DBMS Project Report

PES University

Database Management Systems

UE18CS252

Submitted By

PES1201801749 Thushar V Karanth

This project is based on building database for movie rental system. The idea is to automate things for helping to manage data. This database keeps track of stores across the state, employees working in each store, customer details, movie details and payment details. Employee can find it easy to search for movie that customer wants and can see if that is available. It helps the store manager to keep track of the active customers and provide discount to them. The customer will have a wide variety of choices like he/she can choose a movie based on the actor, language or category. Even he gets to pick the movies which are in his budget.

All these tasks are difficult to manage and it will become hard for the employee to manage a huge number of customers. Hence this requires a properly designed database to keep track of payments, customers, employees and movies. This report tries to ease the process of assigning required resources. This database uses essential concepts of database management such as designing an ER model having essential entities and required attributes, finding functional dependencies and normalizing the model to avoid data anomalies.

This model has six entities which are Customer, Employee, Film, Store, Actor and Payment. These include customer details, employee details, film details, store details, actor details and payment info respectively. This concludes the introduction for this data model.

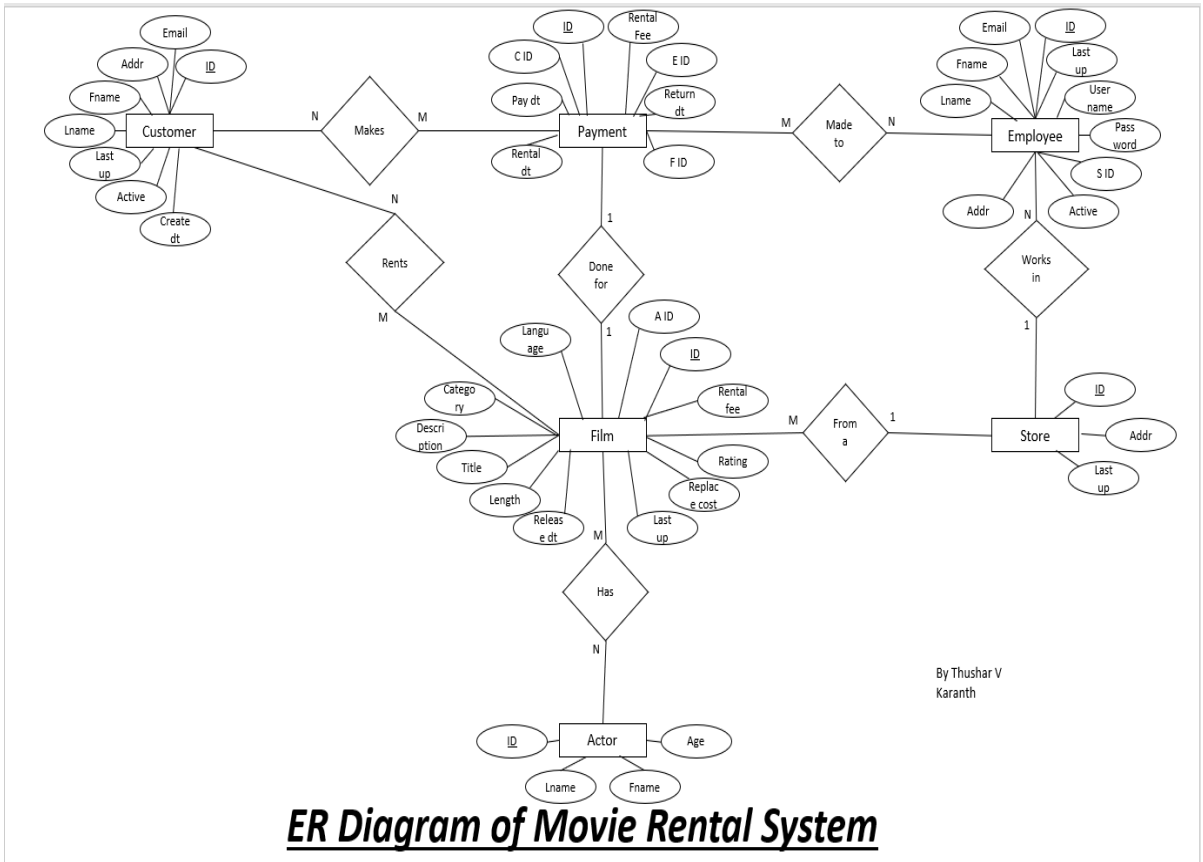
Index

Introduction

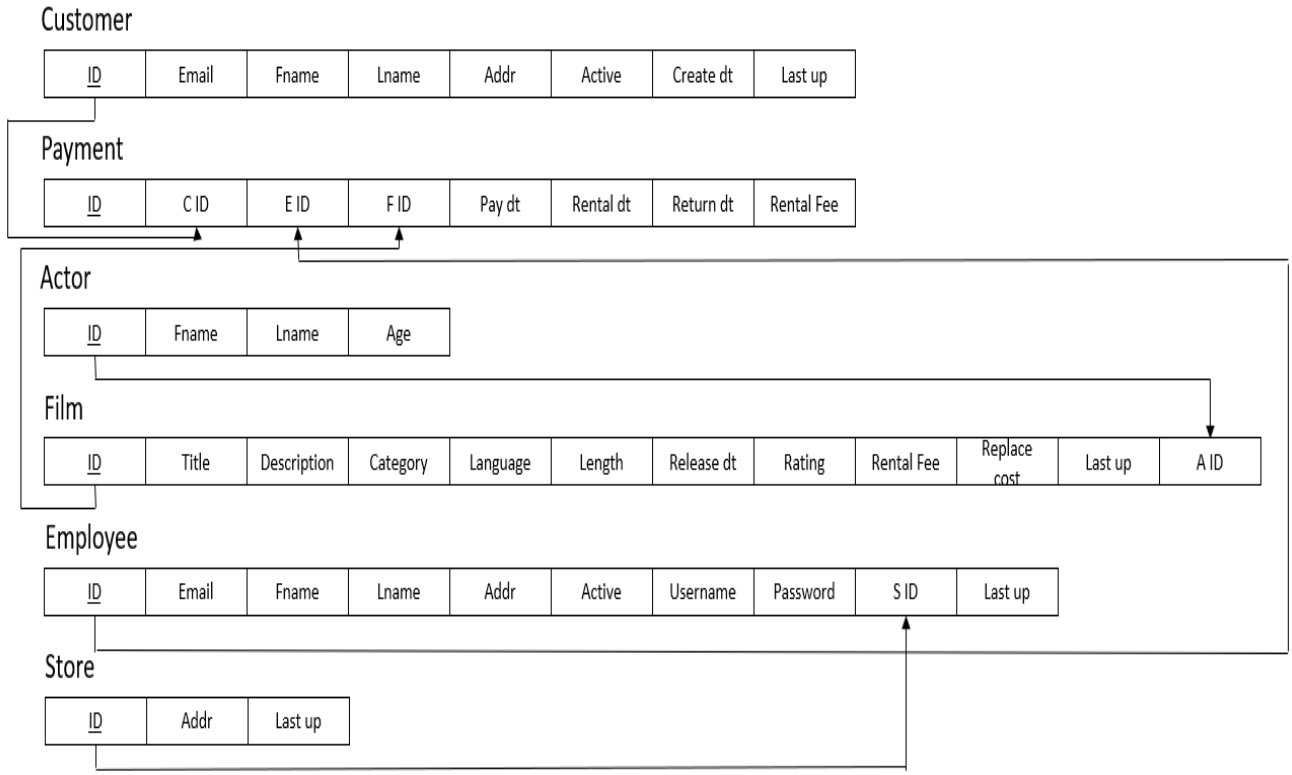
This project is about building a database for the movie rental system. This model has six entities. The entities give the details about the customers, employees, store, actors, films and payments. Each entity has number of attributes which gives characteristics of the entity.

