DBMS Project Report

PES University

Database Management Systems

UE18CS252

Submitted By

|  |  |
| --- | --- |
| PES1201801956 | Kaustubh Kulkarni |

|  |
| --- |
| The database created is of a library. The entities present are branch, customer, customer\_subscription, author, books, borrowed. These entities are connected by meaningful relationships. Triggers have been set for borrowed entity, as well as the other entities for the names, dates, etc. Transactions made include updation of the borrowed table as a customer borrows books. The subscription and customer table also can be updated when a new customer subscribes to the library.  Capabilities of this system include automatically calculating the due date of a borrowed book, the fine to be payed if needed, the return status of that particular book. It can also show which books are in which branch of the library, along with the addresses of these branches, with a correct query. |

[**Introduction**](#_gwqiqapyxr0x) **2**

[**Data Model**](#_p451stoz4u81) **2**

[**FD and Normalization**](#_wc0v2611q54c) **2**

[**DDL**](#_pn441yprtk2f) **3**

[**Triggers**](#_a9bswny9ffj) **3**

[**SQL Queries**](#_exvna2br67rz) **3**

[**Conclusion**](#_k6bbxckcf51j) **3**

# Introduction

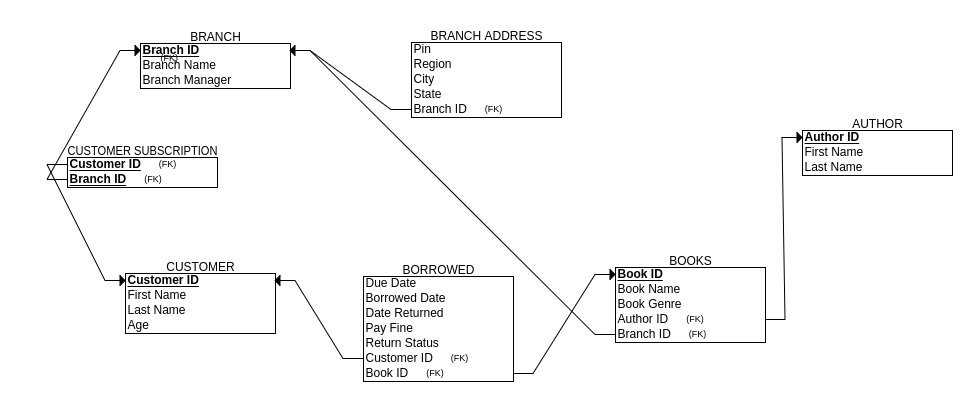
The miniworld chosen is of a library. The entities present are branch, branch\_address, customer, customer\_subscription, author, books, borrowed. The branch and branch\_address entities hold the branch ID and the corresponding branch details. Customer entity holds details of the customer. Author holds details of the author. Books holds the book details along with the author ID of the one who wrote the book. Here, we are assuming that only one author works on books. There are no co-authors for any books. Borrowed holds the logs for any book borrowed by customers.

Transactions of the system include creation of all the tables, creating the triggers, inserting and updating the tables with relevant information.

# Data Model

The branch and customer have an associative entity customer\_subscription between them. The books and author are connected with a relationship that the author writes (a) book(s). The borrowed entity is connected to customer and books entities as the customer borrows books from the books table. The primary keys in their respective tables are branchID, customerID, bookID and authorID. These act as foreign keys for a few other tables. The primary keys, names, addresses ,etc are all of type varchar. The pin in the address is of integer type. The borrowedDate, dueDate, dateReturned attributes are of type date.

RELATIONAL SCHEMA



ER DIAGRAM

# 

# FD and Normalization

A 2NF may be violated when there is a non-prime attribute dependent on the proper subset of any candidate key of the table. Meaning, the non-prime attribute should not depend on the prime attribute of the table. A 3NF may be violated when any non-prime attribute is transitively dependent on the candidate key.

Functional dependencies in my tables are as follows:

branchID -> branchName,branchManager

branchID -> pin,region,city,state

customerID -> firstName,lastName,age

customerID -> branchID

authorID -> firstName,lastName

bookID -> bookName,bookGenre

bookID -> branchID

bookID -> authorID

customerID -> dueDate,borrowedDate,dateReturned

customerID -> bookID

All relations are in Boyce Codd NF (BCNF).

Hypothetical examples of violations:

If I had columns within the branch entity, specifying the bookIDs along with the book details, it would violate 3NF.

If I also had book details in the author entity, it would violate 3NF.

If branch details were in customer\_subscription, it would violate 3NF.

