Create 3 tables: Movie, Member, and Movie\_Rent for a film renting company and insert 5 records to each.

-- Create the 'movie' table

CREATE TABLE movie (

movie\_id INT PRIMARY KEY,

title VARCHAR(100) NOT NULL,

director VARCHAR(100),

release\_year INT,

genre VARCHAR(50),

rating DECIMAL(3,1)

);

-- Insert records into the 'movie' table

INSERT INTO movie (movie\_id, title, director, release\_year, genre, rating) VALUES

(1, 'The Shawshank Redemption', 'Frank Darabont', 1994, 'Drama', 9.3),

(2, 'The Godfather', 'Francis Ford Coppola', 1972, 'Crime', 9.2),

(3, 'The Dark Knight', 'Christopher Nolan', 2008, 'Action', 9.0),

(4, 'Pulp Fiction', 'Quentin Tarantino', 1994, 'Crime', 8.9),

(5, 'Forrest Gump', 'Robert Zemeckis', 1994, 'Drama', 8.8);

-- Create the 'member' table

CREATE TABLE member (

member\_id INT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE,

phone VARCHAR(20)

);

-- Insert records into the 'member' table

INSERT INTO member (member\_id, first\_name, last\_name, email, phone) VALUES

(1, 'John', 'Doe', 'john@example.com', '123-456-7890'),

(2, 'Jane', 'Smith', 'jane@example.com', '456-789-0123'),

(3, 'Alice', 'Johnson', 'alice@example.com', '789-012-3456'),

(4, 'Bob', 'Brown', 'bob@example.com', '012-345-6789'),

(5, 'Emma', 'Davis', 'emma@example.com', '345-678-9012');

-- Create the 'movie\_rent' table

CREATE TABLE movie\_rent (

rental\_id INT PRIMARY KEY,

member\_id INT,

movie\_id INT,

rental\_date DATE,

return\_date DATE,

FOREIGN KEY (member\_id) REFERENCES member(member\_id),

FOREIGN KEY (movie\_id) REFERENCES movie(movie\_id)

);

-- Insert records into the 'movie\_rent' table

INSERT INTO movie\_rent (rental\_id, member\_id, movie\_id, rental\_date, return\_date) VALUES

(1, 1, 1, '2024-04-01', '2024-04-08'),

(2, 2, 3, '2024-04-03', '2024-04-10'),

(3, 3, 4, '2024-04-05', '2024-04-12'),

(4, 4, 2, '2024-04-07', '2024-04-14'),

(5, 5, 5, '2024-04-09', NULL);

Then, create Stored Procedures to implement following queries.

1Show the information of all movies that has been rented:

create procedure GetAllRentedMovies

AS

BEGIN

SELECT movie\_rent.rental\_id, movie.\*

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id;

END;

2Get all members who have rented any movie:

create procedure GetAllRentingMembers

AS

BEGIN

SELECT movie\_rent.rental\_id, member.\*

FROM member

INNER JOIN movie\_rent ON member.member\_id = movie\_rent.member\_id;

END;

3Get all rented movies along with their renting details by corresponding member:

CREATE PROCEDURE GetAllRentals

AS

BEGIN

SELECT movie.\*, member.\*, movie\_rent.rental\_date, movie\_rent.return\_date

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

INNER JOIN member ON member.member\_id = movie\_rent.member\_id;

END;

4Get all movie titles with their average ratings:

CREATE PROCEDURE GetMovieAverageRatings

AS

BEGIN

SELECT title, AVG(rating) AS average\_rating

FROM movie

GROUP BY title;

END;

5Get all members along with the count of movies they have rented:

CREATE PROCEDURE GetMemberRentalsCount

AS

BEGIN

SELECT member.member\_id, COUNT(movie\_rent.movie\_id) AS rental\_count

FROM member

LEFT JOIN movie\_rent ON member.member\_id = movie\_rent.member\_id

GROUP BY member.member\_id;

END;

6Get information of all rented movies by members with return dates before today:

CREATE PROCEDURE GetOverdueRentedMovies

AS

BEGIN

SELECT movie.\*, member.\*, movie\_rent.rental\_date, movie\_rent.return\_date

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

INNER JOIN member ON member.member\_id = movie\_rent.member\_id

WHERE movie\_rent.return\_date < GETDATE() AND movie\_rent.return\_date IS NOT NULL;

END;

7Get information of all rented movies by members with return dates overdue (with parameter for the number of days overdue):

CREATE PROCEDURE GetOverdueRentedMovies1

@daysOverdue INT

AS

BEGIN

SELECT movie.\*, member.\*, movie\_rent.rental\_date, movie\_rent.return\_date

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

INNER JOIN member ON member.member\_id = movie\_rent.member\_id

WHERE DATEDIFF(DAY, movie\_rent.return\_date, GETDATE()) > @daysOverdue;

END;

8Get rental and movie information of all movies rented by a specific member (pass member\_id)

CREATE PROCEDURE GetMoviesRentedByMember1

@memberID INT

AS

BEGIN

SELECT movie\_rent.rental\_id, movie.\*

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

WHERE movie\_rent.member\_id = @memberID;

