**Coventry University**

**4005CEM Module Coursework (REPORT)**

**Ernest Zimus**

Contents

[Introduction 3](#_Toc68532181)

[1 A Task 3](#_Toc68532182)

[1 B Task 4](#_Toc68532183)

[Normalization 5](#_Toc68532184)

[The Entity Relationship Diagram 8](#_Toc68532185)

[Full implementation of the database into SQL 9](#_Toc68532186)

[1 C Task 10](#_Toc68532187)

[Part 1. – Find the customers that have rented films priced £2.6 – output theirID and name. 10](#_Toc68532188)

[Part 2. Find the customers that spent less than £3 on individual film rentals, but have spent a total higher than £15 - output their ID and name. 10](#_Toc68532189)

[Part 3. - Find the customers that have rented the film “Gamer” – output their ID and name. 11](#_Toc68532190)

[Part 4. - Find all films that are currently rented – output their name. 11](#_Toc68532191)

[Part 5. - Find the film that has the highest number of rentals – output their name 11](#_Toc68532192)

[Conclusion 11](#_Toc68532193)

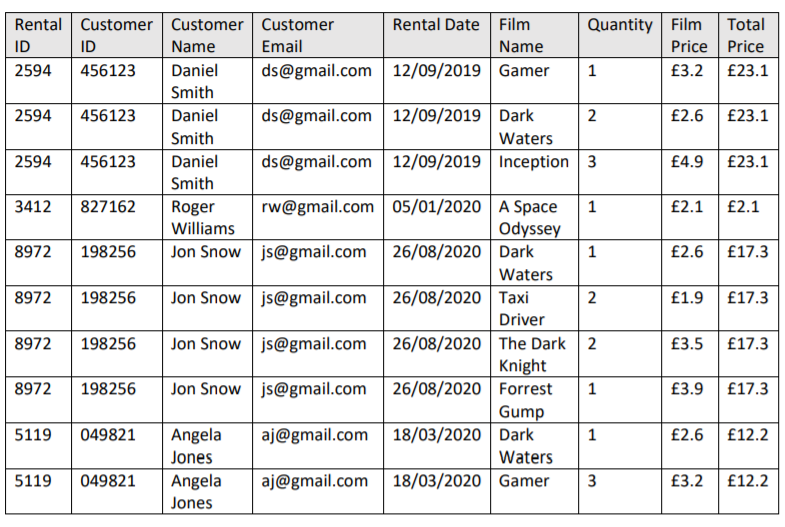
[References 12](#_Toc68532194)

# Introduction

As for the introductory part, the databases are immensely helpful to websites, various businesses etc., they play a huge role as they hold data that could be such as customer names, customer addresses, previous customer orders. Databases are also used for automation purposes for instance: shopping websites contain databases with item available in inventory, their price, title, description and so on. If that kind of database would be deleted, the website just would not work. A lot of businesses rely on them.

# 1 A Task

I have a table that is provided by the coursework brief that looks like this:



As this table is a traditional database and this is for a movie rental system, judging by how this table looks like, the first thing that the data in this table has duplicate entries which lead to a lot of manual work. By what I mean manual work, for instance: the movie rental shop that would use this system with that kind of database(the table), and Jon Snow approaches the customer servant in the shop and asks for the email change and his last name as he decided to legally change it and Jon has rented for example 43 movies in total since this business was operating, the customer servant would need to change every Customer Name and Customer Email attributes on every row that has Jon in it. Or a huge company that has rented over 1000 different movies has an email change too. This is time-consuming. This issue could be fixed by doing the database normalization process. When duplicate attributes could be stored in a separate database and then linked to the main database using primary keys. And no, it should not be kept that way, it needs normalization. Plus, this table cannot contain primary keys due to duplicate values. Primary keys are unique.

# 1 B Task

To implement this table into a SQL database I will be using SQLiteStudio (3.2.1) First thing is that I open SQLiteStudio and click on Database->Add a database, select the type as SQLite 3, click on the + icon and name it as “movierent” click on Save and then click on OK. To begin the implementing process, I will be using custom SQL queries from now on. Well as we have this table it should be normalized as we can not use it properly in SQL, by what I mean it cannot have a primary key, foreign key as we have duplicate values right now. For curiosity, the SQL code would be for this exact table:

**CREATE** **TABLE** Rentals(

Rental\_ID INT **NOT** **NULL**,

Customer\_ID INT **NOT** **NULL**,

Customer\_Name TEXT **NOT** **NULL**,

Customer\_Email TEXT,

Rental\_Date TEXT **NOT** **NULL**,

Film\_Name TEXT **NOT** **NULL**,

Quantity INT **NOT** **NULL**,

Film\_Price INT **NOT** **NULL**,

Total\_Price INT **NOT** **NULL**);

