[Group 20] Coursework 2

College Library Database Management System

ECS740P - Database Systems

Anthony Stockman

Queen Mary
University of London

[Group 20]
| Vahesian Logan Vasudeva | David Haunschild | Edward Monah | Mohsen Razvi |

Disclaimer

This library database is cloud based and accessible to the grader directly over the command prompt with a running MySQL installation using following command:

```
mysql -h coursework-2[redacted].amazonaws.com -P 3306 -u [redacted] -p
```

To select the respective database:

```
use coursework_2;
```

Further information for connecting to the AWS RDS instance:
Hostname: coursework-2[redacted].amazonaws.com
Database: coursework_2
Username: [redacted]
Password: [redacted]
Port: [redacted]

The grader user will have the following access rights:

```
SELECT, INSERT, UPDATE, DELETE, CREATE, DROP
```

This submission is also accompanied with the respective script files to fully create and populate the database with the respective data as demonstrated in this report.

Table of Contents

Introduction

This paper aims to implement the design of a database system for a college library based on the relational schema produced in Coursework 1 and accepted database design principles to fulfil the requirements for Coursework 2 for the module ECS740P - DATABASE SYSTEMS, and the following scenario:

"A college library provides various resources for students and staff, including books, videos, DVDs and CDs. It is common that several copies are kept of some resources, for example, recommended books for courses. The usual loan period of a resource is 2 weeks, but some resources are available for short loan only (2 days) and some other resources can only be used within the library. The library consists of 3 floors. Resources are stored in the library on shelves. To locate a specific item in the library a combination of floor number and shelf number are used. In addition to this, a class number system is used to identify in which subject area a particular item belongs, for example all resources concerned with Database Systems will have the same class number. Students hold library cards which identify them as valid members of the Library. Students can loan a number of different resources at one time, but the total number of resources they may borrow at a given time must never exceed 5. Staff members at the College also hold library cards and are allowed to loan up to 10 different items at one time. The library charges fine for resources that are loaned for longer than the time allowed for that resource. For each day a resource is overdue the member is fined one dollar. When the amount owed in fines by a member is more than 10 dollars, that member is suspended until all resources have been returned and all fines paid in full" (Stockman, 2020).

For the purpose of this coursework a MySQL database server was created on a AWS (Amazon Web Services[1]) located in Paris, France. A collaborative SQL query editor, 'PopSQL'[2], was used. Each group member received a cloud-based shared root-login database access. The queries were shared and synced throughout the coursework and data was generated for testing purposes with Python algorithms, a web-based service called "Mockaroo"[3] and some data was generated manually.

Relational Database Schema

Most of the normalised relational schema from Coursework 1 is found below. Primary keys are underlined while foreign keys are italicised.

Member (Member_ID, Email_Address, First_Name, Last_Name, House_Number or Name, Postcode, Phone_Number, Member_Type, DOB)
Class_Number (Class_ID, Subject, *Dept_No*)
Dept_Number (Dept_No, Department_Name)
Location (Location_ID, Shelf_No, Floor_No)
Country (Country_ID, Country_Name)
Resource (Resource_ID, Borrowable_Time, Type, No_of_Total_Copies, *Location_ID, Class_ID)*
DVD/Video (Resource_ID, DVD/Video, *Film_ID*),

[1] https://aws.amazon.com/rds/mysql/
[2] https://popsql.com/
[3] https://www.mockaroo.com/

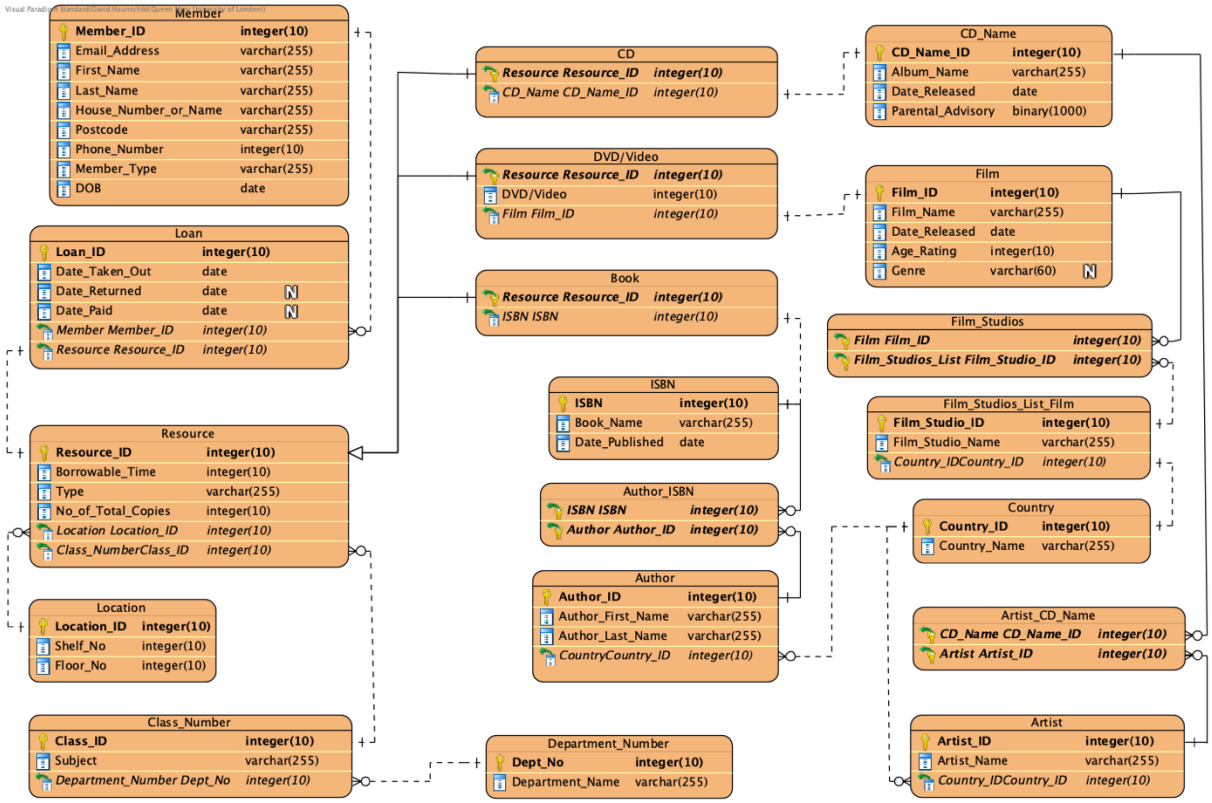
Film_Studio_List_Film (*Film_ID*, *Film_Studio_ID*)
Film_Studios_List (Film_Studio_ID, Film_Studio_Name, *Country_ID*)
CD (Resource_ID, *CD_Name_ID*)
CD_Name(CD_Name_ID, Album_Name, Date_Released, Parental Advisory)
CD_Artist (*CD_Name_ID*, *Artist_ID*)
Artist (Artist_ID, Artist_Name, *Country_ID*)
Book (Resource_ID, *ISBN*)
ISBN (ISBN, Book_Name, Date_Published*)*
Book_Author (*ISBN*, *Author_ID*)
Author (Author_ID, Author_First_Name, Author_Last_Name, *Country_ID*)

For the "Film" relation, it was decided to add a "Genre" attribute as this would very likely be a search criteria by a library user.

Film (Film_ID, Film_Name, Genre, Date_Released, Age_Rating)

The database will be created in MySQL using this schema, which is visualized below again from the first coursework.

Relational Database Schema



Source: Created with Visual Paradigm by the authors of this document.

Create Tables Script

The tables were created in the following order to ensure that foreign key constraints could be fulfilled. Unsigned auto-incrementing primary keys were used throughout, with the integer type selected based on the number of rows the table was likely to have.[4] The full script, with appropriate comments can be found in the submission as a separate SQL file.

Member Table

```
CREATE TABLE Member (
    Member_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k members, more than
enough*/
    First_Name varchar(40) NOT NULL,
    Last_Name varchar(40) NOT NULL,
    /* more than enough max length*/
    Email_Address varchar(255) NOT NULL UNIQUE,
    House_Number_or_Name varchar(30),
    /* more than enough max length*/
    Postcode varchar(12),
    /* more than enough max length*/
    Phone_Number varchar(15) NOT NULL UNIQUE,
    /*should store numbers as string characters |SHALL NOT BE NULL, AS IT
IS VERIFICATION METHOD*/
    Member_Type varchar(7) NOT NULL,
    /*max length of options*/
    DOB date NOT NULL,
    CHECK (
        (Email_Address like '%@%')
        AND (Email_Address like '%.%')
        AND (Email_Address like '%qmul.ac.uk')
    ),
    /* Checks email address has @, ., qmul.ac.uk in it (only QMUL
addressses allowed) */
    CHECK (Member_Type in ('Student', 'Staff'))
);
/*The unsigned range is 0 to 65535. SMALL INT*/
```

The primary key, ‘Member_ID’, is an auto-incrementing unsigned SMALLINT datatype with a display width of 5. This allows over sixty-five thousand members to be stored. This is more than enough for a library of our size. The email address, first and last names are stored as VARCHARs with a reasonable maximum length. The aforementioned three attributes cannot be null (it is a reasonable expectation for prospective members to give this data as contact details), and furthermore, there are some checks to ensure the email address given is unique and the member’s university email address. It is assumed this library is for QMUL, therefore the check is for an email address that ends in ‘qmul.ac.uk’. The phone number is stored as a string VARCHAR (as opposed to an INTEGER datatype) to prevent leading zeros from being removed and as no numerical manipulation will be done with the phone number. It must be unique and not null as it will be used as a verification method and a second form of contact. The member can choose to give their post code and house number/name (both stored as VARCHARs) which can then be used to derive the address, but this is not deemed necessary for library membership. The member type, stored as a VARCHAR, cannot be null

[4] Please refer to https://dev.mysql.com/doc/refman/8.0/en/integer-types.html to see the maximum ranges of integer datatypes in MySQL

and a further check is used to ensure that this is set to either ‘Student’ or ‘Staff’. The date of birth (‘DOB’) must be stored to enforce age-restrictions on items.