Data Model

The ER Diagram for movie rental system is below:



The schema from the ER Diagram is shown below:



Before Normalization

Datatypes of attributes of entities:

Customer:
ID varchar(6)
Email varchar(320)
Fname varchar(32)
Lname varchar(32)
Addr varchar(128)
Active varchar(1)
Create_dt Date
Last_up Date

Payment:
ID varchar(6)
C_ID varchar(6)
E_ID varchar(6)
F_ID varchar(6)
Pay_dt Date
Rental_dt Date
Return_dt Date

Actor:
ID varchar(6)
Fname varchar(32)
Lname varchar(32)
Age int

Film:
ID varchar(6)
Title varchar(32)
Description varchar(128)
Category varchar(32)
Language varchar(32)
Length int
A_ID varchar(6)
Release_dt Date
Rating int
Rental_fee int
Replace_cost int
Last_up Date

Employee:
ID varchar(6)
Email varchar(320)
Fname varchar(32)
Lname varchar(32)
Addr varchar(128)
Active varchar(1)
Username varchar(32)
Password varchar(32)
S_ID varchar(6)
Last_up Date

Store:
ID varchar(6)
Addr varchar(128)
Last_up Date

Keys of the schema:

- Customer has ID as its primary key.
- Payment has ID as its primary key and has three foreign keys that are C_ID (Customer ID), E_ID (Employee ID) and F_ID (Film ID).
- Actor has ID as its primary key.
- Film has ID as its primary key and A_ID (Actor ID) as foreign key from Actor.
- Employee has ID as its primary key and S_ID (Store ID) as foreign key from Store. Email is a candidate key.
- Store has ID as its primary key.

FD and Normalization

Functional decencies before normalization is shown below:

Customer:
ID -> (Email, Fname, Lname, Addr, Active, Create dt, Last dt)

Payment:
ID -> (C ID, F ID, E ID, Pay dt, Return dt, Rental dt, Rental fee)
F ID -> (Rental Fee)

Employee:
ID -> (Email, Fname, Lname, Addr, S ID, Active, Last up)
Email -> (Username, Password)

Store:
ID -> (Addr, Last up)

Film:
ID -> (Title, Description, Length, Category, Rating, Replace cost, Release dt, Language, Rental fee, Last up, A ID)

Actor:
ID -> (Fname, Lname, Addr)

Functional Dependencies before Normalization

1NF:

The following model is in first normal form since it satisfies all these properties

- They have single valued attributes
- Unique name for each attribute
- Attribute domain is not changed
- Order of tuples does not matter

From FD we can see that A_ID in film can have multiple values, so we can put it into many rows.

2NF:

For a model to be in 2NF it should satisfy these properties:

- It should be of 1NF
- There should not be any partial dependency

From FD we can see that it is not possible to have partial dependency in the Customer, Actor, Store entities. In Payment and Employee none of the other attributes is depended on

only one of the composite keys hence there is no partial dependency. But Film's attribute A_ID and ID can be made into another table.

3NF:

For a model to be in 3NF it should satisfy these properties:

- It should be in 2NF
- There should be no transitive dependency

In Employee entity we can see that username and password will depend upon email rather than on ID hence this can lead do redundant values. So we need a separate table having these attributes to remove transitive property.

In Payment the Rental_fee depends upon the F_ID rather than on ID. Since in Film there is already an attribute called Rental_fee, which will be same as Rental_fee in Payment we can simply drop the attribute in Payment.

Here is the functional decencies after normalization:

Customer:
ID -> (Email, Fname, Lname, Addr, Active, Create dt, Last dt)

Payment:
ID -> (C ID, F ID, E ID, Pay dt, Return dt, Rental dt)

Employee:
ID -> (Email, Fname, Lname, Addr, S ID, Active, Last up)

Store:
ID -> (Addr, Last up)

Film:
ID -> (Title, Description, Length, Category, Rating, Replace cost, Release dt, Language, Rental fee, Last up)