**INSERT** **INTO** Rentals **VALUES**('2594', '456123', 'Daniel Smith', 'ds@gmail.com', '12/09/2019', 'Gamer', '1', '3.2', '23.1'),

('2594', '456123', 'Daniel Smith', 'ds@gmail.com', '12/09/2019', 'Dark Waters', '2', '2.6', '23.1'),

('2594', '456123', 'Daniel Smith', 'ds@gmail.com', '12/09/2019', 'Inception', '3', '4.9', '23.1'),

('3412', '827162', 'Roger Williams', 'rw@gmail.com', '05/01/2020', 'A Space Odyssey', '1', '2.1', '2.1'),

('8972', '198256', 'Jon Snow', 'js@gmail.com', '26/08/2020', 'Dark Waters', '1', '2.6', '17.3'),

('8972', '198256', 'Jon Snow', 'js@gmail.com', '26/08/2020', 'Taxi Driver', '2', '1.9', '17.3'),

('8972', '198256', 'Jon Snow', 'js@gmail.com', '26/08/2020', 'The Dark Knight', '2', '3.5', '17.3'),

('8972', '198256', 'Jon Snow', 'js@gmail.com', '26/08/2020', 'Forest Gump', '1', '3.9', '17.3'),

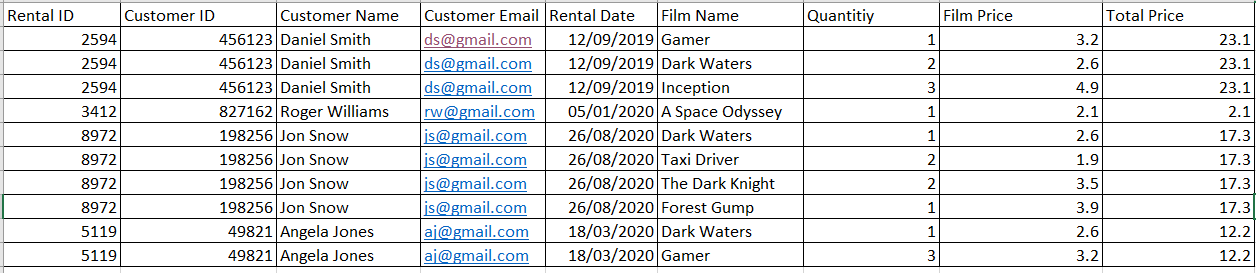
('5119', '049821', 'Angela Jones', 'aj@gmail.com', '18/03/2020', 'Dark Waters', '1', '2.6', '12.2'),

('5119', '049821', 'Angela Jones', 'aj@gmail.com', '18/03/2020', 'Gamer', '3', '3.2', '12.2');

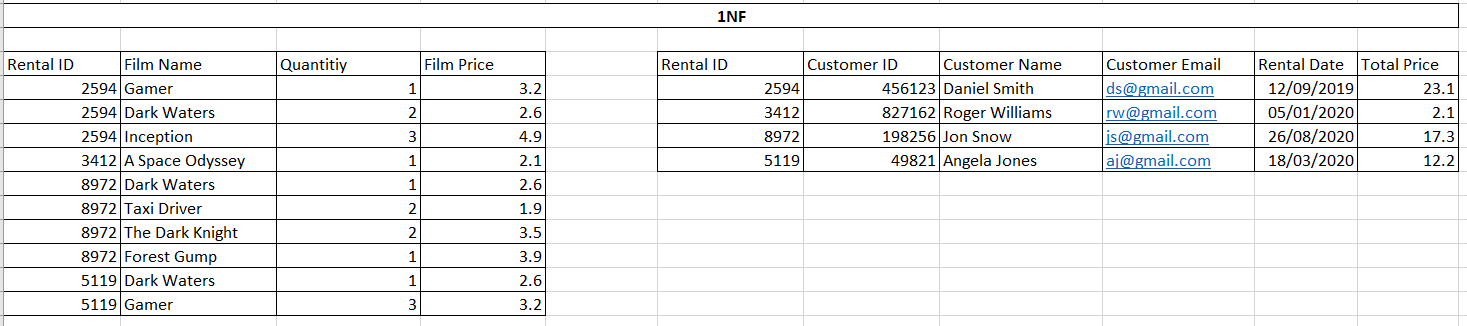
But this is not the final code as we must normalize the database.