Country Table

```
REATE TABLE Country (
    Country_ID TINYINT(3) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*range is 0 to 255. so max 255, more than enough*/
    Country_Name varchar(50) NOT NULL UNIQUE
    /* more than enough max length*/
 );
```

The unsigned auto-incrementing primary key, ‘Country_ID’, is a TINYINT datatype as there are less than 250 countries recognised countries. The name is stored as a VARCHAR with a sufficient max length.

Location Table

```
CREATE TABLE Location (
    Location_ID TINYINT(3) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*range is 0 to 255. so max 255, more than enough*/
    Shelf_No TINYINT(3) UNSIGNED NOT NULL,
    /*range is 0 to 255. so max 255, more than enough*/
    Floor_No TINYINT(1) UNSIGNED NOT NULL,
    CHECK (Floor_No in (0, 1, 2))
    /* Ground is 0, 1st is 1, 2nd is 2  Only 3 floors in library as per
spec*/
);
```

The location numbering system will initially be fifty shelves on each of the three floors. These shelves are assigned a number and a map would be produced for use by library staff. By storing location information in this way, the attributes can be stored as unsigned TINYINTs. A check is made to ensure that the floor number is one of ‘0’, ‘1’ or ‘2’ , corresponding to the ‘Ground floor’, ‘First floor’ and ‘Second floor’ respectively. The database has capacity for over 250 locations, so more shelves can be added onto each floor if necessary.

Department and Class Tables

```
CREATE TABLE Department_Number (
    Dept_No TINYINT(3) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*range is 0 to 255. so max 255, more than enough*/
    Department_Name varchar(50) NOT NULL UNIQUE
    /* more than enough max length*/
);

CREATE TABLE Class_Number (
    Class_ID TINYINT(3) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*range is 0 to 255. so max 255, more than enough*/
    `Subject` varchar(50) NOT NULL UNIQUE,
    /* more than enough max length*/
    Dept_No TINYINT(3) UNSIGNED,
    FOREIGN KEY (Dept_No) REFERENCES Department_Number(Dept_No) ON DELETE
SET NULL
    /*SET NULL IF ASSOCIATED DEPARTMENT IS DELETED*/
);
```

Unsigned auto-incrementing TINYINT primary keys ensure that over 250 departments and subjects can be stored. This is more than enough for a university library. The 'Subject' and the 'Department_Name' are VARCHARs of a sufficient length and cannot be null. A foreign key constraint is made as per the normalised relational schema. If the parent row (department) is deleted, the department for that subject will be set to null. This ensure that the 'Subject' information is not lost and can be reassigned to another department, if necessary. As Dept_No is an auto-incrementing primary key, an ON UPDATE CASCADE is not necessary as one would not expect this value to ever change.

Resource Table

```
CREATE TABLE Resource (
    Resource_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k resources, more than
enough*/
    Borrowable_Time TINYINT(2) UNSIGNED NOT NULL,
    `Type` varchar(12) NOT NULL,
    /* more than enough max length*/
    No_of_Total_Copies TINYINT(3) UNSIGNED NOT NULL,
    /*range is 0 to 255. so max 255, more than enough*/
    Location_ID TINYINT(3) UNSIGNED,
    FOREIGN KEY (Location_ID) REFERENCES Location(Location_ID) ON DELETE
SET NULL,
    Class_ID TINYINT(3) UNSIGNED,
    FOREIGN KEY (Class_ID) REFERENCES Class_Number(Class_ID) ON DELETE SET
NULL,
    CHECK (Borrowable_Time in (0, 2, 14)),
    /* NUMBER OF DAYS BOOK CAN BE BORROWED FOR */
    CHECK (`Type` in ('Book', 'CD', 'DVD/VIDEO'))
    /* Three options, DVD/Video distinction will be made later*/
);
```

The primary key, 'Resource_ID', is an auto-incrementing unsigned SMALLINT datatype with a display width of 5. This allows over sixty-five thousand resources to be stored, and this should be more than sufficient. 'Borrowable_Time' in days is stored as a TINYINT, and there is a check to ensure that this is either '0', '2' or '14' as these are the borrowable times as per the spec. A borrowable time of '0' indicates the item is a reference item. An unsigned TINYINT attribute 'No_of_Total_Copies' assumes a maximum of 255 copies per resource, which is a reasonable assumption. The 'Type' attribute indicates whether the resource is a 'Book', a 'CD' or a 'DVD/VIDEO'. N.B. the distinction between a DVD and Video will be made later. A check constraint ensures that 'Type' is one of those option. The aforementioned attributes cannot be null as the library needs to know this information. The foreign keys, 'Location ID' and 'Class ID' can be null, for example in situations such as a new resource not yet assigned to a location or subject. Furthermore, if a location or subject from those tables were to be deleted, these values would be set to null. As the foreign keys are auto-incrementing primary keys of other tables, no ON UPDATE CASCADE is necessary as one would not expect this value to ever change.

Loan Table

```
CREATE TABLE Loan (
    Loan_ID MEDIUMINT(10) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /* THE unsigned range is 16777215, more than enough */
    Date_Taken_Out date NOT NULL,
    Date_Returned date,
    Date_Paid date,
    Member_ID SMALLINT(5) UNSIGNED,
    FOREIGN KEY (Member_ID) REFERENCES Member(Member_ID) ON DELETE SET
NULL,
    /* IF MEMBER DELETED LOAN INFO STILL STORED, USEFUL FOR POPULARITY OF
RESOURCE */
    Resource_ID SMALLINT(5) UNSIGNED,
    FOREIGN KEY (Resource_ID) REFERENCES Resource(Resource_ID) ON DELETE
SET NULL,
    /* IF RESOURCE DELETED LOAN INFO STILL STORED, USEFUL FOR MEMBER LOANS
AND FINES DETAILS */
    CHECK (Date_Returned >= Date_Taken_Out) /*CAN RETURN ON SAME DAY or
after*/
);
```

The primary key, ‘Loan_ID’, is an auto-incrementing unsigned MEDIUMINT datatype with a display width of 10. This allows over sixteen million loans to be added, and this should be more than sufficient for years of database operation. This database stores one loan for one member for one item (i.e. if a member loaned three items, this would correspond to three loans).

‘Date_Taken_Out’ cannot be null as a loan must store this information. ‘Date_Returned’ is initially null, until a resource is returned. ‘Date Paid’ refers to the payment date of a fine for that particular loan, if necessary. A check is made to check that the ‘Date Returned’ is after on on the ‘Date_Taken_Out”, while there are foreign key constraints to the ‘Member’ table and the ‘Resource’ table. These are set to ON DELETE SET NULL, as, when either parent row is deleted, it would still be useful to keep the loan information. In the case of a member being deleted, it would be useful to keep the loan information to see the popularity of the resource, while if a resource was deleted, it would be useful to see any fines owed. As the foreign keys are auto-incrementing primary keys of other tables, no ON UPDATE CASCADE is necessary as one would not expect this value to ever change.

CD Name Table

```
CREATE TABLE CD_Name (
    CD_Name_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k, more than enough*/
    Album_Name varchar(120) NOT NULL,
    Date_Released date,
    Parental_Advisory CHAR(1) NOT NULL,
    /*Y OR N FOR YES OR NO */
    CHECK (Parental_Advisory in ('Y', 'N'))
);
```

Over sixty-five thousand CDs can be stored in this table, with the name, the date released (if known) and parental advisory information stored. A check is present to ensure the parental advisory information is in the correct form.

