P4

Stanley Inouye (ete4ec), Danny Know (msv7hq), Vaughn Scott (wyu7zt), Neal Langhorne (yhn5yh)

```
/***********************
 * - Procedure 1: AddUserRental
 * - Purpose: Adds a rental record when a user rents a movie
 ***********************/

CREATE PROCEDURE AddUserRental
@userID INT,
@movieID INT
AS
BEGIN
    IF (SELECT COUNT(*) FROM UserMovie WHERE movieID = @movieID) >=
(SELECT totalCopies FROM Movie WHERE movieID = @movieID)
    BEGIN
        PRINT 'No copies available for rental';
        RETURN;
    END

    INSERT INTO UserMovie (userID, movieID)
    VALUES (@userID, @movieID);
END;



/***********************
 * - Procedure 2: DeleteReview
 * - Purpose: Deletes a review for a specific movie by a specific user.
 ***********************/

CREATE PROCEDURE DeleteReview
@reviewID INT
AS
BEGIN
    DELETE FROM Review WHERE reviewID = @reviewID;
END;



/***********************
 * - Stored Procedure 3: UpdateUserRole
 * - Purpose: Updates a users role based off of the user id and role id.
***********************/
```

```
CREATE PROCEDURE UpdateUserRole
    @userID INT,
    @roleID INT
AS
BEGIN
    UPDATE UserRole
    SET roleID = @roleID
    WHERE userID = @userID;
END;




/**********************
 * - Function 1: GetNumberCopiesRented
 * - Parameter: movieID (int)
 * - Returns: int, number of copies of the movie with ID movieID that are
currently being rented
 * - How or why this would be used: this function would be used so that
site administrators can audit demand for movies (to determine if the movie
 *   should still be ‘on the shelf’) and ensure that the system is
functioning properly. For instance, the site administrator would want to
make sure that there are not more
 *   movies rented out than there are digital copies that the company
holds.
 **********************/

CREATE FUNCTION dbo.GetNumberCopiesRented
(
    @movieID int
)
RETURNS INT
AS
BEGIN
    DECLARE @NumRented INT;
    SELECT @NumRented = COUNT(movieID)
    FROM UserMovie
    WHERE movieID = @movieID

    RETURN ISNULL(@NumRented, 0)
END

GO

/**********************
 * - Function 2: GetMoviesMadeByProdComp
 * - Parameter: productionID (int)
 * - Returns: table, contains table with one column that holds the name(s)
of the movie(s) created by the production company with ID 'productionID'
 * - How or why this would be used: This could be used with filtered
search capabilities on the movie rental website, where a user might like a
particular production company
 *   and want to see what movies from that production company are
available on our platform
 **********************/

CREATE FUNCTION dbo.GetMoviesMadeByProdComp (
    @productionID int
)
RETURNS TABLE
AS
RETURN
(
    SELECT title
    FROM Movie
    WHERE productionID = @productionID
)

GO



/**********************
 * - Function 3: GetUserRentalCount
 * - Parameter: @userID INT
 * - Returns: INT (total number of rentals by the user)
 * - Purpose: Counts the total number of movies rented by a specific user.
Useful for administrators monitoring user activity and understanding
customer behavior.
 **********************/
CREATE FUNCTION dbo.GetUserRentalCount (@userID INT)
RETURNS INT
AS
BEGIN
    RETURN (SELECT COUNT(*) FROM UserMovie WHERE userID = @userID);
END;




/**********************
 * - View 1: View_AvailableMovies
 * - Purpose: Lists movies with available copies for rent (i.e., movies
that have less than a specified number of copies rented).
```

```
 **********************/
CREATE VIEW View_AvailableMovies AS
SELECT m.movieID, m.title, m.totalCopies - COUNT(um.movieID) AS
availableCopies
FROM Movie m
LEFT JOIN UserMovie um ON m.movieID = um.movieID
GROUP BY m.movieID, m.title, m.totalCopies
HAVING COUNT(um.movieID) < m.totalCopies;


/**********************
 * - View 2: View_UserReviews
 * - Purpose: Lists reviews made by each user, including their username,
movie title, rating, and comment.
 **********************/
CREATE VIEW View_UserReviews AS
SELECT u.username, m.title AS movieTitle, r.rating, r.comment
FROM AppUser u
JOIN Review r ON u.userID = r.userID
JOIN Movie m ON r.movieID = m.movieID;



/**********************
 * - View 3: MoviesWithRating
 * - Purpose: Returns a list of all movies and their average ratings.
 Can be sorted by ascending and descending after initial query.
**********************/
CREATE VIEW MoviesWithRating AS
SELECT
    m.movieId,
    m.title,
    m.runtime,
    m.productionID,
    m.originalLanguage,
    m.day,
    m.month,
    m.year,
    AVG(r.rating) as averageRating
FROM movie m
JOIN
    review r on m.movieID = r.movieID
GROUP BY
    m.movieId,
    m.title,
    m.runtime,
    m.productionID,
    m.originalLanguage,
    m.day,
    m.month,
    m.year

/***********************
 * - Trigger 1: Trigger_UpdateMovieAvgRating
 * - Action: After INSERT or UPDATE on the Review table
 * - Purpose: When a new review is added or updated, this trigger
recalculates and updates the average rating for the corresponding movie in
the Movie table.
***********************/

CREATE TRIGGER Trigger_UpdateMovieAvgRating
ON Review
AFTER INSERT, UPDATE
AS
BEGIN
    DECLARE @movieID INT;

    -- Get movieID from the inserted/updated review
    SELECT @movieID = i.movieID FROM inserted i;

    -- Update the average rating in the Movie table
    UPDATE Movie
    SET averageRating = (
        SELECT AVG(rating) FROM Review WHERE movieID = @movieID
    )
    WHERE movieID = @movieID;
END;
```

```
/**********************
 * - Encrypted Column 1: password in AppUser
**********************/
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '2N477b;~"GHe';

CREATE CERTIFICATE AppUserCert
WITH SUBJECT = 'Certificate for AppUser Password Encryption';

CREATE SYMMETRIC KEY AppUserPasswordKey
WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE AppUserCert;

ALTER TABLE AppUser
ADD encryptedPassword VARBINARY(128);

OPEN SYMMETRIC KEY AppUserPasswordKey
DECRYPTION BY CERTIFICATE AppUserCert;
UPDATE AppUser
SET encryptedPassword = EncryptByKey(Key_GUID('AppUserPasswordKey'),
password);
CLOSE SYMMETRIC KEY AppUserPasswordKey;




/* Creating nonclustered index on the 'Movie' table's title attribute */
CREATE NONCLUSTERED INDEX idx_MovieTitle ON Movie(title)




/**********************
 * - Non-Clustered Index 2: UserMovie_userID_movieID
 * - Table: UserMovie
 * - Columns: userID, movieID
 * - Purpose: Optimizes queries that retrieve a user’s rented movies or
check if a specific movie is rented by a user. This index will be useful
for quick lookups in user related queries, especially in a scenario where
a user’s rental history needs to be displayed frequently.
 **********************/
```

```
CREATE NONCLUSTERED INDEX idx_UserMovie_userID_movieID
ON UserMovie (userID, movieID);




/***********************
 * - Nonclustered Index 3: idx_user
 * - Purpose: allows all queries of any table with userID as a foreign key
to be queried more efficiently.
***********************/
CREATE NONCLUSTERED INDEX idx_user ON AppUser(userID)
```