# DDL

CREATE TABLE branch (

branchID varchar(10) NOT NULL,

branchName varchar(255) NOT NULL,

branchManager varchar(255),

CONSTRAINT PK\_branchID PRIMARY KEY (branchID)

);

CREATE TABLE branch\_address (

branchID varchar(10) NOT NULL,

pin int(7),

region varchar(255),

city varchar(255) NOT NULL,

state varchar(255) NOT NULL,

CONSTRAINT FK\_branchIDInAddress FOREIGN KEY (branchID) REFERENCES branch(branchID)

);

CREATE TABLE customer (

customerID varchar(10) NOT NULL,

firstName varchar(255) NOT NULL,

lastName varchar(255),

age int(3),

CHECK (age>=4),

CONSTRAINT PK\_customerID PRIMARY KEY (customerID)

);

CREATE TABLE customer\_subscription (

customerID varchar(10) NOT NULL,

branchID varchar(10) NOT NULL,

CONSTRAINT FK\_branchIDInSubscription FOREIGN KEY (branchID) REFERENCES branch(branchID),

CONSTRAINT FK\_customerIDInSubscription FOREIGN KEY (customerID) REFERENCES customer(customerID)

);

CREATE TABLE author (

authorID varchar(10) NOT NULL,

firstName varchar(255) NOT NULL,

lastName varchar(255),

CONSTRAINT PK\_authorID PRIMARY KEY (authorID)

);

CREATE TABLE books (

bookID varchar(10) NOT NULL,

bookName varchar(255) NOT NULL,

bookGenre varchar(255) NOT NULL,

authorID varchar(10) NOT NULL,

branchID varchar(10) NOT NULL,

CONSTRAINT PK\_bookID PRIMARY KEY (bookID),

CONSTRAINT FK\_authorIDInBooks FOREIGN KEY (authorID) REFERENCES author(authorID),

CONSTRAINT FK\_branchIDInBooks FOREIGN KEY (branchID) REFERENCES branch(branchID)

);

CREATE TABLE borrowed (

customerID varchar(10) NOT NULL,

bookID varchar(10) NOT NULL,

borrowedDate date NOT NULL,

dueDate date NOT NULL,

dateReturned date,

payFine int(5),

returnStatus varchar(255) NOT NULL,

CHECK (borrowedDate <= dateReturned),

CONSTRAINT FK\_bookIDInBorrowed FOREIGN KEY (bookID) REFERENCES books(bookID),

CONSTRAINT FK\_customerIDInBorrowed FOREIGN KEY (customerID) REFERENCES customer(customerID)

);

# Triggers

Created triggers to automate the process of generating due dates, the fine to be payed by a customer, and whether the book is returned or not. Also created triggers to validate dates and capitalize names wherever desired. There is also a limit that a customer can only borrow 7 books at a time. A trigger is also set off if a book is being borrowed that has not been returned yet.

DROP TRIGGER IF EXISTS customer\_name;

DROP TRIGGER IF EXISTS book\_return\_trigger;

DROP TRIGGER IF EXISTS branch\_manager\_name;

DROP TRIGGER IF EXISTS author\_name;

DROP TRIGGER IF EXISTS set\_due\_date;

DROP TRIGGER IF EXISTS return\_status\_modify;

DROP TRIGGER IF EXISTS date\_violation;

DROP TRIGGER IF EXISTS borrow\_limit;

DROP TRIGGER IF EXISTS book\_constraints;

DELIMITER //

CREATE TRIGGER customer\_name

BEFORE INSERT ON customer

FOR EACH ROW

BEGIN

SET NEW.firstName = CONCAT(UPPER(SUBSTRING(NEW.firstName, 1, 1)), LOWER(SUBSTRING(NEW.firstName, 2)));

IF NEW.lastName IS NOT NULL THEN

SET NEW.lastName = CONCAT(UPPER(SUBSTRING(NEW.lastName, 1, 1)), LOWER(SUBSTRING(NEW.lastName, 2)));

END IF;

END//

CREATE TRIGGER author\_name

BEFORE INSERT ON author

FOR EACH ROW

BEGIN

SET NEW.firstName = CONCAT(UPPER(SUBSTRING(NEW.firstName, 1, 1)), LOWER(SUBSTRING(NEW.firstName, 2)));

IF NEW.lastName IS NOT NULL THEN

SET NEW.lastName = CONCAT(UPPER(SUBSTRING(NEW.lastName, 1, 1)), LOWER(SUBSTRING(NEW.lastName, 2)));

END IF;

END//

CREATE TRIGGER branch\_manager\_name

BEFORE INSERT ON branch

FOR EACH ROW

BEGIN

SET NEW.branchManager = UPPER(NEW.branchManager);

END//

CREATE TRIGGER book\_constraints

BEFORE INSERT ON borrowed

FOR EACH ROW

BEGIN

DECLARE nBooks int(1);

SET nBooks= (SELECT COUNT(bookID) FROM borrowed WHERE customerID = NEW.customerID AND returnStatus = 'NOT RETURNED');

IF (nBooks > 7) THEN

SIGNAL SQLSTATE '46000'

SET MESSAGE\_TEXT = 'Cannot borrow more than 7 books.';

END IF;

END//

CREATE TRIGGER borrow\_limit

BEFORE INSERT ON borrowed

FOR EACH ROW

BEGIN

DECLARE bID varchar(10);

SET bID = (SELECT bookID FROM borrowed WHERE bookID = NEW.bookID AND returnStatus = 'NOT RETURNED');