Film Table

```
CREATE TABLE Film (
    Film_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k, more than enough*/
    Film_Name varchar(60) NOT NULL,
    Date_Released date,
    Genre varchar(60) NOT NULL,
    Age_Rating VARCHAR(3) NOT NULL, /* STORED AS A STRING*/
    CHECK (
        Genre in (
            'TV Show', 'Thriller', 'Sports Movie',
            'Sci-fi', 'Romantic Movie', 'Music',
            'Horror', 'Foreign Movie', 'Gay & Lesbian Movie',
            'Drama', 'Crime',
            'Documentary', 'Cult Movie', 'Comedy',
            'Children & Family', 'Anime', 'Action',
            'Foreign', 'Animation','Action and Adventure'
        )
    ),/* Netflix Genres*/
    CHECK (Age_Rating in ('U', 'PG', '12', '15', '18'))
    /* UK AGE RATINGS*/
);
```

Over sixty-five thousand films and TV shows can be stored in this table, with the name, the date released (if known), genre and age rating information stored. A check is present to ensure that the age rating is one of the British age ratings, while the genre is the genre stored by Netflix for that particular movie/TV show. (Only one genre per film is stored).

ISBN

```
CREATE TABLE ISBN (
    ISBN VARCHAR(13) PRIMARY KEY,
    /* ISBN STORED AS A STRING WITH DASHES/SPACES REMOVED, ISBN CAN BE 10
OR 13 DIGITS */
    Book_Name varchar(255) NOT NULL,
    /* more than enough max length*/
    Date_Published date,
    CHECK(ISBN REGEXP '^[0-9]+$')
    /*CHECKS ISBN ONLY CONTAINS DIGITS*/
);
```

The primary key, ‘ISBN’, is stored as a string VARCHAR to ensure that leading zeros are not removed, and as no numeric calculation will be done. It will be stored with dashes and spaces removed and there is a check to ensure the ISBN only contains digits.

Author, Artist and Film Studios List Tables

```
CREATE TABLE Author (
    Author_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k, more than enough*/
    Author_First_Name varchar(50),
    /* more than enough max length*/
    Author_Last_Name varchar(50) NOT NULL,
    /* more than enough max length*/
    Country_ID TINYINT(3) UNSIGNED,
    FOREIGN KEY (Country_ID) REFERENCES Country(Country_ID) ON DELETE SET
NULL
    /*IF COUNTRY DELETED, SET TO NULL*/
);

CREATE TABLE Artist (
    Artist_ID SMALLINT(5) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*The unsigned range is 0 to 65535. so max 65k, more than enough*/
    Artist_Name varchar(75) NOT NULL,
    /* more than enough max length*/
    Country_ID TINYINT(3) UNSIGNED,
    FOREIGN KEY (Country_ID) REFERENCES Country(Country_ID) ON DELETE SET
NULL
    /*IF COUNTRY DELETED, SET TO NULL*/
);

CREATE TABLE Film_Studios_List (
    Film_Studio_ID TINYINT(3) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    /*range is 0 to 255. so max 255, more than enough*/
    Film_Studio_Name varchar(75) NOT NULL,
    /* more than enough max length*/
    Country_ID TINYINT(3) UNSIGNED,
    FOREIGN KEY (Country_ID) REFERENCES Country(Country_ID) ON DELETE SET
NULL
    /*IF COUNTRY DELETED, SET TO NULL*/
);
```

Over sixty-five thousand records can be stored in each table, with their name and nationality (or in the case of film studio, the country of origin). As some musical artists are mononymous or bands, this data will be stored as ‘Artist_Name’ as opposed to ‘First_Name’ and ‘Last_Name’. On the unlikely occasion the foreign key ‘Country_ID’ is deleted, the ‘Country_ID’ will be set to null so the other information is still stored. As it is also the auto-incrementing primary key of another tables, no ON UPDATE CASCADE is necessary as one would not expect this value to ever change.

CD, DVD_Video and Book Tables

```
CREATE TABLE CD (
    Resource_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Resource_ID) REFERENCES Resource(Resource_ID) ON DELETE
CASCADE ,
    CD_Name_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (CD_Name_ID) REFERENCES CD_Name(CD_Name_ID) ON DELETE
CASCADE
    /*ROW DELETED IF FK DELETED */
);

CREATE TABLE DVD_Video (
    Resource_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Resource_ID) REFERENCES Resource(Resource_ID) ON DELETE
CASCADE,
    /*ROW DELETED IF FK DELETED */
    `DVD/Video?` VARCHAR(6) NOT NULL,
    /* more than enough max length*/
    Film_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Film_ID) REFERENCES Film(Film_ID) ON DELETE CASCADE,
    /*ROW DELETED IF FK DELETED */
    CHECK(`DVD/VIDEO?` IN ('DVD', 'VIDEO'))
);

CREATE TABLE Book (
    Resource_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Resource_ID) REFERENCES Resource(Resource_ID) ON DELETE
CASCADE,
    /*ROW DELETED IF FK DELETED */
    ISBN VARCHAR(13) NOT NULL,
    FOREIGN KEY (ISBN) REFERENCES ISBN(ISBN) ON UPDATE CASCADE ON DELETE
CASCADE
    /* IF ISBN UPDATED OR DELETED THE ROW WILL CHANGE OR BE DELETED*/
);
```

The above tables are combined in the ‘Resource_ID’ table with either ‘CD_Name_ID’, ‘Film_ID’ and ‘ISBN’ foreign keys to indicate the details of the resource. There is an additional field ‘DVD/Video?’ in the DVD_Video table to indicate whether the resource is a DVD or a Video. If a record from the ‘ISBN’, ‘CD_Name’ or ‘Film’ table is deleted, the row is deleted. Furthermore, if the ‘ISBN’ is updated, this update will cascade. As the other foreign keys are auto-incrementing primary keys of another tables, no ON UPDATE CASCADE is necessary.

Film Studio List Film, Author ISBN and Artist CD Name Tables

```
CREATE TABLE Film_Studios_List_Film (
    Film_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Film_ID) REFERENCES Film(Film_ID) ON DELETE CASCADE,
    /*ROW DELETED IF FK DELETED */
    Film_Studio_ID TINYINT(3) UNSIGNED NOT NULL,
    FOREIGN KEY (Film_Studio_ID) REFERENCES
Film_Studios_List(Film_Studio_ID) ON DELETE CASCADE
    /*ROW DELETED IF FK DELETED */
);

CREATE TABLE Author_ISBN (
    ISBN VARCHAR(13) NOT NULL,
    FOREIGN KEY (ISBN) REFERENCES ISBN(ISBN) ON UPDATE CASCADE ON DELETE
CASCADE,
    Author_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Author_ID) REFERENCES Author(Author_ID) ON DELETE CASCADE
    /*ROW DELETED IF FK DELETED */
);

CREATE TABLE Artist_CD_Name (
    CD_Name_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (CD_Name_ID) REFERENCES CD_Name(CD_Name_ID) ON DELETE
CASCADE,
    /*ROW DELETED IF FK DELETED */
    Artist_ID SMALLINT(5) UNSIGNED NOT NULL,
    FOREIGN KEY (Artist_ID) REFERENCES Artist(Artist_ID) ON DELETE CASCADE
    /*ROW DELETED IF FK DELETED */
);
```

The above all key tables show the relationship between the book and its author(s), the CD and its artist(s) and the film and its film studio(s). This table enables the database to store these many to many relationships e.g. a given book to have multiple authors and an author-to-author multiple books. On deletion on either of the foreign keys, the record will be deleted. As the other foreign keys are auto-incrementing primary keys of another tables, no ON UPDATE CASCADE is necessary.

Paid Fines Record

```
CREATE TABLE Paid_Fines_Record
(LOAN_ID MEDIUMINT(10) NOT NULL,
Member_ID SMALLINT(5) NOT NULL,
Amount_Paid SMALLINT(6) NOT NULL,
Date_Paid date NOT NULL);
```

This final table is peripheral to the database, as it does not contain any foreign keys, and exists in order to keep track of the previously paid fines for audit purposes. The Loan_ID, Member_ID, the amount paid, and the date on which the fine was paid will be stored in this table. This table can be used by the staff to check whether a certain member has paid the fine for a particular loan. Furthermore, as a future database expansion, this table could be expanded to include columns storing the payment method and the library staff member who accepted the fine payment. (N.B. This table was created to store information from a trigger and therefore is created empty, just before the trigger definition).

Sample Data

Sample data was used in this report to test the database. This submission contains the data in the script file ‘Group_20_Database_Creator.sql’. The insert data statements in the MySQL code start from Line 246 in script.
The data includes a number of CDs, Films and Books. To test the functionality of our database, some of these resources have multiple Artists/Authors associated to them. Most, but not all, resources are associated to a ‘Subject’ in the ‘Class_Number’ table and thereby associated to the linked ‘Department’ for that ‘Class’. As some subjects, namely the subjects for the ‘Music’ department are not yet inserted to the ‘Class_Number’ table, many of the CD Resources will have a ‘Class_ID’ of null. Library staff could insert these ‘Subjects’ and associated them to the CDs as they wish. Furthermore, some CDs such as CDs by the ‘The Beatles’ also have ‘John Lennon’ as an associated artist as ‘John Lennon’ is also stored as a solo artist in the database. The ‘Loan’ sample data includes a number of historic loans (in the past and returned) as well as some currently outstanding, unreturned loans. Care was taken to ensure that data constraints were met, and the data was realistic (reference items were not loaned out erroneously). The data constraints were tested and verified.