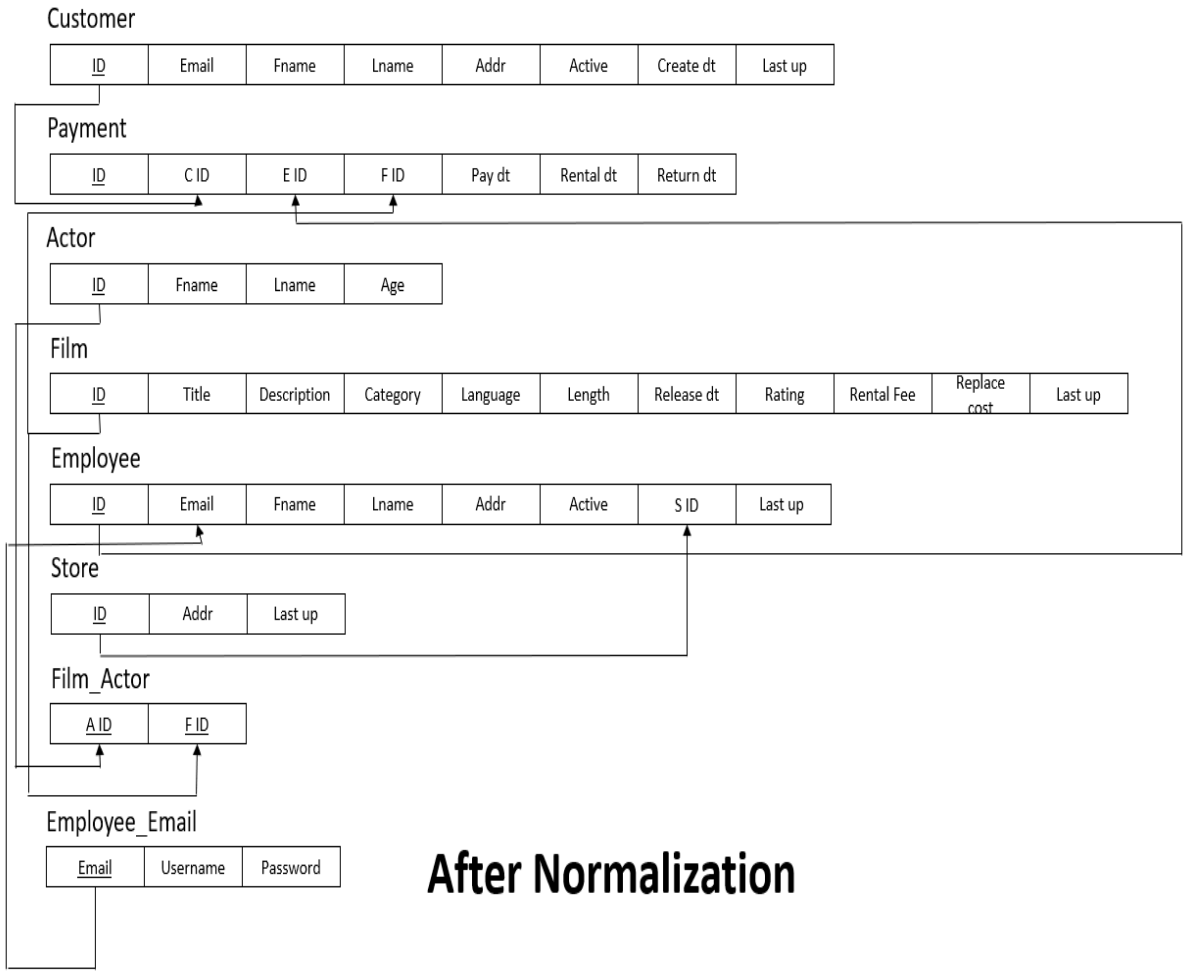
Actor:
ID -> (Fname, Lname, Addr)

Film_Actor:
(A ID, F ID) -> ()

Employee_Email:
Email -> (Username, Password)

Functional Dependencies after Normalization

Schema after normalization:



Lossless Join Property

If we decompose a relation R into relations R1 and R2,
To check for lossless join property using FD set, following conditions must hold:

1. Union of Attributes of R1 and R2 must be equal to attribute R. Each attribute of R must either be in R1 or in R2.
   `Att(R1) U Att(R2) = Att(R)`

2. Intersection of Attributes of R1 and R2 must not be NULL
   `Att(R1) ∩ Att(R2) ≠ Φ`

3. Common attribute must be a key for at least onr relation(R1 or R2)
   `Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)`

Consider the table Employee be R before normalization. After normalization the table is split into two i.e., Employee be R1 and Employee_Email be R2. Here if we apply lossless join property, then R1 union R2 gives R and R1 intersection R2 gives the Email and not NULL. So, the Email should be a primary key in either R1 or R2. Email is primary key in R2. So, this concludes that the join is lossless.