## Normalization

So, for the normalization process I have done these prototypes in Microsoft Excel:

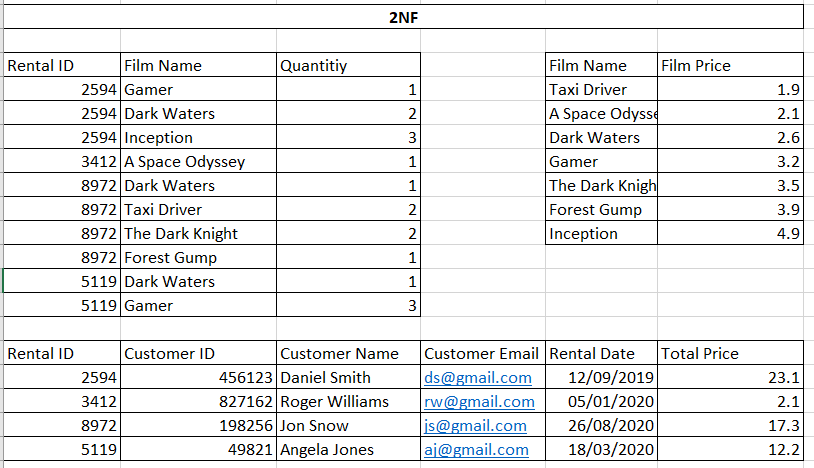
**Basic table** 

**1NF**

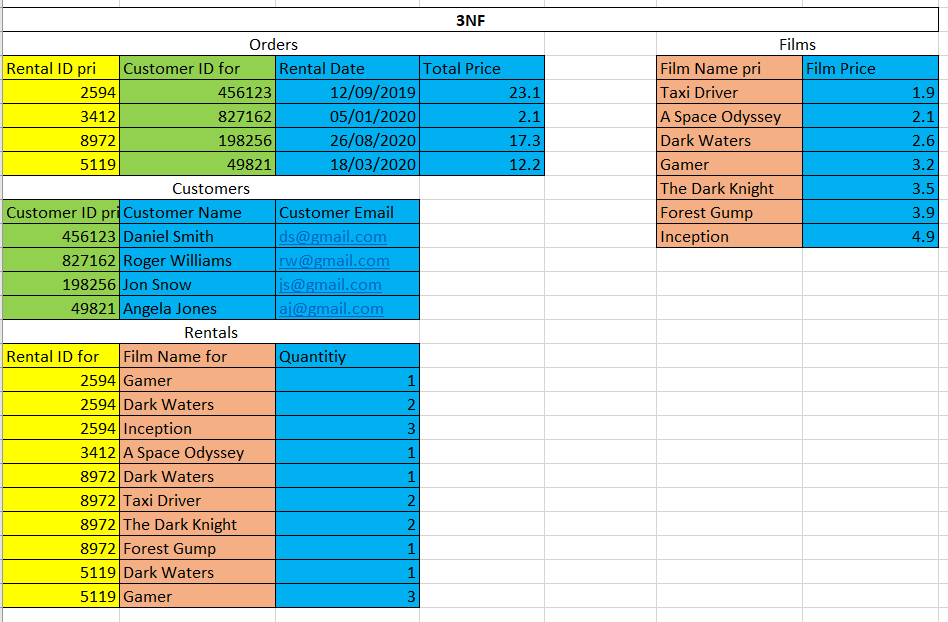


We can already see changes. The primary key(Rental ID) would be in the right table, foreign key(Rental ID) would be in the left table. The primary key is unique and cannot contain duplicates. But still, there could be more changes made.

**2NF**

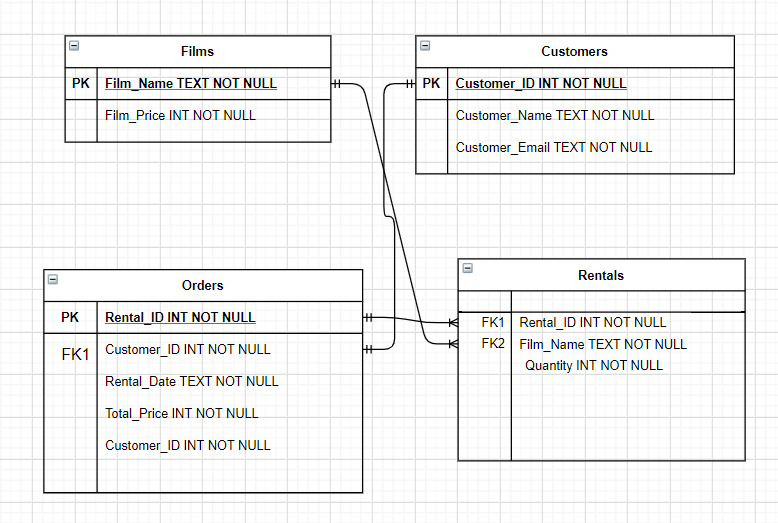


The bottom table remains unchanged, but the table in left that was in 2NF is now changed, the Film Price was moved out into another table in the right and Film Name was added as a primary key, the duplicates were removed from the right table.