Views in MySQL

The following views were created to ensure useful information (derived from stored database values) could be conveniently obtained by database users and used in triggers and simple queries that are not included in this report.

View Popular Items

```
CREATE VIEW Popular_Items AS

SELECT R.Resource_ID, R.Location_ID, COUNT(L.Loan_ID) LOANS,
CONCAT( COALESCE(C2.Album_Name,''), COALESCE(I.Book_Name,''),
COALESCE(F.Film_Name,'')) ITEM_NAME,
CONCAT( COALESCE(A2.Artist_Name,''), COALESCE(CONCAT(
A4.Author_First_Name," ", A4.Author_Last_Name),'') ,
COALESCE(F3.Film_Studio_Name,'')) 'BY',
R.Type
FROM RESOURCE R
        LEFT JOIN CD C ON C.RESOURCE_ID=R.RESOURCE_ID
    LEFT JOIN CD_Name C2 ON C2.CD_Name_Id=C.CD_Name_ID
    LEFT JOIN Artist_CD_Name A ON A.CD_Name_ID=C.CD_Name_ID
    LEFT JOIN Artist A2 ON A.Artist_ID=A2.Artist_ID
    LEFT JOIN Book B ON B.Resource_ID=R.Resource_ID
    LEFT JOIN ISBN I ON I.ISBN=B.ISBN
    LEFT JOIN Author_ISBN A3 ON A3.ISBN=B.ISBN
    LEFT JOIN Author A4 ON A4.Author_ID=A3.Author_ID
    LEFT JOIN DVD_Video D on D.Resource_ID=R.Resource_Id
    LEFT JOIN Film F on F.Film_ID=D.Film_ID
    LEFT JOIN Film_Studios_List_Film F2 ON F2.Film_ID=F.Film_ID
    LEFT JOIN Film_Studios_List F3 ON F3.Film_Studio_ID=F2.Film_Studio_ID
        LEFT JOIN LOAN L ON L.Resource_ID=R.Resource_ID
    GROUP BY R.Resource_ID, C.CD_Name_ID, B.ISBN, D.FIlm_ID,
A2.Artist_Name, A2.Artist_Name, A4.Author_First_Name, A4.Author_Last_Name,
F3.Film_Studio_Name
    ORDER BY LOANS DESC
;
```

Resource_ID	Location_ID	LOANS	ITEM_NAME	BY	Type
47	86	4	What's Going On	Marvin Gaye	CD
75	1	4	Revolver	John Lennon	CD
24	54	4	Forrest Gump	Paramount Pictures	DVD/Video
69	13	4	Astral Weeks	Van Morrison	CD
75	1	4	Revolver	The Beatles	CD
60	14	3	Music From Big Pink	Bob Dylan	CD
60	14	3	Music From Big Pink	The Band	CD
70	2	3	Highway 61 Revisited	Bob Dylan	CD
49	6	3	Kind of Blue	Miles Davis	CD
13	56	3		1984 George Orwell	Book
34	41	3	Born to Run	Bruce Springsteen	CD
29	78	3	Parasite	Barunson E&A	DVD/Video
52	5	2	Horses	Patti Smith	CD
4	15	2	Crime and Punishment	Fyodor Dostoevsky	Book
37	14	2	The Rise and Fall of Ziggy Stardust and the Spiders From Mars	David Bowie	CD
9	100	2	How to Win Friends and Influence People	Dale Carnegie	Book
11	32	2	The Fellowship of the RingÂ	J.R.R Tolkein	Book
58	35	2	Innervisions	Stevie Wonder	CD

The view ‘Popular_Items’ derives the most frequently loaned resources (all time), using information from the Loan table. This will be used by library managers to potentially expand stock upon greater demand or perhaps for marketing/advertising purposes to attract more members or even to provide inspiration to current members.

View Outstanding Loans with Due Date

```
create view `Outstanding Loans with due date`
as
select L.Loan_ID, L.Member_ID,
R.Resource_ID, R.Borrowable_Time,
DATEDIFF(CURDATE(),Date_Taken_Out) AS `Days_Out`,
cast(R.Borrowable_Time as signed) - DATEDIFF(CURDATE(),Date_Taken_Out) as
`Days Before Due Date`
from
Loan L
left join
Resource R on
L.Resource_ID = R.Resource_ID where L.Date_Returned IS  null;
```

Loan_ID	Member_ID	Resource_ID	Borrowable_Time	Days_Out	Days Before Due Date
69	19	47	14	12	2
70	3	34	14	12	2
71	8	25	14	12	2
72	2	69	14	12	2
73	19	75	14	12	2
74	6	49	14	12	2
75	18	58	14	9	5
76	18	24	14	9	5
77	17	39	2	9	-7
78	15	36	14	9	5
79	2	64	14	9	5
80	16	69	14	9	5
81	16	11	14	9	5
82	11	69	14	6	8
83	17	9	14	6	8
84	13	8	14	6	8
85	7	70	14	6	8
86	19	75	14	6	8
87	20	60	14	6	8
88	19	28	14	6	8
89	17	7	14	6	8

The view ‘Outstanding Loans with Due Date’ show the loan details all unreturned items and will be used by library staff to monitor when items should be returned to the library, upon request by a member or for resource management.

View Number of Copies of Resource Available

```
create view `Number of copies of resource available` as

select R.Resource_ID, IFNULL(`Number Out On Loan`,0) as `Out on Loan` ,
R.No_of_Total_Copies,
R.No_of_Total_Copies - IFNULL(`Number Out On Loan`,0) as `Number of Copies
Left`
from
Resource R left join
(select Resource_ID, Count(Resource_ID) as `Number Out On Loan`
from `outstanding loans with due date` group by Resource_ID) as t3
 on R.Resource_ID = t3.Resource_ID
;
```

Resource_ID	Out on Loan	No_of_Total_Copies	Number of Copies Left
1	0	4	4
2	0	2	2
3	0	6	6
4	0	8	8
5	0	10	10
6	0	7	7
7	1	5	4
8	1	1	0
9	1	8	7
10	0	12	12
11	1	6	5
12	0	3	3
13	0	10	10
14	0	5	5
15	0	8	8
16	0	3	3
17	0	4	4
18	0	2	2
19	0	2	2
20	0	10	10
21	0	16	16
22	0	3	3
23	0	2	2
24	1	7	6
25	1	2	1
26	0	6	6
27	0	8	8
28	1	3	2
29	0	7	7
30	0	3	3
31	0	5	5
32	0	4	4
33	0	10	10
34	1	10	9
35	0	5	5
36	1	10	9
37	0	2	2
38	0	36	36
39	1	10	9
40	0	10	10

The view ‘Number of copies of resource available’ will be used by library managers as an inventory management tool when handing out loans or counting copies on shelves.

View Max Loan Ability

```
create view `Max loan ability` as
select Member_ID,
case when Member_Type = 'Student' then 5
when Member_Type = 'Staff' then 10
end  as `Max loanable items`
from Member  ;
```

Member_ID	Max loanable items
1	10
2	5
3	5
4	5
5	10
6	5
7	5
8	5
9	5
10	10
11	5
12	5
13	10
14	5
15	5
16	5
17	10
18	5
19	10
20	5

The view ‘Max loan ability’ shows the borrowing capacity of each member, derived from their member type. This view is later used in triggers to ensure a member does not borrow more resources than allowed.

View Number of Loans by Member

```
create view  `number of loans by member` as
select Member.Member_ID,
Member.Member_Type, t5.`Number of Loans`,
case when Member_Type = 'Student' then 5
when Member_Type = 'Staff' then 10
end  as `Max loanable items`
from Member
join
(select Member_ID, Count(Loan_ID)
as `Number of Loans`
from `outstanding loans with due date`
group by Member_ID order by Member_ID) as t5 on Member.Member_ID =
t5.Member_ID ;
```

Member_ID	Member_Type	Number of Loans	Max loanable items
2	Student	2	5
3	Student	1	5
6	Student	1	5
7	Student	1	5
8	Student	1	5
11	Student	1	5
13	Staff	1	10
15	Student	1	5
16	Student	2	5
17	Staff	3	10
18	Student	2	5
19	Staff	4	10
20	Student	1	5

The view ‘Number of Loans by Member’ displays the current number of loans a member has. Furthermore, it shows the number of items the member can loan.