IF bID IS NOT NULL THEN

SIGNAL SQLSTATE '47000'

SET MESSAGE\_TEXT = 'Cannot borrow book. Already borrowed and not yet returned.';

END IF;

END//

CREATE TRIGGER set\_due\_date

BEFORE INSERT ON borrowed

FOR EACH ROW

BEGIN

SET NEW.dueDate = DATE\_ADD(NEW.borrowedDate, interval 14 day);

END//

CREATE TRIGGER book\_return\_trigger

BEFORE UPDATE ON borrowed

FOR EACH ROW

BEGIN

IF NEW.dateReturned IS NOT NULL THEN

IF NEW.dateReturned > OLD.dueDate THEN

SET NEW.payFine = 10 \* DATEDIFF (NEW.dateReturned, OLD.dueDate);

END IF;

IF NEW.dateReturned < OLD.dueDate THEN

SET NEW.payFine = 0;

END IF;-

SET NEW.returnStatus = 'RETURNED';

END IF;

END//

CREATE TRIGGER date\_violation

BEFORE INSERT ON borrowed

FOR EACH ROW

BEGIN

IF (NEW.borrowedDate > NOW()) THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot insert a borrowed date that is in the future.';

END IF;

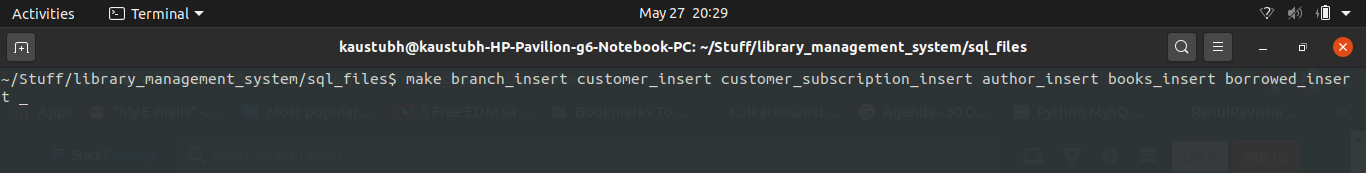
END//

DELIMITER ;

# SQL Queries

INSERTIONS

Screenshot of the command used:

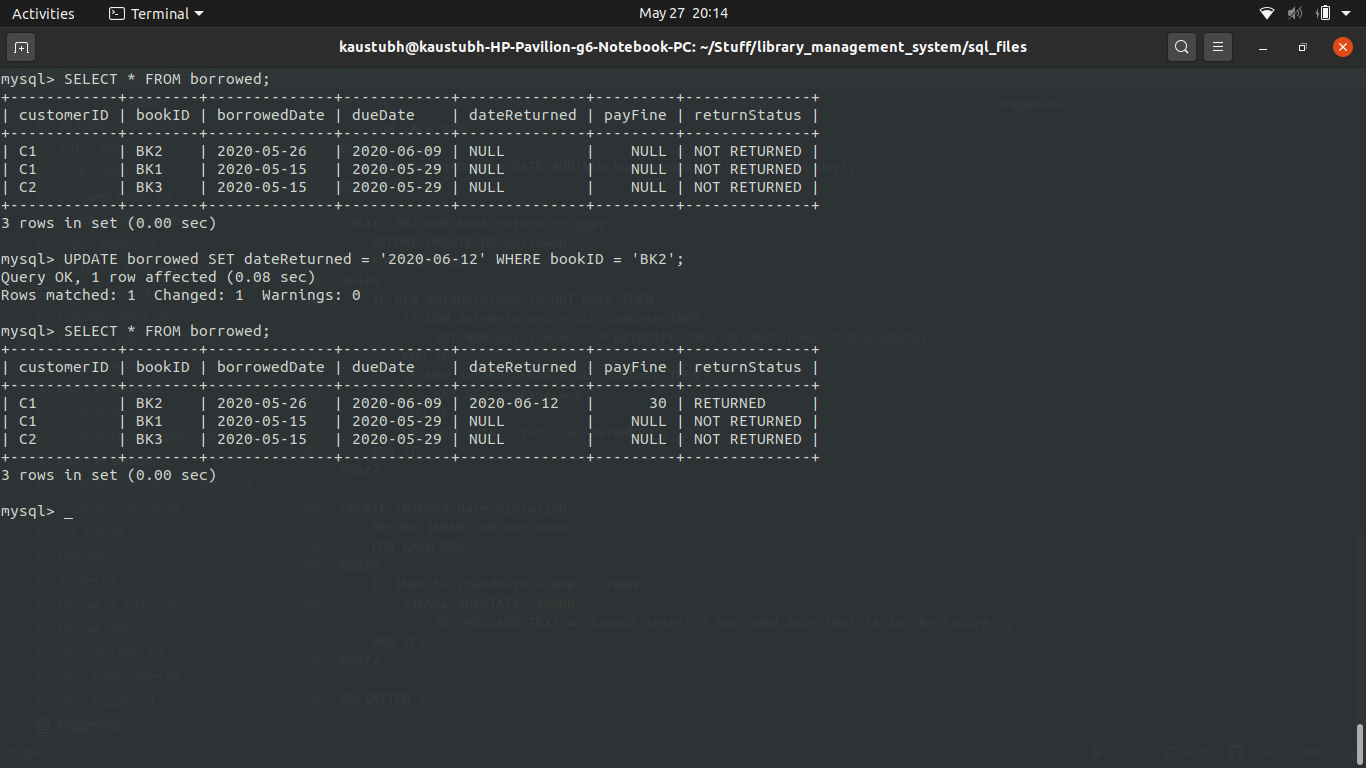


SIMPLE QUERIES

Sentence: Book with bookID ‘BK3’ has been returned on 2020-06-15. Update the borrowed table with relevant information.