DDL

Screenshots of the creating the tables:

1. Customer

```
create table Customer (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                       Email varchar(320) not null CHECK (Email like '_%@__%.__%'),
                       Fname varchar(32) not null,
                       Lname varchar(32) not null,
                       Addr varchar(128) not null,
                       Active varchar(1) not null CHECK (Active IN ('Y', 'N')),
                       Create_dt Date not null,
                       Last_up Date not null,
                       primary key (ID))
```

2. Actor

```
create table Actor (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                    Fname varchar(32) not null,
                    Lname varchar(32) not null,
                    Age int not null CHECK (Age <=100 and Age >=1),
                    primary key (ID))
```

3. Store

```
create table Store (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                    Addr varchar(128) not null,
                    Last_up Date not null,
                    primary key(ID))
```

4. Film

```
create table Film (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                   Title varchar(32) not null,
                   Descr varchar(128) not null,
                   Cate varchar(32) not null,
                   Lang varchar(32) not null,
                   Leng int not null,
                   Release_dt Date not null,
                   Rating int not null CHECK (Rating>=1 and Rating<=10),
                   Rental_fee int not null,
                   Replace_cost int not null,
                   Last_up Date not null,
                   primary key (ID))
```

5. Employee_Email

```
create table Employee_Email(Email_ID varchar(320) not null CHECK (Email_ID like '_%@__%.__%'),
                            Username varchar(32) not null,
                            Password varchar(32) not null CHECK (Password like '%[0-9]%' AND
                            Password like '%[A-Z]%' collate Latin1_General_BIN2 AND
                            Password like '%[!@#$%^&*()-_+=.,;:~]%' AND
                            len(Password)>=(8)),
                            primary key(Email_ID))
```

6. Employee

```
create table Employee (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                       Email varchar(320) not null CHECK (Email like '_%@__%.__%'),
                       Fname varchar(32) not null,
                       Lname varchar(32) not null,
                       Addr varchar(128) not null,
                       Active varchar(1) not null CHECK (Active IN ('Y', 'N')),
                       S_ID varchar(6) not null CHECK (S_ID like replicate('[0-9]', 6)),
                       Last_up Date not null,
                       primary key (ID),
                       foreign key(S_ID) references Store(ID),
                       foreign key(Email) references Employee_Email(Email_ID))
```

7. Film_Actor

```
create table Film_Actor (A_ID varchar(6) not null CHECK (A_ID like replicate('[0-9]', 6)),
                         F_ID varchar(6) not null CHECK (F_ID like replicate('[0-9]', 6)),
                         primary key(A_ID, F_ID),
                         foreign key(A_ID) references Actor(ID),
                         foreign key(F_ID) references Film(ID))
```

8. Payment

```
create table Payment (ID varchar(6) not null CHECK (ID like replicate('[0-9]', 6)),
                      C_ID varchar(6) not null CHECK (C_ID like replicate('[0-9]', 6)),
                      E_ID varchar(6) not null CHECK (E_ID like replicate('[0-9]', 6)),
                      F_ID varchar(6) not null CHECK (F_ID like replicate('[0-9]', 6)),
                      Pay_dt Date not null,
                      Rental_dt Date not null,
                      Return_dt Date not null,
                      primary key(ID),
                      foreign key(C_ID) references Customer(ID),
                      foreign key(E_ID) references Employee(ID),
                      foreign key(F_ID) references Film(ID))
```

Triggers

1. This Trigger is executed when values are inserted into or updated in Payment table. This checks for the following conditions:
    - Customer is not there in table and if he is not active
    - Employee is not there in table and if he is not active
    - Film is present or not
    - If the Rental_dt is greater than Return_dt
    - If the Return_dt is more than 31 days from Rental_dt it tells to pay more

```
create or alter trigger trigger_insert_payment on Payment after insert, update as
begin
declare @Msg varchar(255);
set @Msg = '';
if NOT EXISTS (SELECT 1 FROM Employee,Inserted WHERE Employee.ID = Inserted.E_ID)
set @Msg = ' Employee not found ';
else
begin
declare @Employee_State varchar(1)
select @Employee_State = E.Active from Employee E,Inserted where E.ID = Inserted.E_ID
if(@Employee_State= 'N')
set @Msg = ' Employee not working currently ';
end
if NOT EXISTS (SELECT 1 FROM Customer,Inserted WHERE Inserted.C_ID = Customer.ID)
set @Msg = @Msg + ' Customer not found ';
else
begin
declare @Customer_State varchar(1)
select @Customer_State = Customer.Active from Customer ,Inserted where Customer.ID = Inserted.C_ID
if(@Customer_State= 'N')
set @Msg = @Msg + ' Customer not found ';
end
if NOT EXISTS (SELECT 1 FROM Film,Inserted WHERE Inserted.F_ID = Film.ID)
set @Msg = @Msg + ' Film not found ';
declare @Rental_dt date;
declare @Return_dt date;
select @Rental_dt = Inserted.Rental_dt from Inserted
select @Return_dt = Inserted.Return_dt from Inserted
if(DATEDIFF(day, @Rental_dt, @Return_dt)>31)
set @Msg = @Msg + ' You have to pay extra money ';
if(@Rental_dt > Return_dt)
set @Msg = Msg + 'Invalid Dates'
if(@Msg<> '')
throw 51000, @Msg,1;
end
```