View Overdue Items with Contact Details

```
create view `Overdue Items with Contact Details` as
select
L.Loan_ID, L.Member_ID,
M.First_Name, M.Last_Name,
M.Email_Address, M.Phone_Number,
R.Resource_ID, L.Date_Taken_Out, R.Borrowable_Time,
DATEDIFF(CURDATE(),Date_Taken_Out) AS `Days_Out`,
DATE_ADD(Date_Taken_Out, INTERVAL R.Borrowable_Time Day) as `Date Due`,
cast(R.Borrowable_Time as signed)
- DATEDIFF(CURDATE(),Date_Taken_Out) as `Days Before Due Date`
from
(Loan L join Resource R on L.Resource_ID = R.Resource_ID)
join Member M on L.Member_ID = M.Member_ID

where
L.Date_Returned IS null and
cast(R.Borrowable_Time as signed)
- DATEDIFF(CURDATE(),Date_Taken_Out)<0
order by L.Member_ID asc, `Days Before Due Date` asc;
```

Loan_ID	Member_ID	First_Name	Last_Name	Email_Address	Phone_Number	Resource_ID	Date_Taken_Out	Borrowable_Time	Days_Out	Date Due	Days Before Due Date
77	17	Alfred	Gilbert	a.gilbert@qmul.ac.uk	+447625608325	39	2020-12-04	2	9	2020-12-06	-7

The view ‘Overdue Items with Contact Details’ will be used by library managers in order to chase up overdue items with members – whether this is by automated email or SMS.

Member Fines View

```
CREATE VIEW Member_Fines AS
SELECT A.MEMBER_ID AS `Member No.`, A.FIRST_NAME AS Name, A.LAST_NAME AS
Surname, A.EMAIL_ADDRESS AS Email,
A.PHONE_NUMBER AS `Phone Number`, SUM(A.FINE_AMOUNT) AS `Total Fine`
FROM
(
    SELECT l.Loan_ID, l.Date_Taken_Out, l.Date_Returned, l.Date_Paid,
M.MEMBER_ID, m.First_Name, m.Last_Name,
    m.Email_Address, m.Phone_Number, r.Borrowable_Time
            , CASE WHEN DATE_RETURNED IS NULL THEN
                    DATEDIFF(CURDATE(), DATE_ADD(Date_Taken_Out, INTERVAL
R.Borrowable_Time Day))
                    ELSE DATEDIFF(L.DATE_RETURNED, DATE_ADD(Date_Taken_Out,
INTERVAL R.Borrowable_Time Day))
                    END AS FINE_AMOUNT
    FROM loan l
    INNER JOIN resource r
    ON l.Resource_ID = r.Resource_ID
    INNER JOIN member m
    ON l.Member_ID = m.Member_ID
    WHERE l.Date_Paid is NULL AND ((l.Date_Returned is NULL AND
ADDDATE(l.Date_Taken_Out, INTERVAL r.borrowable_time DAY) < CURDATE())
    OR (ADDDATE(l.Date_Taken_Out, INTERVAL r.borrowable_time DAY) <
l.Date_Returned)) -- all overdue items
) A
GROUP BY A.MEMBER_ID
;
```

Member No.	Name	Surname	Email	Phone Number	Total Fine
4	Logan	Dev	logdev@qmul.ac.uk	+447700900877	1
5	Frank	Barker	f.barker@qmul.ac.uk	+447266986247	2
6	Dewey	Parsons	dewey.parsons@qmul.ac.uk	+447881734090	3
9	Tanya	Stewart	tanya.stewart@qmul.ac.uk	+447666492455	5
10	Rafael	Castro	r.castro@qmul.ac.uk	+447092177480	1
11	Jackson	Russell	jr@qmul.ac.uk	+447779404025	3
12	Ed	Bossman	ed.bossman@qmul.ac.uk	+447135519680	2
13	Bryan	Goodwin	b.goodiwn@qmul.ac.uk	+447958097035	17
15	Leonardo	French	leo.french@qmul.ac.uk	+447424641475	19
17	Alfred	Gilbert	a.gilbert@qmul.ac.uk	+447625608325	7
18	Ryan	Ward	rward@qmul.ac.uk	+447164057510	9
19	Franklin	Thompson	f.thompson@qmul.ac.uk	+447068647165	3

The view ‘Member Fines’ will be used by library managers in order to find each member’s cumulative fine for all resources borrowed. This is represented by the derived column ‘Total Fine’. (N.B. It is assumed that when a resource is returned, any fine associated to it is paid at that moment). Member contact details: ‘Name’, ‘Surname’, ‘Email’ and ‘Phone Number’ are included as renamed columns for better readability.

Individual Fine Lookup

```
CREATE VIEW FINES AS
SELECT l.Loan_ID, M.MEMBER_ID
            , CASE WHEN DATE_RETURNED IS NULL THEN
                    DATEDIFF(CURDATE(), DATE_ADD(Date_Taken_Out, INTERVAL
R.Borrowable_Time Day)) -- IF NOT RETUREND
                    ELSE DATEDIFF(L.DATE_RETURNED, DATE_ADD(Date_Taken_Out,
INTERVAL R.Borrowable_Time Day)) -- IF RETURNED, ONLY OWE OVERDUE FINE UP
UNTIL RETURN
                    END AS FINE_AMOUNT
    FROM loan l
    INNER JOIN resource r
    ON l.Resource_ID = r.Resource_ID
    INNER JOIN member m
    ON l.Member_ID = m.Member_ID
    WHERE l.Date_Paid is NULL AND ((l.Date_Returned is NULL AND
ADDDATE(l.Date_Taken_Out, INTERVAL r.borrowable_time DAY) < CURDATE())
    OR (ADDDATE(l.Date_Taken_Out, INTERVAL r.borrowable_time DAY) <
l.Date_Returned)); -- all overdue items
```

Loan_ID	MEMBER_ID	FINE_AMOUNT
11	11	1
12	12	2
13	13	17
18	18	7
24	4	1
25	5	2
26	6	3
29	9	5
35	15	13
37	17	1
38	18	2
39	19	3
50	10	1
51	11	2
55	15	6
77	17	6

This view partially exists in order to assist the operation of a trigger that can make sure that an individual fine for a certain loan will be paid on the return of the item. It lists all the fines separately alongside the corresponding Loan_ID. This can be used by library staff to check if a particular item being returned has a fine associated with it, by a trigger to prevent the item from being returned without the fine being paid, and to provide data to be entered into the 'Paid_Fines_Record' table.

Triggers in MySQL

The following triggers were implemented using created views and tables in order to enforce functionality requirements of the library.

Loan Capacity Trigger

```
DELIMITER $$

create trigger check_loan_capacity_available before insert on loan for each
row
BEGIN

declare number_of_loans int;
declare max_loans int;
select ifnull(`Number of Loans`,0) into number_of_loans
from `number of loans by member`
where Member_ID = new.Member_ID;

select `Max loanable items` into max_loans from `max loan ability`
where  Member_ID = new.Member_ID  ;

if max_loans - number_of_loans = 0 then signal sqlstate '45000'
set message_text = 'Max loan amount reached';
END IF;

END $$
DELIMITER ;
```

This ‘before insert’ trigger ensures that each member cannot exceed their borrowing right capacity. If a member has already taken out the maximum number of resources as per their member type, the new loan will not be inserted, and an error will be triggered.

This trigger can be tested by repeatedly inserting a loan for Member_ID = 2.
Code could be like the below (with the final attribute, Resource_ID changing as required).

```
insert into Loan values (null,CURDATE(), null, null, 2, 6);
```

Running this repeatedly will eventually trigger the error.

```
EXECUTE FAIL:

 insert into Loan values (null,CURDATE(), null, null, 2, 6)

Message :

 Max loan amount reached
```

Resource Availability Trigger

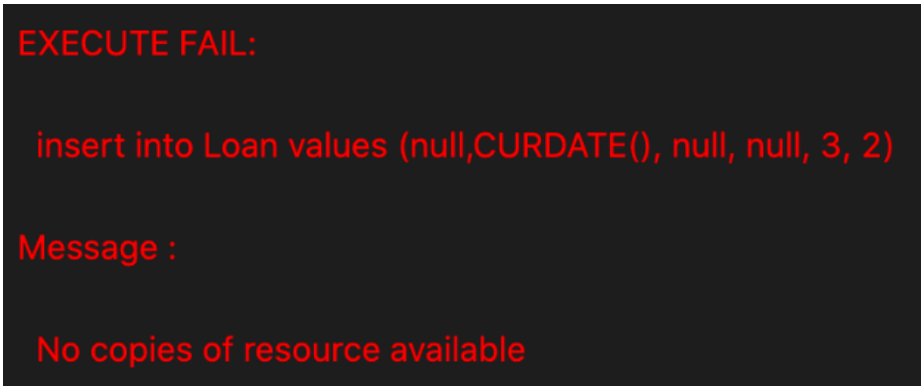
```
DELIMITER $$
create trigger check_resource_available before insert on loan for each row
BEGIN

declare copies_left int;
select `Number of Copies Left` into copies_left
from `Number of copies of resource available`
where Resource_ID = new.Resource_ID;

if copies_left = 0 then signal sqlstate '45000'
set message_text = 'No copies of resource available';
END IF;

END $$
DELIMITER ;
```

This ‘before insert’ trigger verifies that there are still remaining copies to loan out before confirming the loan. By using the view ‘Number of copies of resource available’, when this number is 0 – the message is printed.