UPDATE borrowed SET dateReturned = '2020-06-12' WHERE bookID = 'BK2';

(This updates the table with a returned date and the triggers take care of all other fields once the book with bookID ‘BK3’ is returned)



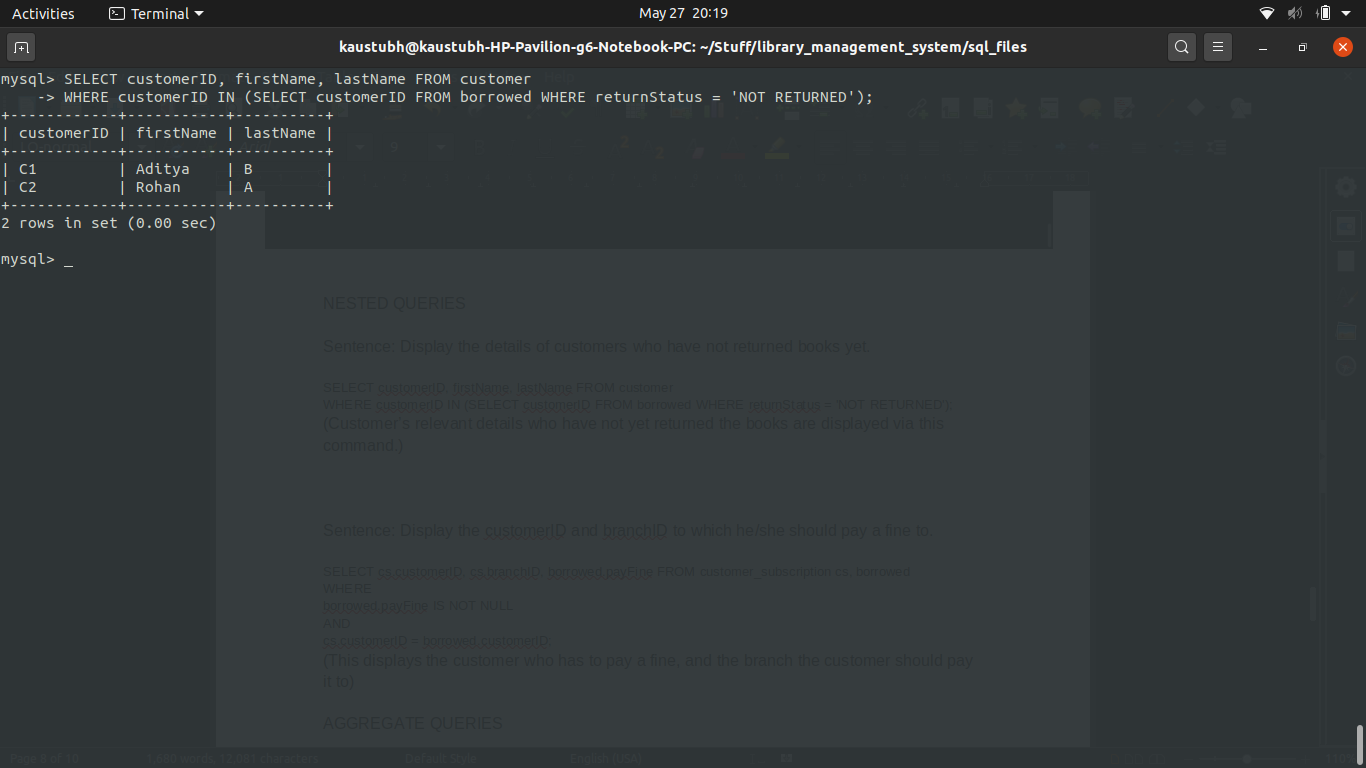
NESTED QUERIES

Sentence: Display the details of customers who have not returned books yet.

SELECT customerID, firstName, lastName FROM customer

WHERE customerID IN (SELECT customerID FROM borrowed WHERE returnStatus = 'NOT RETURNED');

(Customer’s relevant details who have not yet returned the books are displayed via this command.)



Sentence: Display the customerID and branchID to which he/she should pay a fine to.