**3NF** 

For 3NF in this picture, I have had it with colours (they just show the links between each other) and you can see that some entities have pri or for near them. Pri stands for primary, for stands for foreign. All this was used for to build it into an SQL database. Everything was moved out to different tables to contain the least duplicates as possible. Films, Orders, Customers tables do not contain duplicates as they have primary keys. The only table that has duplicates is Rentals. It relies on primary keys.

## The Entity Relationship Diagram



**USED** [**https://draw.io/**](https://draw.io/) **to draw this entity relationship diagram**

Entity Films have attributes: Primary key Film\_Name, Film\_Price

Entity Customers have attributes: Primary key Customer\_ID, Customer\_Name, Customer\_Email

Entity Orders have attributes: Primary key Rental\_ID, Foreign key Customer\_ID that is linked to the Primary key Customer\_ID in the table Customers using one mandatory to one relationship. Rental\_Date, Total\_Price, Customer\_ID

Entity Rentals have attributes: Foreign key Rental\_ID which is linked to Primary Key Rental\_ID in the Orders table using many mandatory to one mandatory relationship and Foreign key Film\_Name which is linked to a Primary key Film\_Name in Films table using many mandatory to one mandatory relationship.

## Full implementation of the database into SQL

For the 3NF table the code is:

**CREATE** **TABLE** Films(

Film\_Name TEXT **PRIMARY** **KEY** **NOT** **NULL**,

Film\_Price INT **NOT** **NULL**);

**CREATE** **TABLE** Customers(

Customer\_ID INT **PRIMARY** **KEY** **NOT** **NULL**,

Customer\_Name TEXT **NOT** **NULL**,

Customer\_Email TEXT **NOT** **NULL**);

**CREATE** **TABLE** Rentals(

Rental\_ID INT **NOT** **NULL**,

Film\_Name TEXT **NOT** **NULL**,

Quantity INT **NOT** **NULL,**

**FOREIGN** **KEY** (Rental\_ID) **REFERENCES** Orders(Rental\_ID),

**FOREIGN** **KEY** (Film\_Name) **REFERENCES** Films(Film\_Name));

**CREATE** **TABLE** Orders(

Rental\_ID INT **PRIMARY** **KEY** **NOT** **NULL**,

Customer\_ID INT **NOT** **NULL**,

Rental\_Date TEXT **NOT** **NULL**,

Total\_Price INT **NOT** **NULL,**

**FOREIGN** **KEY** (Customer\_ID) **REFERENCES** Customers(Customer\_ID));

**INSERT** **INTO** Films **VALUES**('Taxi Driver','1.9'),

('A Space Odyssey','2.1'),

('Dark Waters','2.6'),

('Gamer','3.2'),

('The Dark Knight','3.5'),

('Forest Gump','3.9'),

('Inception','4.9');

**INSERT** **INTO** Customers **VALUES**('456123','Daniel Smith','ds@gmail.com'),

('827162','Roger Williams','rw@gmail.com'),

('198256','Jon Snow','js@gmail.com'),

('049821','Angela Jones','aj@gmail.com');

**INSERT** **INTO** Orders **VALUES**('2594','456123','12/09/2019','23.1'),

('3412','827162','05/01/2020','2.1'),

('8972','198256','26/08/2020','17.3'),

('5119','049821','18/03/2020','12.2');

**INSERT** **INTO** Rentals **VALUES**('2594','Gamer','1'),

('2594','Dark Waters','2'),

('2594','Inception','3'),

('3412','A Space Odyssey','1'),

('8972','Dark Waters','1'),

('8972','Taxi Driver','2'),

('8972','The Dark Knight','2'),

('8972','Forest Gump','1'),

('5119','Dark Waters','1'),

('5119','Gamer','3');

This code above shows creating tables (CREATE TABLE xxxxx),

Adding primary(xxxxx DATATYPE PRIMARY KEY),

Foreign keys((first we initialize them (xxxxx DATATYPE) then FOREIGN KEY(xxxxx) REFERENCES tablenamefromwhereisprimarykey(xxxxx)))

Populating tables with data(INSERT INTO tablename VALUES(‘1’,’test’,’example’))

This database will be used for the queries in 1C parts.

