENTRE	Folic acid 5mg tablets	7
2	AL FAL MEDICAL GROUP	Zapain 30mg/500mg tablets	12
3	BARDOC GP OOH	Prednisolone 5mg soluble tablets	12
4	BOLTON HOSPICE	Clonazepam 500microgram tablets	1
5	BRADFORD STREET SURGERY	Folic acid 5mg tablets	9
6	DEANE CLINIC 1	Metformin 500mg tablets	10

We first create a view of the prescriptions and their count for each medical_practice and bnf_code as **count_of_prescription**. From this view, we create another view called the **max_count_of_prescription** which has practice_code and the maximum prescription for each. We then select practice_name from medical_practice table, bnf_desription from drugs table and join them to count_of_prescription view using practice_code and bnf_code respectively and finally join the result to max_count_of_prescription view on practice_code and max_prescription in the count_of_prescription view. The result is the name of each medical_practice , the drug they predicted most and the number of prescriptions made on that drug.

Conclusion:

The database PrescriptionDB was designed by creating three tables. The tables were filled with given dataset. With the database, more insights can be gained on the data.