SELECT cs.customerID, cs.branchID, borrowed.payFine FROM customer\_subscription cs, borrowed

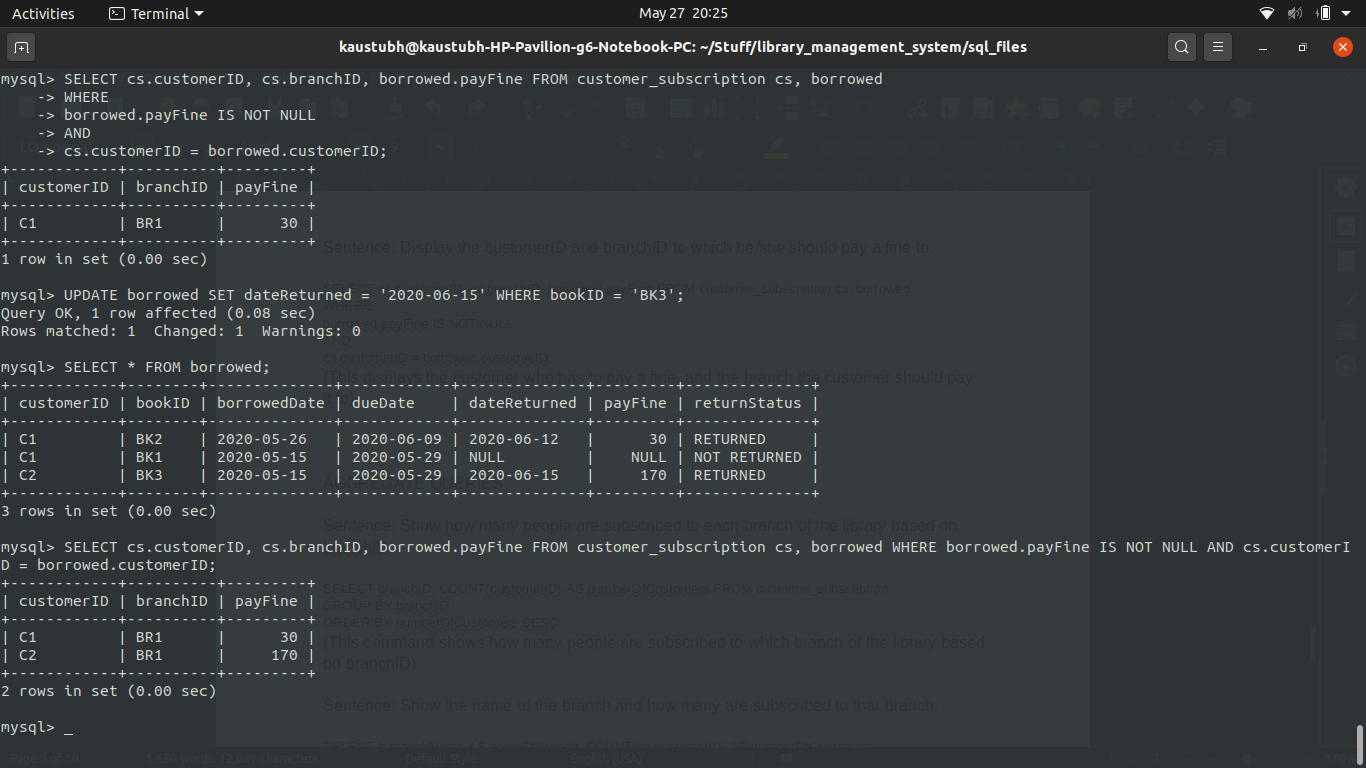
WHERE

borrowed.payFine IS NOT NULL

AND

cs.customerID = borrowed.customerID;

(This displays the customer who has to pay a fine, and the branch the customer should pay it to)