This trigger can be tested by running the code:

```
insert into Loan values (null,CURDATE(), null, null, 1, 2);
insert into Loan values (null,CURDATE(), null, null, 2, 2);
insert into Loan values (null,CURDATE(), null, null, 3, 2);
```

The last value is the Resource_ID = 2 and there are only two copies. Trying to loan it three times will trigger an error.



Member Suspension Status Check

```
DELIMITER $$

CREATE TRIGGER before_loan_insert_member_suspended
BEFORE INSERT ON Loan
FOR EACH ROW
BEGIN
    DECLARE totalfine int;
    SELECT `Total Fine` INTO totalfine
     FROM member_fines
    WHERE `Member No.` = new.Member_ID;

    IF totalfine >= 10 THEN
        signal sqlstate '45000'
set message_text = 'Member suspended';
    END IF;
END $$
DELIMITER ;
```

This ‘before insert’ trigger stops any loan being issue to a currently suspended member (a member who owes £10 or more) by filtering suspended members from the ‘Member_Fines’ view.

This trigger can be tested by running a command like:

```
insert into Loan values (null,CURDATE(), null, null, 13, 1);
```

Member_ID=13 refers to a suspended member and the trigger will be activated and prevent the insert.

```
EXECUTE FAIL:

 insert into Loan values (null,CURDATE(), null, null, 13, 1);

Message :

 Member suspended
```

Control of Member not borrowing duplicate Resources at same time

```
DELIMITER $$
create trigger check_resource_not_already_loaned_by_member
before insert on loan for each row

BEGIN
declare number_of_loans_of_resource int;
select count(Loan_ID) into number_of_loans_of_resource
 from `outstanding loans with due date`
 where Member_ID=new.Member_ID and Resource_ID = new.Resource_ID;

if number_of_loans_of_resource > 0 then signal sqlstate '45000'
set message_text = 'Item already loaned out by member';
END IF;

END $$
DELIMITER ;
```

This ‘before insert’ trigger prevents any member from taking out two of the same resource at once – as per library rules.

This trigger can be tested by running a command like:

```
insert into Loan values (null,CURDATE(), null, null, 1, 7);
```

The important fields for the trigger are the Member_ID = 1 and Resource_ID = 7. Running the same command twice (or once if a copy of the resource is already loaned out) will activate the trigger and prevent the insert.

```
EXECUTE FAIL:

 insert into Loan values (null,CURDATE(), null, null, 1, 7)

Message :

 Item already loaned out by member
```

Checking Resource is not a reference item

```
DELIMITER $$

create trigger check_resource_not_reference
before insert on loan for each row
BEGIN
declare resource_borrowable_time int;

select Borrowable_time into resource_borrowable_time
from Resource
where Resource_ID = new.Resource_ID;
 if resource_borrowable_time = 0 then signal sqlstate '45000'
set message_text = 'Item is reference item';
END IF;

END $$
DELIMITER ;
```

This ‘before insert’ trigger stops any borrowing of a reference item in the library before the loan is added to the loan table.

This trigger can be tested by running a command like:

```
insert into Loan values (null,CURDATE(), null, null, 1, 16);
```

Resource_ID = 16 is a reference item and the trigger will be activated.

```
EXECUTE FAIL:

 insert into Loan values (null,CURDATE(), null, null, 1, 16);

Message :

 Item is reference item
```

Checking Fines are being paid upon return and storing paid fines data

```
delimiter //
Create Trigger fine_paid
Before Update On Loan
For Each Row
Begin
    Declare FINE int;
    SELECT FINE_AMOUNT into FINE
    FROM FINES
    where Loan_ID=new.Loan_ID;

    IF ((FINE IS NOT NULL) AND (NEW.DATE_PAID IS NOT NULL))
    THEN
        BEGIN
        INSERT INTO Paid_Fines_Record
        VALUES(New.Loan_ID,New.Member_ID,FINE,NEW.DATE_PAID);
        END;

        ELSEIF ((FINE>0) AND (NEW.DATE_PAID IS NULL))
        THEN
        signal sqlstate '45000' set message_text = 'Fine needs to be paid';

    END IF;
END//
```

This trigger acts before updates on the Loan table i.e., when an item is returned. It will do nothing if there is no fine associated with a particular loan, but if the loan has a fine that isn't paid when the item is returned it will prevent the data from being updated.

There are two possible scenarios when this trigger would be used. One is when a loan for an already returned item still has an unpaid fine associated to it. (Note, this scenario will be prevented for current and future loans by this trigger). When a fine is paid, the library staff will update the "date_paid" attribute with the current date with the following command:

```
UPDATE LOAN
SET DATE_PAID=CURDATE() WHERE LOAN_ID=13;
```

This will insert the following row into the "Paid_Fines_Record" table showing when the fine for a given loan is paid. In future implementation of the database, the library_staff_id can also be inserted into this table for audit purposes.

LOAN_ID	Member_ID	Amount_Paid	Date_Paid
13	13	17	2020-12-12

The other scenario is when a member returns an overdue item. In the sample data the loan with a LOAN_ID of 77 is overdue and not yet returned. The library staff may attempt to update the return date with the following command:

```
UPDATE LOAN
SET DATE_RETURNED=CURDATE() WHERE LOAN_ID=77;
```

The trigger will not allow the item to be returned unless the fine is paid. The following error will be produced:

```
EXECUTE FAIL:

 UPDATE LOAN SET DATE_RETURNED=CURDATE() WHERE
LOAN_ID=77;

Message :

 Fine needs to be paid
```

The correct command, assuming the fine is paid (as necessary), is:

```
UPDATE LOAN
SET DATE_RETURNED=CURDATE(), DATE_PAID=CURDATE() WHERE LOAN_ID=77;
```

A row is added to the “Paid_Fines_Record” recording the details of this action.

LOAN_ID	Member_ID	Amount_Paid	Date_Paid
13	13	17	2020-12-12
77	17	6	2020-12-12

Queries

The following queries demonstrate the database's ability to display useful information from the stored tables and views. (N.B. a lot of the more useful queries are stored as views and such are not repeated here).

Total Copies of each CD in the Library

```
SELECT C2.Album_Name, A.Artist_Name, R.No_of_Total_Copies
FROM Resource R
INNER JOIN CD C ON R.Resource_ID = C.Resource_ID
INNER JOIN CD_Name C2 ON C.CD_Name_ID=C2.CD_Name_ID
INNER JOIN Artist_CD_Name A1 ON A1.CD_Name_ID=C2.CD_Name_ID
INNER JOIN Artist A ON A.Artist_ID=A1.Artist_ID
ORDER BY R.No_of_Total_Copies DESC;
```

Album_Name	Artist_Name	No_of_Total_Copies
Hotel California	Eagles	36
Live at the Apollo, 1962	James Brown	36
John Lennon/Plastic Ono Band	Plastic Ono Band	21
Are You Experienced	The Jimi Hendrix Experience	21
John Lennon/Plastic Ono Band	John Lennon	21
Abbey Road	The Beatles	12
The Beatles ("The White Album")	The Beatles	12
Blood on the Tracks	Bob Dylan	12
Astral Weeks	Van Morrison	12
Forever Changes	Love	12
Abbey Road	John Lennon	12
The Beatles ("The White Album")	John Lennon	12
Revolver	John Lennon	10
Rubber Soul	John Lennon	10
Born to Run	Bruce Springsteen	10
A Love Supreme	John Coltrane	10
Blue	Joni Mitchell	10
Rubber Soul	The Beatles	10
Rumours	Fleetwood Mac	10
What's Going On	Marvin Gaye	10
Revolver	The Beatles	10
Kind of Blue	Miles Davis	10
The Great Twenty_Eight	Chuck Berry	10
Horses	Patti Smith	10
Music From Big Pink	Bob Dylan	10
Legend: The Best of Bob Marley and The Wailers	Bob Marley & The Wailers	10
Ramones	Ramones	10
Exile on Main St.	The Rolling Stones	10
Innervisions	Stevie Wonder	10

This query aims to check how much the library has capacity for each CD, by joining all CD's and its respective attributes into one table – all CD details are listed as well as their total copy number in the library. Not all rows are shown, and an album is repeated if it has more than one artist associated to it. This can be used by library staff for bookkeeping purposes.

Number of Books from each Country

```
SELECT Country_Name, COUNT(I2.ISBN) No_of_Books_From_Country FROM Country C
LEFT JOIN Author a ON a.Country_ID=C.Country_ID
LEFT JOIN Author_ISBN I ON I.Author_ID=a.Author_ID
LEFT JOIN ISBN I2  ON  I2.ISBN = I.ISBN
GROUP BY Country_Name
ORDER BY No_of_Books_From_Country DESC;
```

Country_Name	No_of_Books_From_Country
United Kingdom	5
United States	4
Belgium	3
Germany	2
Brazil	1
Czech Republic	1
Greece	1
Ireland	1
Israel	1
Japan	1
Russian Federation	1
Switzerland	1
Tajikistan	1
Afghanistan	0
Åland Islands	0
Albania	0
Algeria	0
American Samoa	0
Andorra	0
Angola	0
Anguilla	0
Antarctica	0

This query uses joins Books and ISBN to country table and counts all books according to country. This query includes countries that do not have books. Not all rows are shown. The user can choose the naitonalities of the author per country when choosing a book.