END;

9Get rental and movie information of all members who have rented a specific movie (pass movie\_id)

CREATE PROCEDURE GetMembersWhoRentedMovie

@movieID INT

AS

BEGIN

SELECT movie\_rent.rental\_id, movie\_rent.movie\_id, member.\*

FROM member

INNER JOIN movie\_rent ON member.member\_id = movie\_rent.member\_id

WHERE movie\_rent.movie\_id = @movieID;

END;

10Rent a movie to a member (pass all required parameters based on movie\_rent table)

CREATE PROCEDURE RentMovieToMember

@rentalID INT,

@memberID INT,

@movieID INT,

@rentalDate DATE,

@returnDate DATE

AS

BEGIN

INSERT INTO movie\_rent (rental\_id, member\_id, movie\_id, rental\_date, return\_date)

VALUES (@rentalID, @memberID, @movieID, @rentalDate, @returnDate);

END;

exec RentMovieToMember 6, 1, 3, '2024-04-07', '2024-04-14'

11Return a rented movie accepting/passing rental\_id and return\_date:

CREATE PROCEDURE ReturnRentedMovie

@rentalID INT,

@returnDate DATE

AS

BEGIN

UPDATE movie\_rent

SET return\_date = @returnDate

WHERE rental\_id = @rentalID;

END;

exec ReturnRentedMovie 5, '2024-04-13'

12.Get rental history of a member (passing member\_id to a temporary stored procedure)

CREATE PROCEDURE #GetRentalHistoryForMember

@memberID INT

AS

BEGIN

SELECT movie\_rent.member\_id, movie\_rent.rental\_date, movie\_rent.return\_date, movie.\*

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

WHERE movie\_rent.member\_id = @memberID;

END;

exec #GetRentalHistoryForMember 1

13Get information of all members who rented movies in a specific year (passing year to a temporary stored procedure)

CREATE PROCEDURE #GetMembersByRentalYear

@year INT

AS

BEGIN

SELECT DISTINCT member.\*

FROM member

INNER JOIN movie\_rent ON member.member\_id = movie\_rent.member\_id

INNER JOIN movie ON movie\_rent.movie\_id = movie.movie\_id

WHERE YEAR(movie\_rent.rental\_date) = @year;

END;

exec #GetMembersByRentalYear 2024

14Get all movies rented by a specific member within a date range (passing member\_id, start\_date, and end\_date to a temporary stored procedure)

CREATE PROCEDURE #GetMoviesRentedByMemberInRange

@memberID INT,

@startDate DATE,

@endDate DATE

AS

BEGIN

SELECT movie\_rent.member\_id, movie.\*

FROM movie

INNER JOIN movie\_rent ON movie.movie\_id = movie\_rent.movie\_id

WHERE movie\_rent.member\_id = @memberID

AND movie\_rent.rental\_date >= @startDate

AND movie\_rent.rental\_date <= @endDate;

END;

exec #GetMoviesRentedByMemberInRange 1, '2023-04-01', '2024-05-08'

Create 3 tables Programs, Courses, and Program\_Courses with appropriate attributes and then insert some records to each table.

CREATE TABLE Programs (

program\_id INT PRIMARY KEY,

program\_name VARCHAR(50) NOT NULL,

program\_description VARCHAR(50)

);

CREATE TABLE Courses (

course\_id INT PRIMARY KEY,

course\_name VARCHAR(50) NOT NULL,

course\_description VARCHAR(50)

);

CREATE TABLE Program\_Courses (

program\_id INT,

course\_id INT,

FOREIGN KEY (program\_id) REFERENCES Programs(program\_id),

FOREIGN KEY (course\_id) REFERENCES Courses(course\_id),

PRIMARY KEY (program\_id, course\_id)

);

-- Inserting records into Programs table

INSERT INTO Programs (program\_id, program\_name, program\_description)

VALUES

(1, 'Computer Science', 'Study of computation'),

(2, 'Electrical Engineering', 'Study of electrical systems'),

(3, 'Business Administration', 'Study of management');

-- Inserting records into Courses table

INSERT INTO Courses (course\_id, course\_name, course\_description)

VALUES

(101, 'Introduction to Programming', 'Basic concepts of programming'),

(102, 'Database Management', 'Study of organizing'),

(103, 'Microeconomics', 'Study of individual economic'),

(104, 'Digital Electronics', 'Study of digital circuits'),

(105, 'Marketing Principles', 'Study of marketing');

-- Inserting records into Program\_Courses table to establish many-to-many relationships

INSERT INTO Program\_Courses (program\_id, course\_id)

VALUES

(1, 101), -- Computer Science -> Introduction to Programming

(1, 102), -- Computer Science -> Database Management

(2, 104), -- Electrical Engineering -> Digital Electronics

(3, 103), -- Business Administration -> Microeconomics

(3, 105); -- Business Administration -> Marketing Principles

1Write an After Insert, Update, Delete Trigger on Programs Table to get fired in case of trying to insert, update, or delete the data on it. Then, test the trigger and show these added, updated, or deleted data.