# 1 C Task

## Part 1. – Find the customers that have rented films priced £2.6 – output theirID and name.

**SELECT** Customer\_ID, Customer\_Name

**FROM** Customers

**WHERE** Customer\_ID **IN** (**SELECT** Customer\_ID **FROM** Orders **WHERE** Rental\_ID **IN** (**SELECT** Rental\_ID **FROM** Rentals **WHERE** Film\_Name = (**SELECT** Film\_Name **FROM** Films **WHERE** Film\_Price = '2.6')))

First what code does is it gets Film\_Name rows where their matching Film\_Price is 2.6 pounds. Then as we have those Film\_Name values, we get Rental\_ID values by comparing if Film\_Name is equal to the Film\_Name values we have from the previous step. As we have Rental\_ID values then if the Rental\_ID values are IN the values of Rental\_ID that we have from the previous step, we will get Customer\_ID values from the Orders table. Then we look for Customer\_ID and Customer\_Name values which will be the final output where Customer\_ID will be IN Customer\_ID values we have from the previous step.

## Part 2. Find the customers that spent less than £3 on individual film rentals, but have spent a total higher than £15 - output their ID and name.

**SELECT** Customer\_ID **AS** 'Customer ID', Customer\_Name **AS** 'Customer Name'

**FROM** Customers

**WHERE** Customer\_ID **IN** (**SELECT** Customer\_ID **FROM** Orders **WHERE** Total\_Price **IN** (**SELECT** Total\_Price **FROM** Orders **WHERE** Rental\_ID **IN** (**SELECT** Rental\_ID **FROM** Rentals **WHERE** Film\_Name **IN** (**SELECT** Film\_Name **FROM** Films **WHERE** Film\_Price < '3') **AND** Total\_Price > '15')))

For this part. First, we get Film\_Name values from the Films table where their Film\_Price is less than 3 pounds. Then we get Rental\_ID values from the Rentals table where the Film\_Name is IN the Film\_Name values we got from the previous step, but as we need the Total\_Price more than 15 pounds the comparison happens within the Orders table and values that are eligible are selected to the next step. As we will have Customer\_ID values from the Orders table we will be able to output Customer\_ID and Customer\_Name values.

## Part 3. - Find the customers that have rented the film “Gamer” – output their ID and name.

**SELECT** Customer\_ID **AS** 'Customer ID', Customer\_Name **AS** 'Customer Name'

**FROM** Customers

**WHERE** Customer\_ID **IN** (**SELECT** Customer\_ID **FROM** Orders **WHERE** Rental\_ID **IN** (**SELECT** Rental\_ID **FROM** Rentals **WHERE** Film\_Name = 'Gamer'))

First, we need to get Rental\_ID values from the Rentals table where the Film\_Name is Gamer. As we have Rental\_ID values we get Customer\_ID values by doing where Rental\_ID is in Rental\_ID values from the previous step. Then we have to output Customer\_ID and Customer\_Name from the Customers table where the Customer\_ID matches the Customer\_ID values we have got from the previous step.

## Part 4. - Find all films that are currently rented – output their name.

**SELECT** **DISTINCT** Film\_name **AS** 'Currently Rented Films' **FROM** Rentals

\*As reference I had to get the only word “DISTINCT” from the website that is listed in references\*

DISTINCT refers to unique values. Just outputting Film\_Name values from Rentals table that are currently rented.

## Part 5. - Find the film that has the highest number of rentals – output their name

**SELECT** **SUM**(Quantity) **FROM** Rentals **GROUP** **BY** Film\_Name;

Gets the sum of Quantity of films in Rentals table. As I have multiple values I wanted just to write MAX() query but the SQL does not like that. I cannot get this part done.

Reference for this part is listed in references.

# Conclusion

Overall, it was fun to do this coursework, so far by writing the report and doing the tasks gave me a lot of practice in justification, database normalization, designing an entity-relationship diagram, writing SQL code. I will need all those skills in the future.

# References

*SQL SELECT DISTINCT, COUNT, ROWS - with Examples - Dofactory*. (n.d.). Dofactory.Com. Retrieved 6 April 2021, from https://www.dofactory.com/sql/select-distinct#:%7E:text=How%20do%20I%20return%20unique,%2C%20AVG%2C%20MAX%2C%20etc

Diwan, A. (2019, September 3). *Get the sum of columns for duplicate records in MySQL*. Tutorialspoint.Com. https://www.tutorialspoint.com/get-the-sum-of-columns-for-duplicate-records-in-mysql