Books Written by Russian Authors

```
SELECT I2.Book_Name, a.Author_First_Name, a.Author_Last_Name, C.Country_Nam
e FROM Author a
INNER JOIN Author_ISBN I ON I.Author_ID=a.Author_ID
INNER JOIN ISBN I2 ON I2.ISBN=I.ISBN
INNER JOIN Country C ON C.Country_ID=a.Country_ID
WHERE C.Country_Name='Russian Federation';
```

Book_Name	Author_First_Name	Author_Last_Name	Country_Name
Crime and Punishment	Fyodor	Dostoevsky	Russian Federation

This query aims to display all available book in the library written by Russian authors. Filtering resources by country upon searching is a library requirement and users can use this query with a slight alteration to find books written by an author of a certain nationality, due to their personal interest or the relevance for a certain class.

CDs for Jazz Class in the Music Department

```
Select CD_Name.Album_Name, Artist.Artist_Name, Class_Number.Subject
FROM CD_Name,CD,Artist_CD_Name, Artist, Resource, Class_Number
WHERE CD.CD_Name_ID=CD_Name.CD_Name_ID
AND CD_Name.CD_Name_ID = Artist_CD_Name.CD_Name_ID
AND Artist.Artist_ID = Artist_CD_Name.Artist_ID
AND CD.Resource_ID=Resource.Resource_ID
AND Class_Number.Class_ID=Resource.Class_ID
AND Class_Number.Subject='Jazz' ;
```

Album_Name	Artist_Name	Subject
A Love Supreme	John Coltrane	Jazz

This query shows that the database can be used by student of a class to find their class resources with ease.

Searching for Specific Films

```
SELECT Loan.Resource_ID, Film.Film_Name, Film.Date_Released, Member.First_N
ame
as 'Loaned by', Member.Member_ID, Loan.Date_Taken_Out
from Film
inner join Loan on Film_ID=Loan_ID
inner join Member on Loan.Member_ID = Member.Member_ID
where Film_Name like '%the%'
order by Loan_ID ASC;
```

Resource_ID	Film_Name	Date_Released	Loaned by	Member_ID	Date_Taken_Out
17	The Shawshank Redemption	1995-02-17	John	1	2020-11-01
24	The Dark Knight	2008-07-24	David	2	2020-11-01
30	The Matrix	1999-06-11	Frank	5	2020-11-01
56	The Green Mile	2000-03-03	Dewey	6	2020-11-01

This query displays all members who took out films whose title starts with ‘The’. Different forms of this query can of course be used to find a specific film or a film with a certain key word in the title.

Current Loan Duration Details

```
select L.Loan_ID, L.Member_ID,
R.Resource_ID, R.Borrowable_Time,
DATEDIFF(CURDATE(),Date_Taken_Out) AS `Days_Out`,
cast(R.Borrowable_Time as signed) - DATEDIFF(CURDATE(),Date_Taken_Out) as `
Days Before Due Date`
from Loan L
left join
Resource R on L.Resource_ID = R.Resource_ID
where L.Date_Returned IS  null;
```

Loan_ID	Member_ID	Resource_ID	Borrowable_Time	Days_Out	Days Before Due Date
69	19	47	14	11	3
70	3	34	14	11	3
71	8	25	14	11	3
72	2	69	14	11	3
73	19	75	14	11	3
74	6	49	14	11	3
75	18	58	14	8	6

This query aims to search for the loan details, specifically those which pertain to borrowing time. The library can know when to potentially expect a resource back into the library - thereby generating further statistics.

Number of Resources of each type

```
select type as `Resource_Type`, count(resource_id) as `Number` from
Resource group by Type.
```

Resource_Type	Number
Book	21
DVD/Video	11
CD	47

This query enables the user to split total resource numbers into total per type of resource for ad-hoc queries.

List of Overdue Loans for Given Member

```
select * from `overdue items with contact details` where Member_ID = 17;
```

Loan_ID	Member_ID	First_Name	Last_Name	Email_Addre	Phone_Numl	Resource_ID	Date_Taken_	Borrowable_Time	Days_Out	Date Due	Days Before Due Date
77	17	Alfred	Gilbert	a.gilbert@qr	+447625608	39	2020-12-04	2	8	2020-12-06	-6

This query uses a view to find overdue loan details of a specific member.

Checks Class Number Association of Films

```
SELECT F.Film_Name, F.Genre from Film F, DVD_Video D, Resource R
WHERE F.Film_ID=D.Film_ID
AND R.Resource_ID=D.Resource_ID
AND Genre='Horror'
```

Film_Name	Genre
Alien	Horror

The query above results in the output of the movie within that genre.

Cybersecurity Considerations

Cybersecurity can be defined as “protecting computers and their infrastructure, including networks, from damage to hardware, software, data disruption, or misdirection.”
The importance of this cannot be understated, especially due to an increased reliance and usage of information technology since the start of the COVID-19 pandemic.
Concerning vulnerabilities and attacks - there are many categories, the most relevant ones for this coursework are:

- Backdoor
- Denial-of-service attack
- Direct-access attacks
- Privilege escalation
- Spoofing
- Social engineering

A vast majority of supply chain risks lie in many an inferior product design, development and production, distribution, acquisition and deployment, maintenance, and disposal phases. Possible backdoors for hackers may be overlooked during the design phase and or even purposefully implemented as an industrial espionage act during the design, development, production, distribution, acquisition, and deployment maintenance phases. Simultaneously, inadequate care in a disposal phase may lead to unauthorized access to critical data by third parties (Cybersecurity & Infrastructure Security Agency, 2020).
The above discussed threats mainly apply to the protection of a server host in the case of the library system rather than the MySQL server discussed in a later subsection of this report.

Data Protection Act 2008

This is UK’s implementation of the European GDPR responsible for a fair, lawful and transparent processing of data, stating that all data must be accurate, kept up to date and deleted if no longer required (GOV.UK, 2018).

From a library database system provider standpoint, it is important to identify the data that is processed by the library. Clearly, the ‘Member’ table contains very sensitive personal data, and a library member may not want their loan history to be public. It is vital to define how sensitive data will be handled by the library. After this step appropriate measures on handling must be implemented, protecting the data accordingly with database views, cybersecurity measures and deleting user data once no longer needed. To comply with the Data Protection, Act the user of the library database system must consent to the storage and future use of their data, when they first provide their data, meaning, in this library scenario, all new registered users are required to check a tick box when registering to acknowledge their understanding and give consent for data usage and storage.

The last step is demonstrating compliance, by documenting the processes and explaining the measures regarding privacy and processes such audits for users who which to access the, erase, or move the data. A designation of a data protection officer is required, as the library database in this scenario is a public authority.

Specific Addressable Issues for a Library database system

Firstly, it is clear that members' personal data should not be available to the public nor library staff who would have no use for it. As such, permission will be set so only the library manager is able to see all members' contact details, as the only reason this data would be required would be to contact the member regarding fines, suspension or overdue item. Of course, the library could also send out a weekly newsletter, but in that case, members would have to give permission for their email to be added to the mailing list, and only the member of staff responsible for the mailing list (for example the manager or data protection officer) would know the emails on the mailing list. This prevents normal "frontline" staff, responsible for issuing loans etc. from knowing the contact details of the library member they are interacting with.
Furthermore, while loans do have to be tied to a Member_ID, the contact details will not be stored in that table to allow some anonymization of this data. In fact, the only view that links a resource to a member and their contact details is the view showing overdue items, where contact details are necessary for reminder emails etc.

Finally, of course, it is very important for library staff, including database managers, to behave ethically. No library member should be contacted unless necessary and the member has given permission, and the topic must be the relevant agreed upon topic (overdue items or suspensions). Library staff must not use their personal email/phone to contact members, nor pass on a member's contact details to another individual under any circumstances.

User Permissions

To prevent data being seen by the wrong eyes, the right user permissions are necessary. Users, such as library staff, will only be given permissions to access views and tables relevant to their role at the library. Furthermore, they will not be given permission to edit database data, unless their role requires it.