2. This trigger is executed when the Employee_Email is executed
    - First we need to create a table named Encode_Pass

```
create table Encode_Pass (Password varchar(32) not null CHECK (Password like '%[0-9]%' AND
                          Password like '%[A-Z]%' collate Latin1_General_BIN2 AND
                          Password like '%[!@#$%^&*()-_+=.,;:~]%' AND
                          len(Password)>=(8)),
                          Encode varchar(max) not null,
                          primary key(Password))
```

    - This trigger takes Password and then encrypt into BASE64 version and insert the values into Encode_Pass

```
create or alter trigger trigger_insert_Employee_Email on Employee_Email after insert, update as
begin
declare @Password varchar(32);
select @Password = Inserted.Password from Inserted

declare @source varbinary(max), @encoded varchar(max), @decoded varbinary(max)
set @source = convert(varbinary(max), @Password)
set @encoded = cast('' as xml).value('xs:base64Binary(sql:variable("@source"))', 'varchar(max)')
set @decoded = cast('' as xml).value('xs:base64Binary(sql:variable("@encoded"))', 'varbinary(max)')
select
convert(varchar(max), @source) as source_varchar,
@source as source_binary,
@encoded as encoded,
@decoded as decoded_binary,
convert(varchar(max), @decoded) as decoded_varchar
insert into Encode_Pass values(@Password,@encoded)
end
```

- The output after inserting into Encode_Pass

	Password	Encode
1	Password0!	UGFzc3dvcmQwIQ==
2	Password1!	UGFzc3dvcmQxIQ==
3	Password2!	UGFzc3dvcmQyIQ==
4	Password3!	UGFzc3dvcmQzIQ==
5	Password4!	UGFzc3dvcmQ0IQ==
6	Password5!	UGFzc3dvcmQ1IQ==
7	Password6!	UGFzc3dvcmQ2IQ==
8	Password7!	UGFzc3dvcmQ3IQ==
9	Password8!	UGFzc3dvcmQ4IQ==
10	Password9!	UGFzc3dvcmQ5IQ==

SQL Queries

1. Nested query:
   This query gets all the customers who are Active and they have made payment atleast once

```
select Customer.ID, Customer.Fname
from Customer
where Customer.Active = 'Y'
and Customer.ID in (select Payment.C_ID from Payment)
```

2. Aggregate query:

   This query gets the count of all the films which are 3 years or older

```
select  count(*) as count
from Film
where DATEDIFF(year,Film.Release_dt,getdate()) > =3
```

This query gets number of actors in each film

```
select Film_Actor.F_ID, Film.Title, count(*) as num
from Film_Actor,Film
where Film.ID = Film_Actor.F_ID
group by Film_Actor.F_ID,Film.Title
```

3. Outer join query:

This query to select ID, Username and Fname of employee who is active

```
select Employee.ID, Employee_Email.Username, Employee.Fname
from Employee full outer join Employee_Email
on Employee.Email = Employee_Email.Email_ID
where Employee.Active='Y'
```

Conclusion

This model of Movie rental system tries to imitate the actual scenario in the real world example. This can handle nearly all the ambiguities. There are some parts where this database may fail i.e., handling security. This project can further be taken forward and give a frontend and expand the model to storing more information in tables (attributes). This concludes the project report.