AGGREGATE QUERIES

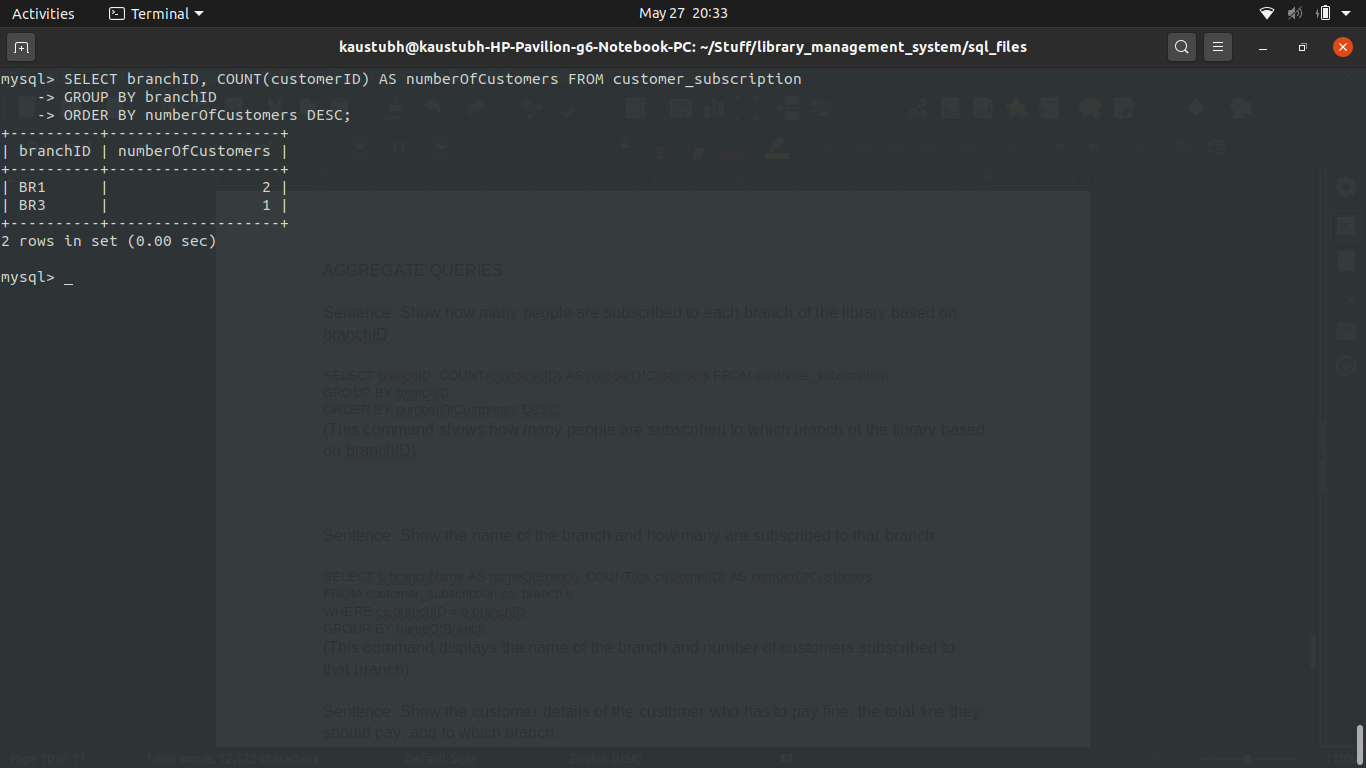
Sentence: Show how many people are subscribed to each branch of the library based on branchID.

SELECT branchID, COUNT(customerID) AS numberOfCustomers FROM customer\_subscription

GROUP BY branchID

ORDER BY numberOfCustomers DESC;

(This command shows how many people are subscribed to which branch of the library based on branchID)



Sentence: Show the name of the branch and how many are subscribed to that branch.

SELECT b.branchName AS nameOfBranch, COUNT(cs.customerID) AS numberOfCustomers

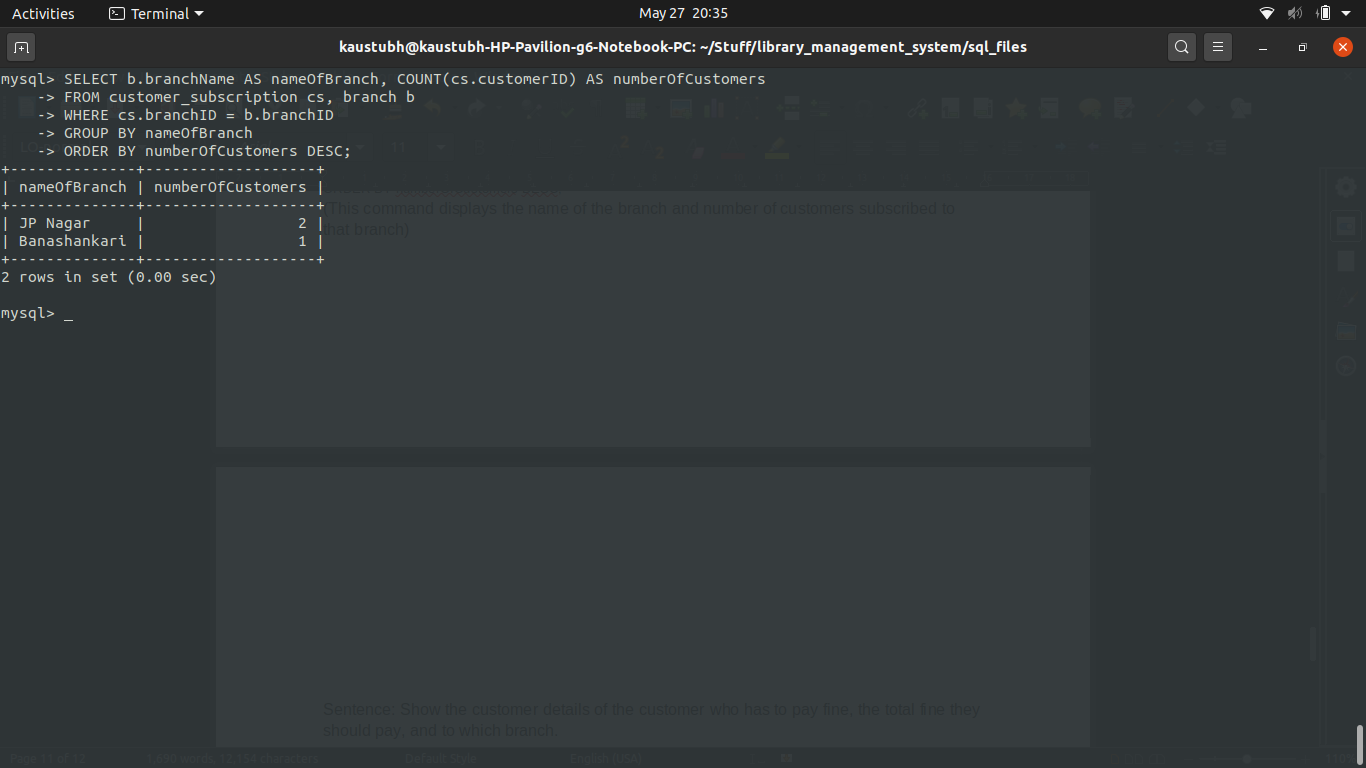
FROM customer\_subscription cs, branch b