An example, using an SQL query to set user permissions, utilizing most commonly available permissions options is shown below:

```
grant SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD, SHUTDOWN,
PROCESS, FILE, REFERENCES, INDEX, ALTER, SHOW DATABASES, SUPER, CREATE
TEMPORARY TABLES, LOCK TABLES, EXECUTE, REPLICATION SLAVE, REPLICATION
CLIENT, CREATE VIEW, SHOW VIEW, CREATE ROUTINE, ALTER ROUTINE, CREATE USER,
EVENT, TRIGGER, CREATE TABLESPACE, CREATE ROLE, DROP ROLE on *.* to
DESIRED_USERNAME with grant option;
```

An unadvisable command granting all privileges to a user, which should solely be used for database administrator users, is shown below:

```
grant ALL PRIVILEGES on DATABASE_NAME.* to DB_ADMIN;
```

SQL injection

A highly relevant threat worth mentioning within the scope of this course (Database Systems EC740P) is SQL injection, which can be used to attack data-driven applications. Data-driven applications already dominate the market, as data-driven businesses grow 30% on average every year (Forrester, 2018), which increases this threat significance annually and its potential consequences resulting from damages.
SQL injection exploits the application's security vulnerability and is mostly used as an attack vector for websites by placing malicious code into SQL statements.
Most commonly used SQL injection techniques:

- Boolean-based blind
- Time-based blind
- Error-based
- UNION query-based
- Stacked queries
- Out-of-band

SQL Injection (JavaScript Code Injection Example)

A commonly known vulnerability uses user inputs in SQL statements, opening a back door for SQL injection hackers. One vulnerability may be using user input in an SQL database. This specific attack method would be relevant for the library database created in this coursework, if hypothetically linked to a web-based application that may use JavaScript in its front end, resulting in vulnerabilities in the backend being exploited. For instance, if there are no constraints in a SQL statement preventing the user from entering the "wrong" input, the user could enter the "smart" input.
Regarding the library database, as the given case for coursework 2, the database could be linked to a web-based application asking for a username and password as follows:

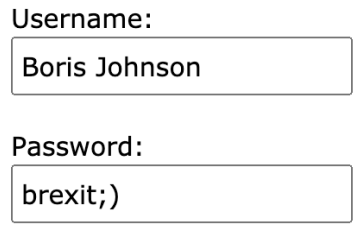


Image 1: Username & Password Example

With the username and password box above the following code could be triggered in the background when submitting the request (JSON & SQL):

```
username = getRequestString("username");
pass = getRequestString("userpassword");

sql = 'SELECT * FROM Member WHERE Username ="' + username + '" AND Password
="' + pass + '"'
```

The above code will trigger the following SQL query:

```
SELECT * FROM Member WHERE Username ="Boris Johnso" AND Password
="brexit;)"
```

Now when exploiting the vulnerability and entering a smart input of " or ""=" for the username and " or ""=" for the password field, the following SQL query will be triggered:

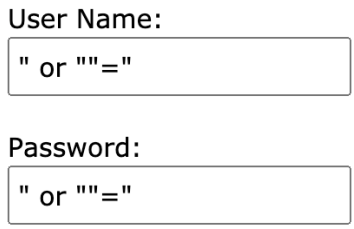


Image 2: Username & Password Example for Vulnerability Exploitation

And result as shown above from a JSON executable (if web bases) to the following SQL Query:

```
SELECT * FROM Member WHERE Username ="" or ""="" AND Password ="" or ""=""
```

Which as an SQL statement knowingly returns all rows from the username and password table, since *OR ""=""* is always *True*.

This will result in a severe data leak. It is now a matter of the hacker's creativity to decide what to do with the freshly exploited password and username list. A 2019 survey by Google showed that over 50% of all surveyed participants reuse the same passwords for multiple accounts and 13% for all accounts (Google, 2019). This gives the hacker an estimated success that half of the mined passwords could grant the hacker access to further services/ applications in the victims' use (DELOITE, n.d).
As an additional example, this could lead to property damage and is commonly used to practice industrial espionage used by Chinese government hackers (The United States Department of Justice, 2020).
As a result liability issues for companies arise with governmental data protection acts, such as GDPR and the UK Data Protection Act 2018, enforcing standards that minimize cybersecurity risks and handle data ownership (“General Data Protection Regulation (GDPR) Compliance Guidelines,” 2020).
To prevent this vulnerability, a specific user entry constraint could be created in the front end, as well as the back end.

Additional features that MySQL is shipped with are password strength validators, which protect against most brute force attacks and can be used with the following line of code:

```
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('N0Tweak$_@123!');
```

User Privileges
Firstly, the database should be password protected and not accessible without a password request by the following statement:

```
mysql -u root
```

Before the database is launched, privileges of all existing database users should be checked. User rights, that were granted for development purposes, can be updated and removed with the ‘REVOKE’ command.

Concerning users of the database, such as staff, students and library employees, the front end could create each usernames and passwords, which in the backend creates normal MySQL users, but will only have a ‘SELECT’, ‘UPDATE’ and ‘DELETE’ privilege solely on tables the users require. With PHP the entered password could get encrypted already by the front end and store the hashed key on the MySQL database, which can be hashed again .
For example, in php a hash would use the following command:

```
password_hash ( string $password , mixed $algo [, array $options ] )
: string|false
```

This way the users would create their very own passwords and usernames, which will remain private without any intermediaries, and only be eligible to create an account if they have a QMUL email address in our database scenario as discussed in an earlier section concerning the Members table in this report.

When storing the hash keys on the database it could also get encrypted locally once more though the following SQL function, via a trigger so every new entry is encrypted:

```
CREATE TRIGGER `Hash_New_Entry`
AFTER INSERT on Member
FOR EACH ROW
update Member set new.Email_Address=to_base64(aes_encrypt(Email_Address,
'keys')),new.key_1=to_base64(aes_encrypt(key_1,'keys'))
```

In the example above the hash key could get stored
If the database ever gets viewed by a hacker the usernames and passwords will be useless, alternatively the email addresses, as well as address details could get stored as hashed values on the database as well, which could decrease the chance of a point of contact for social network engineering or phishing if data is leaked, and also makes data storage more secure.

Scalability of the Database and Future Expansion

During a pandemic, with team members that have never met physically and residing in two different countries, collaboration on the same database has been successful using a cloud infrastructure, namely AWS. To minimize downtime of the library system and given that the student body will be international, it is advisable to make use of common cloud services such a Google Cloud or AWS. This has the advantage of low latency, useful when library resources are planned to go fully digital in the future. The database will be scalable though a cloud service, and therefore, administrative services of QMUL physical servers could be used for other applications. As the student body of QMUL is international and it is likely the service will be provided to students remotely and globally, it is advisable to use a common cloud service. Google Cloud, for instance, has a compliance offering that cover most global data standards (Americas, EMEA, Asia Pacific), such as ISO/IEC 27001 and many more offering a globally compliant solution.
Furthermore, the database can be scaled and hooked up to a graphical user interface by adding additional “username” and “password” columns into the Member table.
For an initial new user, a regex (regular expression) function could be used to easily generate usernames for the user’s first usage. The new username column could use the first three digits of a user’s last name and the first three of a user’s first name and all lower-case letters. This way the usernames are compact and easy to remember by the user. An example of this regex function could be as follows:

```
lower("#{field('Last_Name')[0,3]}#{field('First_Name')[0,3]}")
```

Concerning the passwords, it is highly recommended to solely store the respective hash keys for user privacy and cybersecurity considerations.

In addition, further tables and attributes can be added. A future consideration may be adding a language of inventory items for the user to know what language the medium is in, as not all media is in the local country of origin language. For instance, the author Franz Kafka is born in Prague with the country Czech Republic in our database, but his poems are all in German, therefore it could be useful to add language information.
A table storing library staff details would also be useful and this could be then used to track who processes a given fine payment and even who returns a resource to a particular shelf and when. This is useful for audit purposes.

Concerning the user-friendliness of the library system, the database can be used to send reminder emails when an item is overdue etc.

Conclusion

MySQL was used to develop a university library database, as per the normalised design shown in Coursework 1. A number of views and triggers were generated to fulfil functional requirements, while ad-hoc queries are provided to show the power of MySQL language in the context of this database.

Bibliography

5 insights on cyberattacks and intellectual property, n.d.

Cybersecurity & Infrastructure Security Agency, 2020. ICT SUPPLY CHAIN RISK MANAGEMENT [WWW Document]. URL https://www.cisa.gov/supply-chain (accessed 6.28.20).

Forrester, 2018. Insights-Driven Businesses Set The Pace For Global Growth [WWW Document]. URL https://www.forrester.com/report/InsightsDriven+Businesses+Set+The+Pace+For+Global+Growth/-/E-RES130848 (accessed 12.5.20).

General Data Protection Regulation (GDPR) Compliance Guidelines [WWW Document], 2020. . GDPR.eu. URL https://gdpr.eu/ (accessed 12.5.20).

GOV.UK, 2018. Data protection [WWW Document]. GOV.UK. URL https://www.gov.uk/data-protection (accessed 12.8.20).

Online Security Survey Google / Harris Poll, n.d.

says, K., n.d. sqlmap usage guide. Part 1: Basic web-site checks (GET) - Ethical hacking and penetration testing. URL https://miloserdov.org/?p=1320 (accessed 12.5.20).

SQL Injection [WWW Document], 2020. URL https://www.w3schools.com/sql/sql_injection.asp (accessed 12.5.20).

Stockman, T., 2020. Coursework 1 Specification.

The United States Department of Justice, 2020. Chinese Military Personnel Charged with Computer Fraud, Economic Espionage and Wire Fraud for Hacking into Credit Reporting Agency Equifax [WWW Document]. URL https://www.justice.gov/opa/pr/chinese-military-personnel-charged-computer-fraud-economic-espionage-and-wire-fraud-hacking (accessed 12.13.20).