CREATE TRIGGER AfterProgramChanges

ON Programs

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

IF EXISTS(SELECT \* FROM inserted)

BEGIN

PRINT 'Program data has been modified:';

SELECT 'New Program ID: ' + CAST(program\_id AS VARCHAR(10)),

'New Program Name: ' + program\_name,

'New Program Description: ' + program\_description

FROM inserted;

END;

IF EXISTS(SELECT \* FROM deleted)

BEGIN

PRINT 'Program data has been deleted:';

SELECT 'Deleted Program ID: ' + CAST(program\_id AS VARCHAR(10)),

'Deleted Program Name: ' + program\_name,

'Deleted Program Description: ' + program\_description

FROM deleted;

END;

END;

To test

INSERT INTO Programs (program\_id, program\_name, program\_description)

VALUES (4, 'Social Science', 'History studies')

delete from programs where program\_id = 4

update programs set program\_name = 'Computer Sciences' where program\_id = 1

2Write an After Insert, Update, Delete Trigger on Courses Table to get fired in case of trying to insert, update, or delete the data on it. Then, test the trigger and show these added, updated, or deleted data.

CREATE TRIGGER AfterCourseChanges

ON Courses

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

IF EXISTS(SELECT \* FROM inserted)

BEGIN

PRINT 'Course data has been modified:';

SELECT 'New Course ID: ' + CAST(course\_id AS VARCHAR(10)),

'New Course Name: ' + course\_name,

'New Course Description: ' + course\_description

FROM inserted;

END;

IF EXISTS(SELECT \* FROM deleted)

BEGIN

PRINT 'Course data has been deleted:';

SELECT 'Deleted Course ID: ' + CAST(course\_id AS VARCHAR(10)),

'Deleted Course Name: ' + course\_name,

'Deleted Course Description: ' + course\_description

FROM deleted;

END;

END;

To test

INSERT INTO Courses (course\_id, course\_name, course\_description)

VALUES (106, 'Java', 'JAVA programming')

delete from courses where course\_id = 106

update courses set course\_name = 'DBMS' where course\_id = 102

3Create a view of Programs table.

CREATE VIEW ProgramView

AS select \* from Programs

Then, write an Instead Of Insert Trigger on this view that when fired will prompt a message 'This operation is not allowed on Programs table'.

CREATE TRIGGER InsteadOfThis

ON ProgramView

INSTEAD OF INSERT

AS

BEGIN

PRINT 'This operation is not allowed on Programs table';

END;

To test

INSERT INTO dbo.ProgramView VALUES (4,'Medicine','Doctorate')

select \* from ProgramView

4Create a view of Program\_Courses table.

CREATE VIEW ProgramCoursesView

AS select \* from Program\_Courses

Then, write an Instead Of Insert Trigger on this that when fired will prompt a message 'This operation is not possible'.

CREATE TRIGGER InsteadOfThat

ON ProgramCoursesView

INSTEAD OF INSERT

AS

BEGIN

PRINT 'This operation is not possible';

END;

To test

INSERT INTO dbo.ProgramCoursesView VALUES (2, 103)

select \* from ProgramCoursesView

5Write a Trigger that avoids creating the tables in database and prompts an appropriate message. Then, test if the trigger functions properly.

CREATE TRIGGER trRestrictCreateTable

ON DATABASE

FOR CREATE\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT CREATE A TABLE IN THIS DATABASE'

ROLLBACK TRANSACTION

END

6Write a Trigger that avoids altering the tables in database and prompts an appropriate message. Then, test if the trigger functions properly.

CREATE TRIGGER trRestrictAlterTable

ON DATABASE

FOR ALTER\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT ALTER TABLES'

ROLLBACK TRANSACTION

END

7Write a Trigger that avoids dropping the tables in database and prompts an appropriate message. Then, test if the trigger functions properly.

CREATE TRIGGER trRestrictDropTable

ON DATABASE

FOR DROP\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT DROP TABLES'

ROLLBACK TRANSACTION

END

To test

create table aaa (number int)

drop table Program\_Courses

alter table programs add area int

8Write an After Insert, Update, Delete Trigger on Program\_Courses Table to get fired in case of trying to insert, update, or delete the data on it. Then, test the trigger and show these added, updated, or deleted data.

CREATE TRIGGER AfterProgramCoursesChanges

ON Program\_Courses

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

IF EXISTS(SELECT \* FROM inserted)

BEGIN

PRINT 'Program\_Courses data has been modified:';

SELECT 'Program ID: ' + CAST(program\_id AS VARCHAR(10)),

'Course ID: ' + CAST(course\_id AS VARCHAR(10))

FROM inserted;

END;

IF EXISTS(SELECT \* FROM deleted)

BEGIN

PRINT 'Program\_Courses data has been deleted:';

SELECT 'Deleted Program ID: ' + CAST(program\_id AS VARCHAR(10)),

'Deleted Course ID: ' + CAST(course\_id AS VARCHAR(10))

FROM deleted;

END;

END;

To test

INSERT INTO Program\_Courses (program\_id, course\_id)

VALUES (1,103)

delete from program\_courses where program\_id = 1 and course\_id = 103

select \* from Program\_Courses