WHERE cs.branchID = b.branchID

GROUP BY nameOfBranch

ORDER BY numberOfCustomers DESC;

(This command displays the name of the branch and number of customers subscribed to that branch)



Sentence: Show the customer details of the customer who has to pay fine, the total fine they should pay, and to which branch.

SELECT br.customerID, c.firstName, c.lastName, SUM(br.payFine) AS totalFine, b.branchName AS payHere

FROM borrowed br, customer c, branch b

WHERE

br.payFine IS NOT NULL

AND

br.customerID = c.customerID

AND

b.branchName IN (SELECT branchName FROM branch WHERE branchID IN (

SELECT branchID FROM customer\_subscription WHERE customerID IN (

SELECT customerID FROM borrowed WHERE payFine IS NOT NULL

)

)

)

GROUP BY br.customerID, b.branchName

ORDER BY totalFine ASC;

(This command is used to display the total fine each customer has to pay, the customer details, and where to pay the fine)

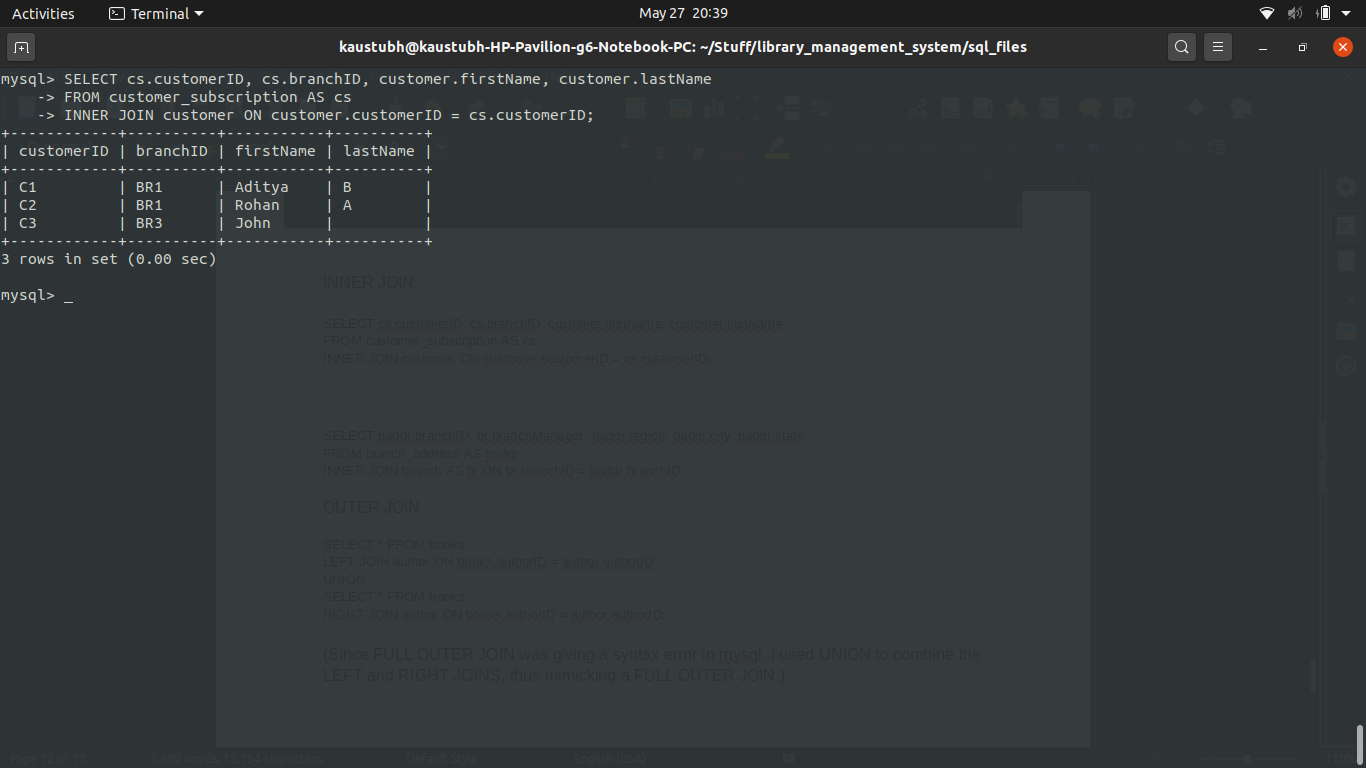


INNER JOIN

SELECT cs.customerID, cs.branchID, customer.firstName, customer.lastName

FROM customer\_subscription AS cs

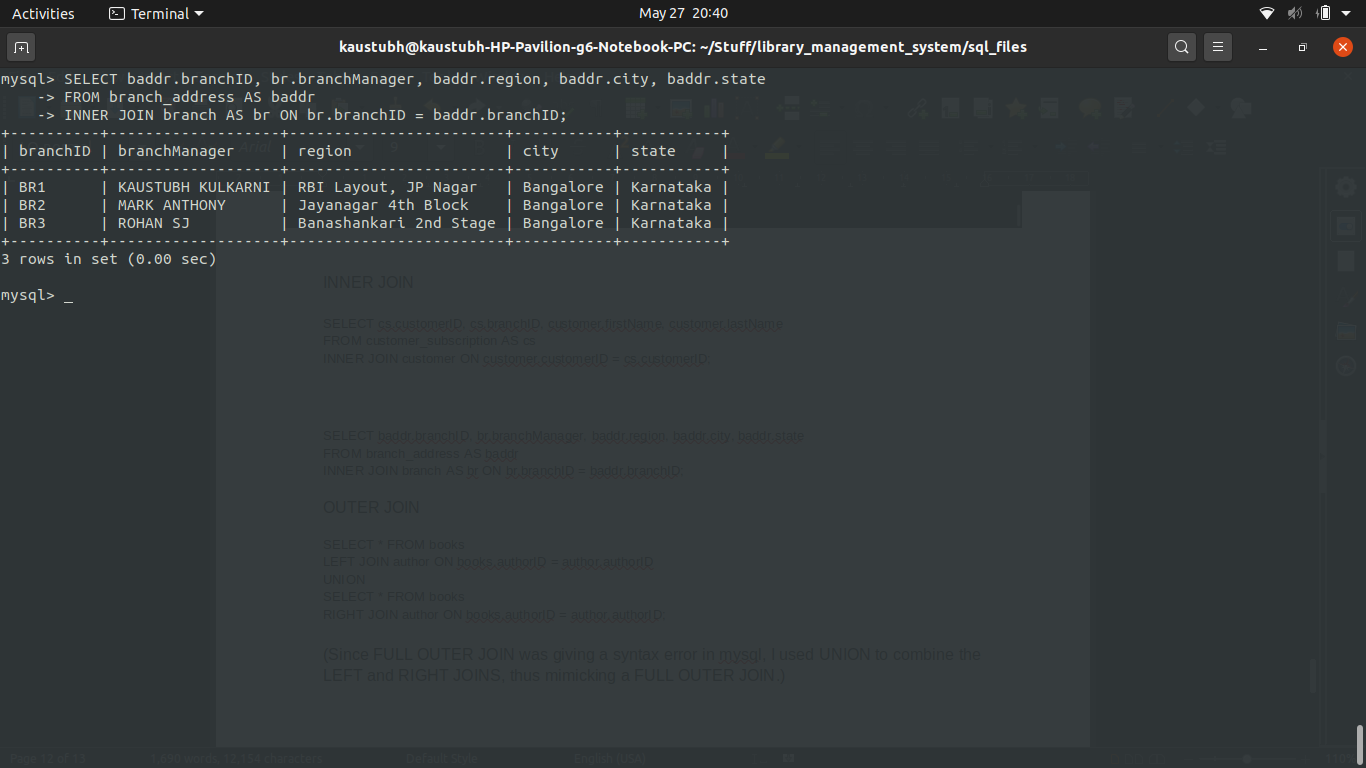
INNER JOIN customer ON customer.customerID = cs.customerID;



SELECT baddr.branchID, br.branchManager, baddr.region, baddr.city, baddr.state

FROM branch\_address AS baddr

INNER JOIN branch AS br ON br.branchID = baddr.branchID;



OUTER JOIN

SELECT \* FROM books

LEFT JOIN author ON books.authorID = author.authorID

UNION

SELECT \* FROM books

RIGHT JOIN author ON books.authorID = author.authorID;

(Since FULL OUTER JOIN was giving a syntax error in mysql, I used UNION to combine the LEFT and RIGHT JOINS, thus mimicking a FULL OUTER JOIN.)

# Conclusion

This system can be used to manage a small library quite efficiently. It can be used to track returned and not returned books. It can also track fines to be payed by customers, which branch has which book, etc. All in all, it is accurate and can be used on a small scale.

LIMITATIONS

There is no UI for customers or library management staff.

FUTURE ENHANCEMENTS

Keeping track of the books borrowed and returned, and if they are available in the particular branch or not.

Keeping one more table for premium customers who can keep the book for longer and also borrow more books.

Can store customer e-mail IDs and phone numbers to contact them in case of any clarification.